IBM ILOG Scheduler V6.7

User's Manual

June 2009

Copyright notice

Trademarks

IBM, the IBM logo, ibm.com, WebSphere, ILOG, the ILOG design, and CPLEX are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Table of Contents

Meet IBM ILOG Scheduler

This User's Manual introduces you to IBM© ILOG© Scheduler, how it works and how to use it, and how to use the documentation to learn more. Examples from the basic to the complex are presented sequentially.

What is IBM ILOG Scheduler?

IBM ILOG Scheduler helps you develop problem-solving applications requiring the management of resources over time. Scheduler provides a library of re-usable and maintainable C++ classes that can be used just as they are, or extended to meet special needs. Those C++ classes define objects in the problem domain in a natural and intuitive way so that you can cleanly distinguish the problem *representation* from the problem *resolution*.

Scheduler is a C++ library based on IBM ILOG Solver, and like Solver, it employs all the facilities of object-orientation and constraint programming. Scheduler offers features specially adapted to solving problems in *scheduling and resource allocation*. There are, for example, classes of objects particularly designed to represent such aspects as the schedules themselves, the activities, the resources, and the temporal constraints. Scheduler offers you a workbench of tools that intuitively and naturally tackle the issues inherent in scheduling and allocation problems.

One of the main advantages of constraint programming is that it enables you to represent your problem explicitly in a natural and intuitive model, so that your problem *representation* serves simultaneously as a declarative *specification*. This congruence between the problem specification and the problem representation guarantees that the resolution of the constraints does, indeed, solve the problem as defined. In other words, there is no "slip" between the model of the problem and the implementation of its solution. Scheduler embodies this advantage of modeling and solving in a ready-to-use library of tools.

How Scheduler Works

Since Scheduler is a C++ library, you can use it directly in your own applications. There is no need to start Scheduler; there's no need to "get into" or "get out of" it either. It's not another programming language that you have to learn before you can make use of it. Nor is it a closed system that you have to find your way around. In fact, you can continue to use your preferred development environment while you build Scheduler into your applications. It's ready to go once you install the library on your platform.

When you use Scheduler, you link the library to your application. The command that you use for linking depends on your platform (that is, the combination of hardware and software you are using). In the standard distribution of the product, there is a `readme` file that explains where to find a sample project, and a `make` file containing the link command and other details appropriate to your platform.

You will use Scheduler to create a model of a scheduling or resource allocation problem. The model that is created belongs to an environment, and both the model and the environment are managed by IBM ILOG Concert Technology. After the model is complete, it is extracted to a solver engine, such as IBM ILOG Solver. All the examples in this book are based on extracting the model to Solver, and using Solver search methods to find a solution.

Typical Problems and Applications

Scheduling can be seen as the process of assigning *activities* to *resources* over *time*. Scheduling problems also require the management of minimal or maximal capacity constraints over time. While there is great diversity in scheduling problems, we can distinguish three main categories of problems:

- In pure *scheduling* problems, solving the problem consists of placing activities in *time*. It is already known which resources are demanded by which activities and in which quantities. The capacities of the resources are also known, although they may vary over time. The problem then is to place the activities in time, without ever exceeding the available capacity. A well-known example of a pure scheduling problem is the job-shop scheduling problem.

- In pure *resource allocation* problems, solving the problem consists of allocating *resources* to activities. It is already known *when* the activities are to be scheduled. The problem is to guarantee that the supply of resources at each point in time equals or exceeds the demand. An example of a pure resource allocation problem is the assignment of personnel to planes or trains.
- In *joint* scheduling and resource allocation problems, degrees of freedom exist for deciding which activities to perform and when to perform them, and for deciding which resources to make available for these activities.

There are many other scheduling problem types that can be solved using Scheduler, such as:

- joint scheduling and planning problems,
- joint scheduling and configuration problems,
- joint scheduling and sequencing problems.

Within these different problem types, a wide variety of constraints may need to be satisfied. Different environments are subject to different constraints which contribute more or less to the complexity of the problem. For example, one factory scheduling problem may involve only machines as resources, while another may also require the consideration of the abilities of human operators. The methods to schedule operators may be very different from those dealing with machines.

There also exists a wide variety in the size of scheduling problems. This size may vary from a few dozen activities to hundreds of thousands of activities. For complexity reasons, the methods that work well for small problems may not be applicable to bigger problems.

Several other properties that may be used to guide the search for a solution can also be quite different, not only from one environment to another but even from one day to another. For example, the presence and importance of bottleneck resources varies with the global load of a factory, and that load obviously may differ from day to day. The effectiveness and efficiency of scheduling rules depending on these kinds of properties differ from one situation to another.

Besides the diversity in the possible data of scheduling problems, different environments may require different performance from the algorithms solving the problems. For example, for some applications an acceptable response time for the construction of a schedule is not more than a few microseconds; other applications, however, will allow a response time of a few days. In contrast, for some applications, it suffices to have a “reasonable” solution, whereas others require near-optimal solutions.

The existence of such disparities implies that rigid scheduling procedures, designed to provide optimal or near-optimal schedules in particular circumstances, in general are not satisfactorily applicable in other circumstances. This led to the development of an extendible library—Scheduler—for representing scheduling and resource allocation problems.

Since its development, Scheduler has been used to create a variety of applications, including:

- personnel and equipment scheduling for installation, maintenance, and repair activities,
- finite-capacity production scheduling,
- project planning and scheduling,
- logistic resources scheduling.

Prerequisites

In order to use Scheduler effectively, you need to be familiar with your operating system. This manual assumes you already know how to create and manage files, and how to compile and execute a program.

Scheduler requires a working knowledge of C++. This manual assumes you already know how to program in C++ and that you will consult your favorite programming guide for C++ when you have questions in that area.

As you develop your own applications using Scheduler, you can use any C++ development environment that you find comfortable and familiar. Since it is a C++ library, Scheduler does not require any idiosyncratic or peculiar resources itself. Scheduler does not impose any extensions on C++ as a language, so you do not have to learn a new syntax or a new way of doing things.

The standard distribution comes with `make` files that you can copy and adapt to your own environment. The standard distribution also includes appropriate project files, in case you prefer a graphic user interface for compiling and linking.

Using This Manual

Think of this manual as a programmer's guide to using Scheduler. Scheduling problems are tackled using code from Scheduler, Concert Technology, and Solver. The solutions are fully discussed and the code completely explained. Since most chapters in this manual present at least one scheduling problem, you will find a wide variety of techniques and solutions throughout the manual, and this will help you adapt examples to your own scheduling problems. The examples are generally organized in order of increasing complexity. We recommend that the typical user follow the examples in order, as a cumulative knowledge base is then progressively built.

The manual is organized into four parts. Part I, *Getting Started with Scheduler*, introduces you to Scheduler with basic problems using simple resources and activities. Part II, *Advanced Concepts*, continues with more complex examples using a variety of resources and constraints. These examples, although more complex, will still be of interest to most Scheduler users. Part III, *Local Search in Scheduler*, presents detailed examples of ways to use, guide, and modify search in Scheduler. Part IV, *Integrated Applications*, discusses principles, applications, and programming techniques of interest to a narrower audience.

Three chapters discuss some Scheduler features, concepts, and tools without presenting a code example. Chapter 1, *Scheduler Building Blocks*, introduces the basic building blocks used in Scheduler to create applications and solve scheduling problems. Chapter 2, *Searching for Solutions*, introduces the predefined search goals and tools available in Scheduler. Chapter 11, *Advanced Features and Concepts*, is an introduction to the advanced features of Scheduler, including state resources, energy resources, and transition expressions. All other chapters deal explicitly with real scheduling problems, and are discussed in the next section.

Examples in This Manual

With minor variations, each example adheres to the following format.

- The chapter starts with a discussion of the main concepts illustrated by the example.
- The introduction is followed by a precise description of the problem to be solved. This description resembles the sort of verbal description that you should write before you design the model for your own problem.
- Next is a presentation of the proposed solution, generally divided in two sections. The first section is for the definition of the problem and the design of an appropriate model; the second section is for the problem solution.

  This division into problem *definition* and problem *solution* reflects the well-known constraint programming principle that urges a clean separation between the two. We strongly encourage you to follow this principle.
- Next, the entire program is listed, along with the solutions to a few instances of the problem.
- In summary, a few remarks conclude the example solution under consideration.

In case you prefer studying code online, the entire source for every example is provided in the standard distribution of Scheduler. The following table indicates the C++ file that corresponds to each example.

Table 1 *Examples Presented in this Manual*

Example	PC source	
Getting Started with Scheduler		
Using the Building Blocks (first example)	gsBasic.cpp	
Using the Building Blocks (second example)	gsUnary.cpp	
Using Discrete Resources (first example)	gsDisc.cpp	
Using Discrete Resources (second example)	gsBudget.cpp	
Adding Transition Times	gsTTime.cpp	

Table 1 *Examples Presented in this Manual*

Example	PC source	
Adding Breaks	gsBreak.cpp	
Adding Integral and Functional Constraints	gsEff.cpp	
Adding Alternative Resources	gsAlt.cpp	
Using Reservoirs	gsReserv.cpp	
Using the Trace Facilities	gsTrace.cpp	
Advanced Concepts		
Using Scheduler with Solver	greedy.cpp	
More Advanced Problem Modeling with Concert Technology	cassign.cpp	
Bridge Construction Scheduling	bridge.cpp	
Job-Shop Scheduling (Minimize Makespan)	jobshopm.cpp	
Dichotomizing Binary Search	jopshopd.cpp	
Ship-Loading Scheduling	ship.cpp	
Ship-Loading Scheduling (Decompose Problem)	shipdec.cpp	
Moving Activities from Discrete to Unary Resources	assign.cpp	
Working with Transition Times	jobshopt.cpp	
Using Transition Times and Costs	tcost.cpp	
Strong Propagation on Reservoirs using the Balance Constraint	balpos.cpp	
Scheduling with State Resources	trolley1.cpp	
Scheduling with Durable Resources	durable.cpp	
Using Trace on an Overconstrained Problem	relgreed.cpp	
Local Search in Scheduler		
Job-Shop Problem with Shuffling	jobshopshuf.cpp	
Tabu Job-Shop Problem	jobshopn1.cpp	
Tabu Job-Shop Problem with Alternatives	altls.cpp	

Table 1 *Examples Presented in this Manual*

Example	PC source	
Large Neighborhood Search for the Jobshop Problem with Alternatives	altlns.cpp	
Integrated Applications		
Randomizing Algorithm	jobshopr.cpp	
Handling Overconstrained Problem	bridgeov.cpp	
Extending Transition Cost Usage	maintain.cpp	
Scheduling with State Resources with Transition Times	trolley2.cpp	
Scheduling with State Resources with Transition Times and Limited Capacity	trolley3.cpp	
Representation of Time-Varying Capacity	capvar.cpp	
Scheduling with Alternative Resources	alttext.cpp	
Incremental Scheduling: Using Multiple Problem SubModels	resched2.cpp	
Writing Multi-Threaded Applications with Durable Resources	durablem.cpp	

Naming and Notations

Some notation and naming conventions used in this manual:

- Important ideas are *italicized* the first time they appear.
- The names of types, classes, and functions in the libraries begin with `Ilo` or `Ilc`, and appear in the documentation in `Courier typeface`; for example, `IloActivity` and `IlcScheduler`.
- The names of member functions begin with a lower case letter and appear in `Courier typeface`; for example, `getName()`. Accessors generally begin with the keyword `get`. Accessors for Boolean data members begin with `is`. Modifiers begin with `set`.
- A lower-case letter also begins the first word in names of arguments and instances. Other words in the identifier begin with an upper-case letter. For example, an activity in a scheduling problem might be called `IloActivity makeHouse`.
- Samples of C++ code appear in `Courier typeface`.

- Intervals of time are defined by two values, `timeMin` and `timeMax`. By convention, an interval is closed on the left and open on the right, represented this way: `[timeMin timeMax)`, so that time-varying parameters can be unambiguously defined for any given `time`.

Using Other Manuals in the Library

As you study the examples in this manual, you'll probably refer frequently to the reference manuals for Scheduler, Concert Technology, and Solver. They fully document the predefined classes—the objects, data members, and member functions. These reference manuals explain certain concepts more formally, such as breaks, ranking, rounding, transition expressions, and disjunctive and sequence constraints.

The *Scheduler Reference Manual* includes complete descriptions with class synopses, function declarations, and explanations. It outlines algorithms implemented by the Scheduler library and provides the last word on any given Scheduler topic.

The first part of the *Scheduler Reference Manual* documents the modeling components of Scheduler. The second part documents a layer called *Scheduler Engine*. When Scheduler is used with Solver, the extraction of the Scheduler modeling classes first creates corresponding instances of Solver-specific classes of Scheduler. This layer is called Scheduler Engine.

All of these manuals are delivered with the standard distribution.

IBM software support handbook

This guide contains important information on the procedures and practices followed in the service and support of your IBM products. It does not replace the contractual terms and conditions under which you acquired specific IBM Products or Services. Please review it carefully. You may want to bookmark the site so you can refer back as required to the latest information. We are interested in continuing to improve your IBM support experience, and encourage you to provide feedback by clicking the Feedback link in the left navigation bar on any page. The "IBM Software Support Handbook" can be found on the web at

http://www14.software.ibm.com/webapp/set2/sas/f/handbook/home.html

Accessing software support

When calling or submitting a problem to IBM Software Support about a particular service request, please have the following information ready:

IBM Customer Number

The machine type/model/serial number (for Subscription and Support calls)

Company name

Contact name

Preferred means of contact (voice or email)

Telephone number where you can be reached if request is voice

Related product and version information

Related operating system and database information

Detailed description of the issue

Severity of the issue in relationship to the impact of it affecting your business needs

Contact via web

Open service requests is a tool to help clients find the right place to open any problem, hardware or software, in any country where IBM does business. This is the starting place when it is not evident where to go to open a service request.

Service Request (SR) tool offers Passport Advantage clients for distributed platforms online problem management to open, edit and track open and closed PMRs by customer number. Timesaving options: create new PMRs with prefilled demographic fields; describe problems yourself and choose severity; submit PMRs directly to correct support queue; attach troubleshooting files directly to PMR; receive alerts when IBM updates PMR; view reports on open and closed PMRs.

You can find information about assistance for SR at http://www.ibm.com/software/support/help-contactus.html.

System Service Request (SSR) tool is similar to Electronic Service request in providing online problem management capability for clients with support offerings in place on System i, System p, System z, TotalStorage products, Linux, Windows, Dynix/PTX, Retail, OS/2, Isogon, Candle on OS/390 and Consul z/OS legacy products.

IBMLink - SoftwareXcel support contracts offer clients on the System z platform the IBMLink online problem management tool to open problem records and ask usage questions on System z software products. You can open, track, update, and close a defect or problem record; order corrective/preventive/toleration maintenance; search for known problems or technical support information: track applicable problem reports: receive alerts on high impact problems and fixes in error; and view planning information for new releases and preventive maintenance.

Contact via phone

If you have an active service contract maintenance agreement with IBM , or are covered by Program Services, you may contact customer support teams via telephone. For individual countries, please visit the Technical Support section of the IBM Directory of worldwide contacts via http://www.ibm.com/planetwide/.

Part I

Getting Started with Scheduler

The chapters in Part I introduce you to using Scheduler.

Chapter 1, *Scheduler Building Blocks*, introduces the basic building blocks used in Scheduler to create applications and solve scheduling problems.

Chapter 2, *Searching for Solutions*, introduces the predefined search goals and tools available in Scheduler.

Tutorials begin in Chapter 3, *Using the Building Blocks*, and continue through Chapter 10, *Using the Trace Facilities*. In each tutorial we tackle a problem, showing you how to exploit the description of the problem to design an appropriate model. With that model as a basis, we show how to define the problem in Scheduler by declaring the unknowns in the problem and posting its constraints. Then we walk through a solution of the problem and show the results.

When you finish the tutorial problems of Part I, you'll be familiar enough with Scheduler to plan your own work with it. You can then continue with Part II, which goes into greater detail about designing models, and provides more complex examples using advanced features of Scheduler.

C H A P T E R

1

Scheduler Building Blocks

Scheduling is, basically, the act of creating a schedule—a timetable for planned occurrences. Generally, scheduling consists of allocating resources to activities over time. A scheduling problem can be viewed as a constraint satisfaction problem or as a constrained optimization problem, but regardless of how it is viewed, a scheduling problem is defined by:

- a set of *activities*—tasks or work to be completed—to schedule;
- a set of *temporal constraints*—definitions of possible relationships between the start and end times of the activities—to link the activities together;
- a set of *resources*—objects such as workers, machines, vehicles, supplies, raw materials, etc., which add value to a product or service in its creation, production, or delivery—to allocate to the activities;
- a set of *resource constraints*—definitions of demands of the activities upon the resources—to link the activities to the resources.

This chapter provides a theoretical overview of the concepts implemented in Scheduler to facilitate the representation of scheduling problems. After reading this chapter, you'll be prepared to understand technical terms encountered in the examples that occur in later chapters. If you prefer, you can start with the examples and return to this overview later when you have questions about the terminology or when you are ready to begin representation of your own scheduling or allocation problem.

Scheduler Modeling and Solving

The activities, resources, temporal constraints, and resource constraints that define a scheduling problem are C++ classes and functions of Scheduler. Scheduler is a modeling-based language; that is, you build a model of the scheduling or resource allocation problem. The model that you build (an instance of the class `IloModel`), and each Scheduler class instance pertaining to that model (each activity, resource, and so forth), belong to an environment, an instance of the class `IloEnv`. IBM ILOG Concert Technology manages the model and the environment. An environment may contain one or more models, and each model can use the various activities and resources available in that environment. (For example, you can use multiple models to simulate distributed scheduling.) After the model is complete, it is extracted to a solver engine, such as IBM ILOG Solver. Building a successful model depends on correctly implementing Scheduler object classes.

Scheduler Object Classes

Scheduler offers a simple object model for the representation of scheduling and resource allocation problems. Figure 1.1 is a diagram of the main Scheduler object classes.

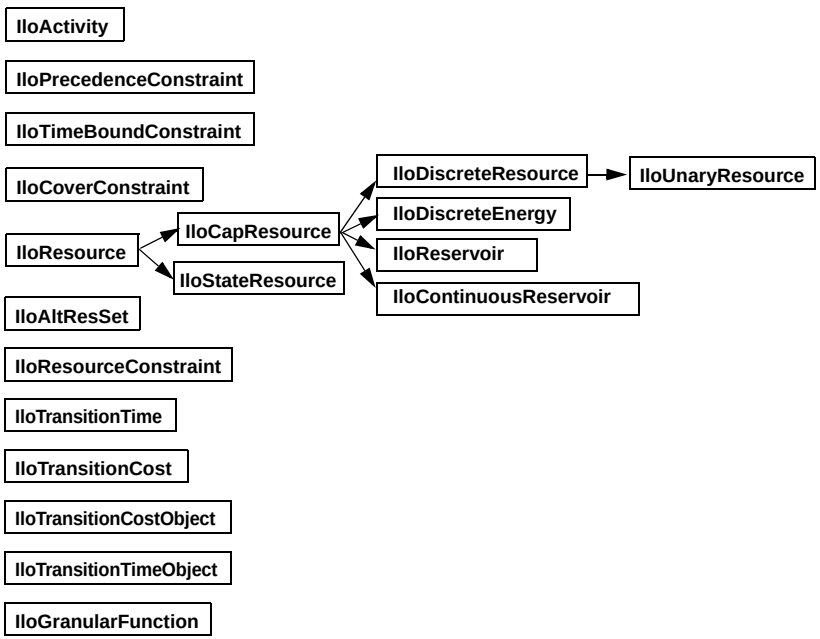


Figure 1.1 *Scheduler Object Classes*

Activities

An activity is the task or work to be performed in a schedule: filling bottles in a schedule for a beverage company, for example, or caring for patients in a schedule for a medical clinic, etc. Activities have a duration; they execute over a specific interval of time in a schedule, which may be subject to temporal constraints, and they require resources. For example, in a beverage company, the activity of filling bottles requires as resources a quantity of the appropriate beverage, the correct size bottles, and the filling machinery. In a medical clinic, an appropriate schedule of activities might require nurses, medical supplies, and treatment rooms as resources.

`IloActivity` is the Scheduler activity class. It defines the time interval on which the activity is processed. An instance of `IloActivity` includes a start time variable, an end or completion time variable, and a processing time variable. The *processing time* of an activity is the elapsed time during which the activity is effectively processed by the resource.

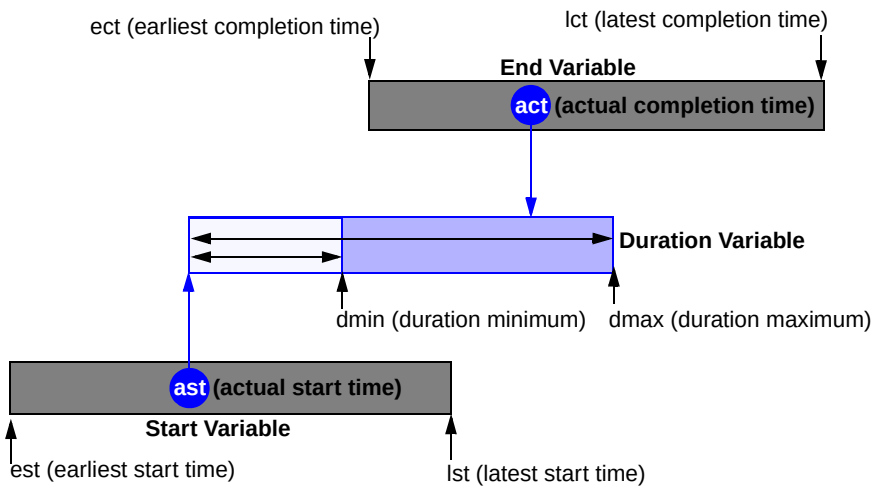


Figure 1.2 *Variables Associated with the Class* `IloActivity`

By default, the earliest possible start time of an activity is the time origin of the scheduling environment it belongs to, and the latest possible end time of an activity is the time horizon of its scheduling environment. Both the origin and the horizon of a schedule can be set using the `IloSchedulerEnv` object. The duration of an activity can be 0.

An instance of `IloActivity` is non-breakable by default. A non-breakable activity is an activity which executes without suspension from its start time to its end time, and which may require some resources throughout its execution. With a non-breakable activity, the processing time is equal to the duration.

If the activity is set to be breakable (with the member function `setBreakable`), then the activity can be suspended during resource breaks and *only* during resource breaks. The activity cannot be suspended by other activities. (Resource breaks are discussed in more detail in Breaks.) For breakable activities, the processing time can differ from the duration of the activity, since the activity can be suspended by one or more breaks. During a break, the activity is not processed, but the duration grows longer. The processing time of a breakable activity is the duration minus the sum of the durations of the breaks that suspend it.

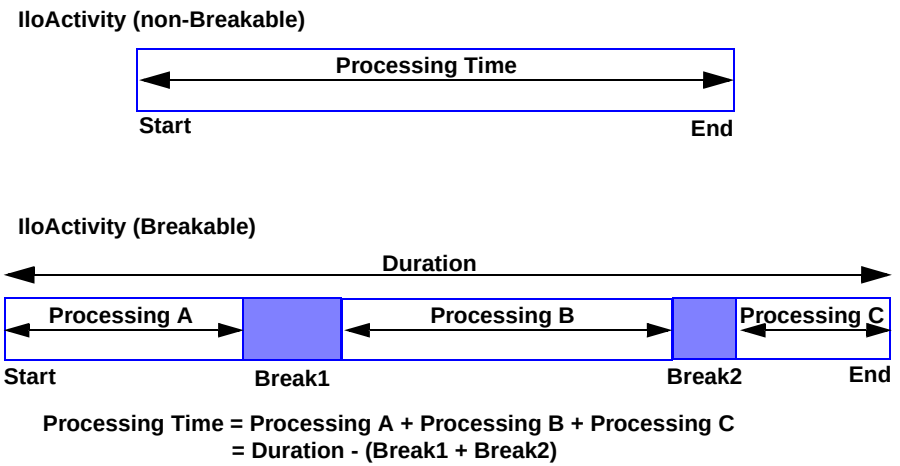


Figure 1.3 *Activity Processing Time and Duration*

To represent other classes of activities, you need to define new C++ classes. For example, a composite activity, built from several subactivities, might be defined as an instance of a new (user-defined) class containing a list of activities among its data members.

Activities are linked to other activities by temporal constraints. Activities are linked to resources by resource constraints. We'll say more about resources later, but for now, we should note that resources usually have a *capacity* that can be required by activities. The capacity that is required by an activity can be represented in two ways: as a constant or as a constrained Solver variable.

Temporal Constraints

In Scheduler there are two types of temporal constraints. *Precedence constraints*, instances of the class `IloPrecedenceConstraint`, are used to specify when one activity must start or end with respect to the start or end time of another activity. *Time-bound constraints*, instances of the class `IloTimeBoundConstraint`, are used to specify when one activity

must start or end with respect to a given time. All temporal constraints must be added to the model.

Precedence Constraints

Precedence constraints restrict the order of activities. They constrain an activity to start or end before, at, or after the start or end time of another activity.

Precedence constraints are created through member functions of the class `IloActivity`. They must be added to the model of the associated scheduling problem to be considered in the search for solutions.

Precedence constraints may involve the concept of *delay*. Delay is an amount of time that must elapse between the two variables involved in the precedence constraint. If delay is negative, it indicates the inverse of the maximal duration allowed to elapse between the two variables. In other words, endpoint2 can occur before endpoint1, but the difference between them cannot exceed -delay. The delay can be a numerical constraint or a numerical variable.

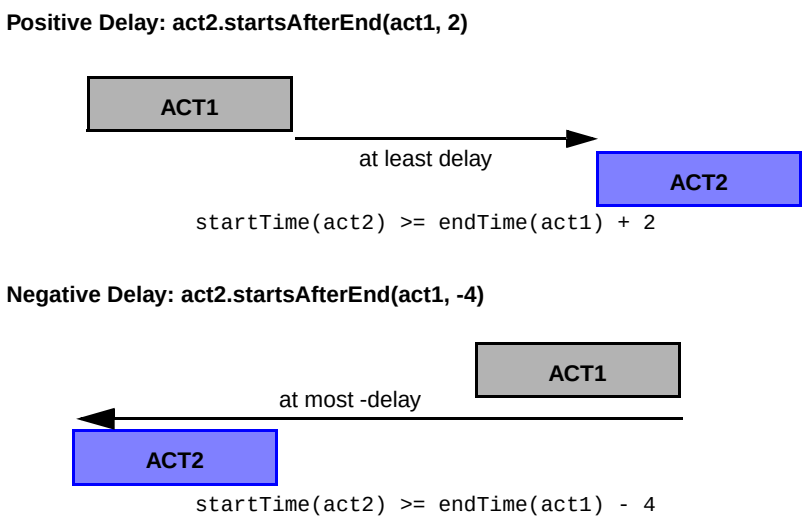


Figure 1.4 *Positive and Negative Delay*

If `act1` and `act2` denote activities, and `delay` is a number (`0` by default), then:

- `act2.endsAfterEnd(act1, delay)` constrains at least the given `delay` to elapse between the end of `act1` and the end of `act2`. It imposes the inequality `endTime(act2) >= endTime(act1) + delay`.

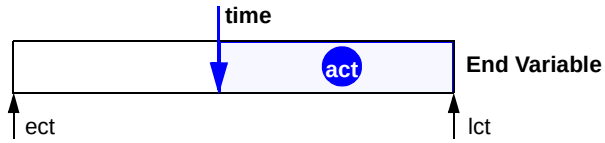
- `act2.endsAfterStart(act1, delay)` constrains at least the given `delay` to elapse between the start of `act1` and the end of `act2`. It imposes the inequality `endTime(act2) >= startTime(act1) + delay`.
- `act2.endsAtEnd(act1, delay)` constrains the given `delay` to separate the end of `act1` and the end of `act2`. It imposes the equality `endTime(act2) == endTime(act1) + delay`.
- `act2.endsAtStart(act1, delay)` constrains the given `delay` to separate the start of `act1` and the end of `act2`. It imposes the equality `endTime(act2) == startTime(act1) + delay`.
- `act2.startsAfterEnd(act1, delay)` constrains at least the given `delay` to elapse between the end of `act1` and the start of `act2`. It imposes the inequality `startTime(act2) >= endTime(act1) + delay`.
- `act2.startsAfterStart(act1, delay)` constrains at least the given `delay` to elapse between the start of `act1` and the start of `act2`. It imposes the inequality `startTime(act2) >= startTime(act1) + delay`.
- `act2.startsAtEnd(act1, delay)` constrains the given `delay` to separate the end of `act1` and the start of `act2`. It imposes the equality `startTime(act2) == endTime(act1) + delay`.
- `act2.startsAtStart(act1, delay)` constrains the given `delay` to separate the start of `act1` and the start of `act2`. It imposes the equality `startTime(act2) == startTime(act1) + delay`.

Time-Bound Constraints

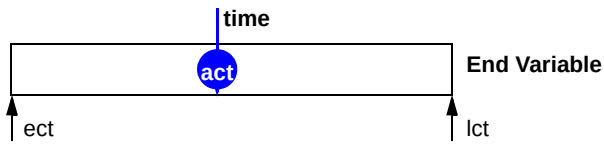
Time-bound constraints constrain an activity to start or end before, at, or after a given time. They are created by a member function of `IloActivity` and must be added to the model of the associated scheduling problem to be considered in the search for solutions. The time argument can be a numerical constant or a numerical variable.

The six time-bound constraints are described in the following diagrams. In the diagrams, `ect` and `lct` mean earliest and latest possible completion times, `est` and `lst` mean earliest and latest possible start times. These are the earliest and latest times available to the activity before the time-bound constraints are applied. `act` and `ast` mean actual completion and start times. The shaded area in the examples is the domain of the end variable.

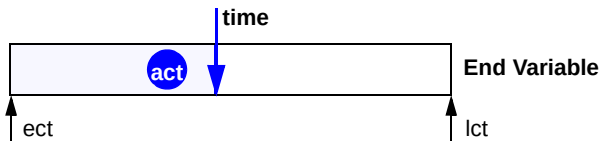
- `act1.endsAfter(IloNumVar time);` constrains `act1` to end after or at `time`.



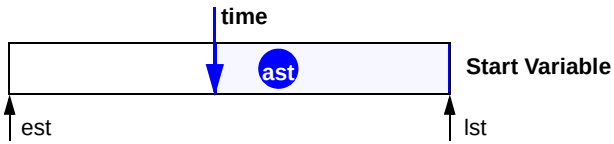
- `act1.endsAt(IloNumVar time);` constrains `act1` to end at `time`.



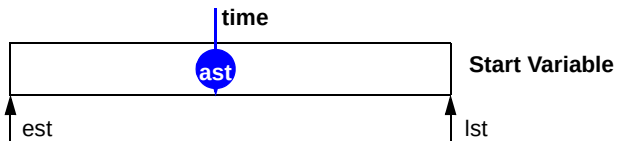
- `act1.endsBefore(IloNumVar time);` constrains `act1` to end before or at `time`.



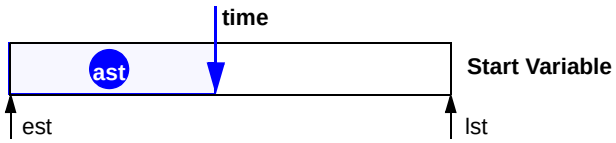
- `act1.startsAfter(IloNumVar time);` constrains `act1` to start after or at `time`.



- `act1.startsAt(IloNumVar time);` constrains `act1` to start at `time`.



- `act1.startsBefore(IloNumVar time);` constrains `act1` to start before or at `time`.



Resources

Activities in a scheduling problem usually require *resources*. For example, building a house may require resources of money, laborers, tools, and materials.

Scheduler provides several classes to represent resources. The class `IloResource` is the root class for all the resources managed in Scheduler. `IloResource` has two subclasses, `IloStateResource` and `IloCapResource`.

An instance of the class `IloStateResource` represents a resource of infinite capacity whose *state* can vary over time. Throughout its execution, each activity may require a state resource to be in a given state (or in any of a given set of states). Consequently, two activities may not overlap if they require incompatible states during their execution.

The class `IloCapResource`, which is the root class for all resources having some sort of capacity, has five subclasses.

- `IloDiscreteResource`: An instance of `IloDiscreteResource` represents a resource of finite discrete capacity. *Discrete* means that capacity is defined to be a positive integer number. Capacity varies with time: at any time *t*, the capacity represents the number of copies or instances of the resource that are available; examples include the number of milling machines available in a manufacturing shop, or the number of bricklayers at work on a construction site.

  Each activity may require some amount of the resource capacity, for example, one milling machine or three bricklayers. This requirement is represented by resource constraints. Propagation of resource constraints entails an update of the earliest and latest start and end times of activities.

- `IloUnaryResource`: An instance of `IloUnaryResource` represents a resource whose capacity is one. An instance of `IloUnaryResource` is either occupied or free. No more than one activity can be executed at a given time. The class `IloUnaryResource` is a subclass of `IloDiscreteResource`.

- `IloDiscreteEnergy`: An instance of the class `IloDiscreteEnergy` is divided into time buckets (for example, minutes, hours, months, years) that contain a certain amount of energy (for example, watt-hours or human-months) that can be made available over

those intervals. The available energy is used by the activities defined on the resource, and as a consequence capacity constraints on the discrete energy resource use energy rather than capacity.

- `IloReservoir`: An instance of `IloReservoir` represents a resource that can be dynamically replenished by producing activities. The capacity of an instance of `IloReservoir` can be simultaneously required and provided by activities. This class can be used to model situations in which certain types of activities consume capacity from the reservoir, while others produce capacity.

  A reservoir must not overflow its *maximal capacity* or underflow its *minimal capacity.*

- `IloContinuousReservoir`. An instance of `IloContinuousReservoir` represents a resource which activities can either fill or empty in a continuous and linear process between the start and the end of the activity. If the duration of the activity is null, the filling (or emptying) process is instantaneous (so not continuous). The maximum and minimum levels of a continuous reservoir can vary over time.

Alternative Resources

An *alternative resource set* contains two or more resources with equivalent capabilities. An activity may require or provide only one resource from this set of alternative resources during its execution.

The class `IloAltResSet` is designed for use in resource allocation problems. An instance of `IloAltResSet` is an aggregate of `IloResource` instances. The resources of a set must be instances of the class `IloCapResource`.

An activity may only require or provide one of the resources of an instance of `IloAltResSet`, without specifying which one.

Breaks

A resource *break* is a time interval during which the resource is not available for any activity. For example, breaks could include weekend or vacation time for workers, or maintenance time for machines. Breaks have a start time, an end time, and a duration (which can be either positive or null). Null duration breaks are used to define times during which no activity can be in process on a resource. Periodic breaks—those that reoccur at specified intervals—can also be defined.

By default, the processing time of an activity cannot overlap with the breaks defined on the resource it requires. However, it is possible to create activities that can be fully or partially processed inside a break (overlap the break). Constraints can be posted on the duration of overlap allowed.

The class `IloIntervalList` is provided for defining a list of breaks. Break lists can be shared between resources. The class `IloActivityBreakParam` is provided for changing the default behavior of activities with respect to breaks.

Breakable instances of `IloActivity` can be suspended during resource breaks. For a given instance of a breakable `IloActivity`, some breaks may be allowed to suspend the activity while others may not. A break is said to be disjunctive with respect to a breakable activity if the activity cannot be suspended by the break. That is, the activity must execute entirely before or entirely after the break. By default, only breaks with a duration of zero are considered as disjunctive.

Resource Constraints

Resource constraints—instances of the class `IloResourceConstraint`—link an activity, a resource, and some required capacity.

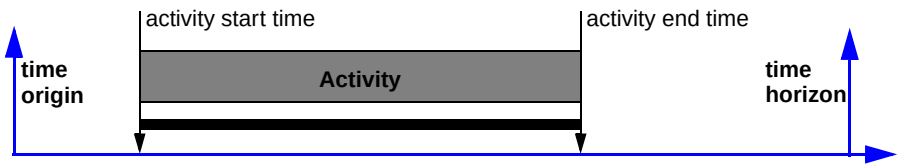
For example, the primary resource constraint for instances of `IloCapResource`, `act.requires(resource, capacity, time extent)`, constrains `act` to require `capacity` of `resource` for the given `time extent`. The required capacity can be a positive integer number (`IloNum`) or a variable (`IloNumVar`). Zero capacity is allowed, in which case the resource constraint does not change the availability of the resource.

Resource constraints are defined for other resources, such as instances of `IloStateResource` or `IloContinuousReservoir`. Refer to the *IBM ILOG Scheduler Reference Manual* for more information.

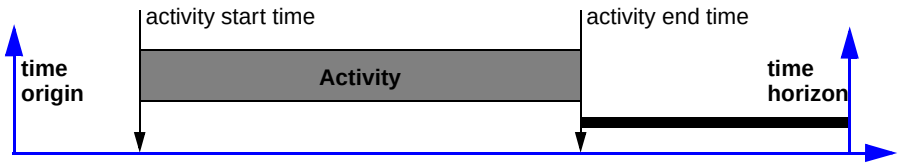
Time Extents

By default, instances of `IloActivity` use the resource from the start time of the activity to the end time of the activity. Breakable instances of `IloActivity` also use the resource throughout their duration, including break times. It is possible to specify a time range different from the default by declaring a time extent. The time extent defines when the invoking activity requires the resource, given its start and end times.

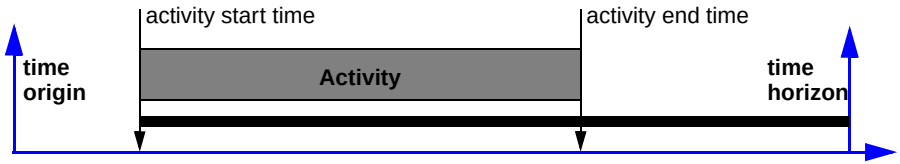
- `IloFromStartToEnd` indicates the activity requires the resource at all times from its start time to its end time. This is the default time extent.



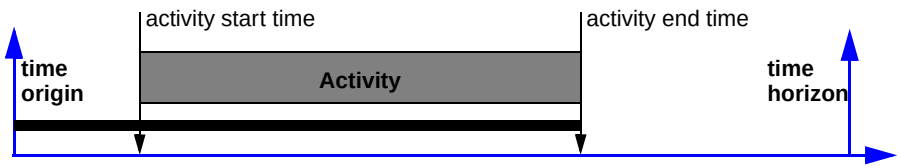
- `IloAfterEnd` indicates the activity requires the resource at all times after its end time.



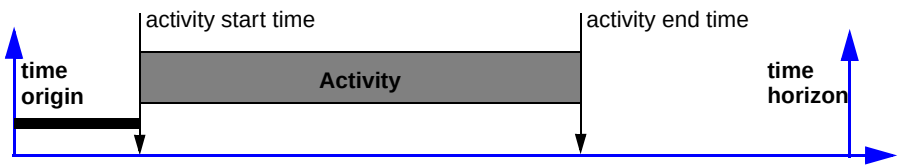
- `IloAfterStart` indicates the activity requires the resource at all times after its start time.



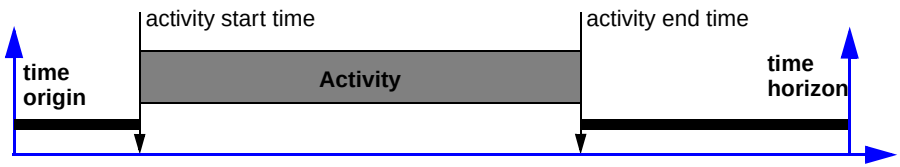
- `IloBeforeEnd` indicates the activity requires the resource at all times before its end time.



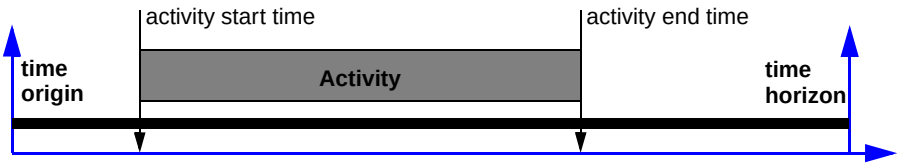
- `IloBeforeStart` indicates the activity requires the resource at all times before its start time.



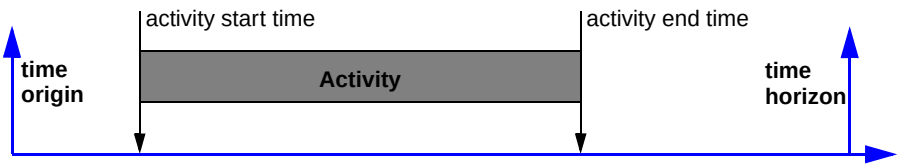
- `IloBeforeStartAndAfterEnd` indicates the activity requires the resource at all times before its start time and after its end time.



- `IloAlways` indicates the activity requires the resource at all times. This time extent can be used to optimize the maximal available capacity of a resource.



- `IloNever` indicates the activity requires the resource at no time.



Maximal and Minimal Capacity Constraints

We distinguish three main families of scheduling problems, depending on the degrees of freedom that exist for positioning resource supply and resource demand intervals over time. In pure sequencing problems (for example, job-shop machine scheduling), the *maximal capacity* of each resource is defined, and the problem consists of positioning resource-demanding activities over time, without ever exceeding the available capacity. In pure resource allocation problems (for example, allocation of personnel to planes or trains), the demand for each resource is defined, and the problem consists of allocating resources in time to guarantee that the supply always equals or exceeds the demand. In joint problems, degrees of freedom exist for deciding both which activities to perform and when, and which resources to make available for these activities.

In Scheduler, there are two ways to handle these needs. When the resource has a maximal capacity that cannot be exceeded, one can use the member functions `setCapacityMax`

(with `IloDiscreteResource`), `setEnergyMax` (with `IloDiscreteEnergy`) or `setLevelMax` (with `IloReservoir` and `IloContinuousReservoir`). When the problem is defined in terms of capacity that must be available over certain time intervals, one can use the member functions `setCapacityMin` (with `IloDiscreteResource`), `setEnergyMin` (with `IloDiscreteEnergy`) or `setLevelMin` (with `IloReservoir` and `IloContinuousReservoir`).

In general (but this rule allows exceptions), on the resources of a pure sequencing problem only *maximal capacity constraints* are defined. For example, in a manufacturing environment the activities require machines, the capacity of which is known to be bounded over time. Conversely, on the resources of a pure resource allocation problem only *minimal capacity constraints* are defined. For example, work shifts in a hospital require that the activities representing the presence of nurses are scheduled in such a way that minimal capacity levels are met. Examples follow.

Example—Maximal Capacity Constraints

To illustrate the use of maximal capacity constraints, let's take as an example a machine shop where groups of machines have to be serviced or maintained regularly, and consequently each machine is not available for production during its maintenance period.

The following function creates a group of `n` machines, `m` of which are not available over the interval `[maintenanceStart maintenanceEnd)`.

```
IloDiscreteResource MakeMachineGroup(IloEnv env,
                                     IloNum n,
                                     IloNum m,
                                     IloNum maintenanceStart,
                                     IloNum maintenanceEnd) {
   IloDiscreteResource group(env, n);
   group.setCapacityMax(maintenanceStart, maintenanceEnd, n - m);
   return group;
}
```

Example—Minimal Capacity Constraints

To illustrate the use of minimal capacity constraints, let's consider the problem of scheduling nurses for surgical operations.

The following function creates a group of `n` nurses, `m` of which must be present over the interval `[operationStart operationEnd)`.

```
IloDiscreteResource MakeNurseGroup(IloEnv env,
                                   IloNum n,
                                   IloNum m,
                                   IloNum operationStart,
                                   IloNum operationEnd) {
   IloDiscreteResource group(env, n);
   group.setCapacityMin(operationStart, operationEnd, m);
   return group;
}
```

In this example, a certain number of nurses are needed during an operation, so the schedule has to provide that minimal capacity throughout the interval.

Enforcement Levels

The enforcement level allows specifying how much effort the solver will expend enforcing a given resource constraint on a given resource. All enforcement levels ensure that any solution found by the scheduler will satisfy the type of constraint to the associated level. `IloBasic` is the default enforcement level. There are other enforcement levels that represent degrees of enforcement lower or higher than the `IloBasic`; for example, `IloLow` or `IloHigh`. Each level represents a certain degree of effort spent by the scheduler to enforce constraints.

Transitions

Costs and delays may arise when successively executing activities on the same resource (for example, there might be setup/teardown costs to prepare the resource for the next activity). The Scheduler classes `IloTransitionCost` and `IloTransitionTime` are used to model these expenses and delays.

IloTransitionCost

In Scheduler, transition cost is defined as the cost between an activity and the activity that will execute next to it on a unary resource. These costs may be related to modifications to the resource that require manpower, material, and energy, such as adjusting or purging a machine. In addition, Scheduler lets you define a setup cost for the activity that starts the usage of the resource, and a teardown cost for the activity that ends the usage of the resource. `IloTransitionCost` is used to model these costs that may occur for using an `IloUnaryResource`.

IloTransitionTime

Given two activities, the transition time between them is the time that must elapse between the end of the first and the start of the second when both are using the same resource. The class `IloTransitionTime` is used to model this delay.

C H A P T E R

2

Searching for Solutions

Scheduling can be viewed either as a constraint satisfaction problem or as a constrained optimization problem. When we think of scheduling as a constraint satisfaction problem, our aim is to find a schedule that satisfies the constraints, whatever they may be. When we think of scheduling as an optimization problem, our aim is to find a schedule that is optimal or close to optimal with respect to a given optimization criterion. The optimization criteria usually relate to time, capacity, and sequence: typically the makespan, tardiness, peak capacity, or transition cost.

- The *makespan* is the amount of time that elapses between the beginning of the first activity and the end of the last activity (i.e., the total amount of time needed to finish the project).
- The *tardiness* may relate to minimizing the "lateness" of any number of activities. In some problems, the criterion is to minimize *average* tardiness of activities so that most of the activities are not "too" late. In other problems, the criterion is to minimize the *maximal* tardiness so that even the most delayed activity is not too greatly overdue.
- The *peak capacity* of a resource is the greatest amount of the resource that is used over time.
- The *transition cost* of an activity is the cost incurred to switch from processing one activity to processing the next.

Tools for Searching

The search for a solution to any type of problem is aided by search goals. In practice, a good scheduling search goal depends on the structural properties of the problem and what the aim is: to find a solution, to improve a known solution, or to prove that some conditions are optimal. Scheduler provides several facilities, including predefined search goals, to facilitate the search for solutions. Examples presented here are solved using high-level search techniques of IBM© ILOG© Solver.

Search Goals

Scheduler search goals are designed to be easy to use and suited to the more usual cases. However, they have limited scope and may not be the best solution for certain types of problems. While Scheduler search goals can help in prototyping a project, they do not prevent you from developing the best heuristic algorithms for your problem within the Solver environment.

A goal is used to define the search for a solution to a model. The model for which an instance of a goal will search for a solution is specified via the `IloSolver` API. That is, once an `IloSolver` has extracted a model, a goal is associated with the model with the function `IloSolver::solve(const IloGoal)` or (`IloSolver::startNewSearch(const IloGoal)`). For more details see the *IBM ILOG Solver Reference Manual*.

Scheduler provides four predefined functions that return a goal to assign start times to activities in a schedule. For best results with these goals, activities should have bounded durations and resource demands. If precedence constraints between activities have negative delays, solutions may be missed.

- `IloGoal IloSetTimesForward(const IloEnv env);`
  Returns a goal that assigns a start time to each activity in the model. This goal can be used with an activity selector.
- ```
  IloGoal IloSetTimesForward(const IloEnv env,
                             const IloNumVar criterion);
  ```
  Returns a goal that assigns a start time to each activity in the model, while minimizing `criterion`. This goal can be used with an activity selector.
- `IloGoal IloSetTimesBackward(const IloEnv env);`
  Returns a goal that assigns an end time to each activity in the model. This goal can be used with an activity selector.
- ```
  IloGoal IloSetTimesBackward(const IloEnv env,
                              const IloNumVar criterion);
  ```
  Returns a goal that assigns an end time to each activity in the model, while maximizing `criterion`. This goal can be used with a activity selector.

Scheduler provides several goals for selecting resources from alternative resource sets.

- `IloGoal IloAssignAlternative(const IloEnv env,`
  `IloResourceSelector possibleSel = IloSelAltRes);`
  Returns a goal that assigns a possible resource as the selected one for an alternative resource constraint. All the alternative resource constraints for which more than one resource is still possible are considered.

- `IloGoal IloAssignAlternative(const IloEnv env,`
  `const IloAltResSet resources,`
  `IloResourceSelector possibleSel = IloSelAltRes);`
  Returns a goal that assigns a possible resource as the selected one for an alternative resource constraint. All the constraints on the resources in the alternative resource set `resources` are considered.

- `IloGoal IloAssignAlternative(const IloEnv env,`
  `const IloResource resource);`
  Returns a goal that assigns a `resource` as the selected one for an alternative resource constraint. All the alternative resource constraints for which `resource` is a possible alternative are considered in arbitrary order. To customize the order in which the alternative resources constraints are tried, it is necessary to use the Scheduler Engine classes (that is, create an `IlcAltRCSelectorObject` to parameterize an `IlcAssignAlternative` goal.

Ranking Goals

A set of instances of `IloResourceConstraint` may be ranked (ordered along the time line) for a resource which defines a mutually exclusive relationship between activities that require it. Ranking is defined for the classes `IloUnaryResource` and `IloStateResource`.

Ranking is possible for any resource constraint with an `IloFromStartToEnd` time extent.

When a constraint is ranked first, the activity corresponding to it is positioned at the head of the activities not already ranked. Therefore, at any moment when ranking the resource constraints of a resource, there are:

- resource constraints already chronologically ranked;
- resource constraints which can be ranked first, that is, which can be positioned at the head of the not ranked constraints;
- resource constraints which cannot be ranked first, that is, another not ranked resource constraint will necessarily be positioned at the head of the not ranked constraints.

See Figure 2.1.

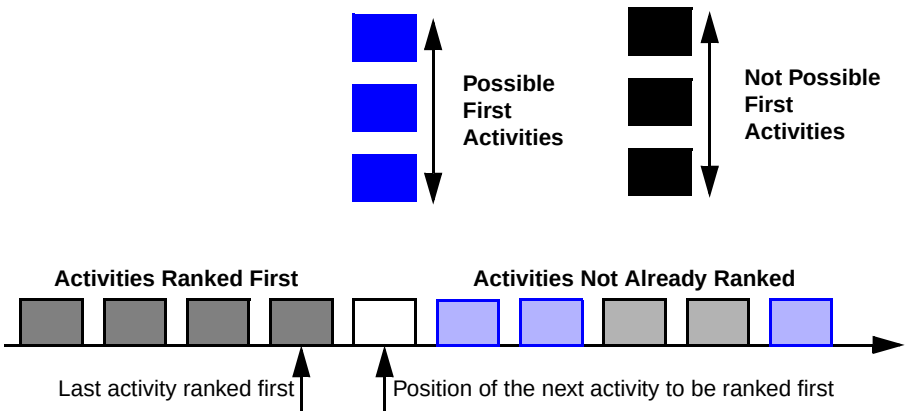


Figure 2.1 *Ranking Process*

This same breakdown applies in reverse chronological order for ranking resource constraints last.

> ***Note:*** *Ranking the activities on a resource does not necessarily bind their start and end times.*

Scheduler provides predefined functions that return a goal that ranks resource constraints.

- `IloGoal IloRankForward(const IloEnv env,`
  `const IloUnaryResource resource);`
  Returns a goal that ranks all resource constraints of the unary resource `resource`. The duration and capacity required by the activities is assumed to be bounded. By default, the resource constraint selector `IloSelFirstRCMinStartMax` selects the next resource constraint to be ranked first.

- `IloGoal IloRankForward(const IloEnv env,`
  `const IloStateResource resource);`
  Returns a goal that ranks all resource constraints of the state resource `resource`. The duration required by the activities is assumed to be bounded. By default, the resource constraint selector `IloSelFirstRCMinStartMax` selects the next resource constraint to be ranked first.

- `IloGoal IloRankForward(const IloEnv env);`
  Returns a goal that ranks all resource constraints of all unary resources in the model whose capacity constraints are not ignored (see `IloResourceParam` in the *IBM ILOG Scheduler Reference Manual*). By default the next resource to be ranked is chosen by the selector `IloSelResMinGlobalSlack` and the next resource constraint on the resource is selected by the selector `IloSelFirstRCMinStartMax`.

- `IloGoal IloRankForward(const IloEnv env,`
  `const IloNumVar criterion);`
  Returns a goal that ranks all resource constraints of all unary or state resources in the model whose capacity constraints are not ignored (see `IloResourceParam` in the *IBM ILOG Scheduler Reference Manual*), while minimizing `criterion`. By default, the next resource to be ranked is chosen by the selector `IloSelResMinGlobalSlack` and the next resource constraint on the resource is selected by the selector `IloSelFirstRCMinStartMax`.

- `IloGoal IloRankBackward(const IloEnv env,`
  `const IloUnaryResource resource);`
  Returns a goal that ranks all resource constraints of the unary resource `resource`. The duration and capacity required by the activities are assumed to be bounded. By default, the resource constraint selector `IloSelLastRCMaxEndMin` selects the next resource constraint to be ranked last.

- `IloGoal IloRankBackward(const IloEnv env,`
  `const IloStateResource resource);`
  Returns a goal that ranks all resource constraints of the state resource `resource`. The duration required by the activities are assumed to be bounded. By default, the resource constraint selector `IloSelLastRCMaxEndMin` selects the next resource constraint to be ranked last.

- `IloGoal IloRankBackward(const IloEnv env);`
  Returns a goal that ranks all resource constraints of all unary resources in the model whose capacity constraints are not ignored (see `IloResourceParam` in the *IBM ILOG Scheduler Reference Manual*). By default the next resource to be ranked backward is chosen by the selector `IloSelResMinGlobalSlack` and the next resource constraint on the resource is selected by the selector `IloSelLastRCMaxEndMin`.

- `IloGoal IloRankBackward(const IloEnv env,`
  `const IloNumVar criterion);`
  Returns a goal that ranks all resource constraints of all unary or state resources in the model whose capacity constraints are not ignored (see `IloResourceParam` in the *IBM ILOG Scheduler Reference Manual*), while maximizing `criterion`. By default the next resource to be ranked backward is chosen by the `IloSelResMinGlobalSlack` selector and the next resource constraint on the resource is selected by the `IloSelLastRCMaxEndMin` selector.

A Systematic Method for Solving Problems

When we look for a systematic method to solve problems in scheduling and resource allocation, we find that one of the most important principles of constraint programing consists of clearly separating the *definition* of a problem from its *resolution*. As a first step in

that direction, we recommend that you first write in natural language a precise description of the problem to be solved.

This description may suggest several possible representations (that is, several models) of the same problem based on the classes of constraints available in Scheduler (and as appropriate, Concert Technology and Solver). You should then evaluate those alternative models with respect to a number of criteria, including:

- The number of resources, activities, constrained variables, and constraints necessary to represent the problem according to each alternative (the *Solver User's Manual* explains how these numbers impact efficiency).
- The size of the search space and how this size could be reduced to guarantee that the scheduling algorithm will not consider thousands of possibilities very similar to each other. (By "very similar," we mean for example, symmetric schedules, or the same schedules except for one activity starting one or two microseconds sooner or later).
- The problem decompositions and heuristics that each alternative problem definition suggests.
- Application-dependent criteria, such as the importance of interaction with the user of the final application.

Once you have chosen a model, implement it using classes and functions from the Scheduler library (and as appropriate, from the Concert Technology and Solver libraries). In the following examples, we implement models by defining a function named `DefineModel` that takes problem data as its arguments and returns an instance of the class `IloModel`. Activities and resources can be accessed from that model, so returning the model is generally sufficient to represent a problem.

There are situations, however, where returning a model (an instance of `IloModel`) is not sufficient, for example, when `DefineModel` also defines optimization criteria in the form of additional variables to be used in the problem-solving part of the program. When that return value is not fully sufficient, you can specify additional return values as parameters passed by reference. The following example illustrates this implementation technique, the one most frequently employed in this manual.

```
IloModel DefineModel(/* parameters */, IloNumVar& criterion) {
  /* ... */
  IloModel model = /* ... */;
  /* ... */
  criterion = /* ... */;
  /* ... */
  return model;
}
```

The next step consists of implementing and experimenting with one or more strategies for exploring the search space. Except for very simple problems, for which straightforward algorithms exist, the solution of a scheduling problem requires some form of exploration of the solution space. In some cases, we define a global non-deterministic goal named `SolveProblem`; to define that goal, we use non-deterministic programing functions.

We can summarize the process of solving a scheduling or allocation problem in the following steps.

1. Describe the problem in natural language.
2. Use the description to design alternative models of the problem.
3. Study the models to determine their relative advantages (decomposition, heuristics, size of solution space, end-user interaction, etc.).
4. Use predefined classes of Scheduler (and as appropriate, Concert Technology and Solver) to implement the most advantageous model of the problem as your own `DefineModel`.
5. If necessary, refine return values of `DefineModel` through parameters passed by reference.
6. Explore the solution space by specifying your own search.
7. Refine the display of solutions to meet the needs of your end-users.

C H A P T E R

3

Using the Building Blocks

The first example in this chapter shows how the concepts of *activity* and of *temporal constraint* are used in Scheduler. It is organized, as are all examples in Part I, , around the problem of building a house—its foundation, masonry, roofing, painting, and so forth—with the obvious goal of scheduling the tasks to finish the entire house as quickly as possible. Some activities must necessarily take place before others, and that fact is expressed through *precedence constraints*.

The second example in the chapter shows how the concepts of *unary resource* and *resource constraint* are used in Scheduler. The example includes the same activities as the first, but adds a single worker who is required by all the activities.

Following our own guidelines for solving scheduling problems, we begin with a verbal description of the problem (see A Systematic Method for Solving Problems). Then we consider the model to represent it before we implement a solution.

Satisfying Temporal Constraints

In our first example, resources are not taken into account. The problem simply consists of assigning start and end times to activities, so as to satisfy temporal constraints and minimize a criterion. The criterion to minimize is typically the *makespan* of the overall schedule, that is, the amount of time between the start of the first activity and the end of the last activity.

Describe the Problem—Example 1

The following example illustrates the use of Scheduler on the problem of planning the construction of a house. For each activity in the project, the following table and Figure 3.1 show the duration of the activity in days along with the activities that must precede it.

Table 3.1 *House Construction Activities*

Activity	Duration	Preceding Activities
masonry	7	
carpentry	3	masonry
plumbing	8	masonry
ceiling	3	masonry
roofing	1	carpentry
painting	2	ceiling
windows	1	roofing
facade	2	roofing, plumbing
garden	1	roofing, plumbing
moving	1	windows, facade, garden, painting

Solving the problem consists of determining the least possible makespan for the project, that is, the least possible amount of time that elapses between the beginning of the first activity (`masonry`) and the end of the last activity (`moving`).

> ***Note:*** *In Scheduler, the unit of time represented by a variable is not defined. As a result, the duration of the masonry activity in this problem could be 7 hours or 7 weeks or 7 months. However, all time variables in a problem must represent the same unit of time.*

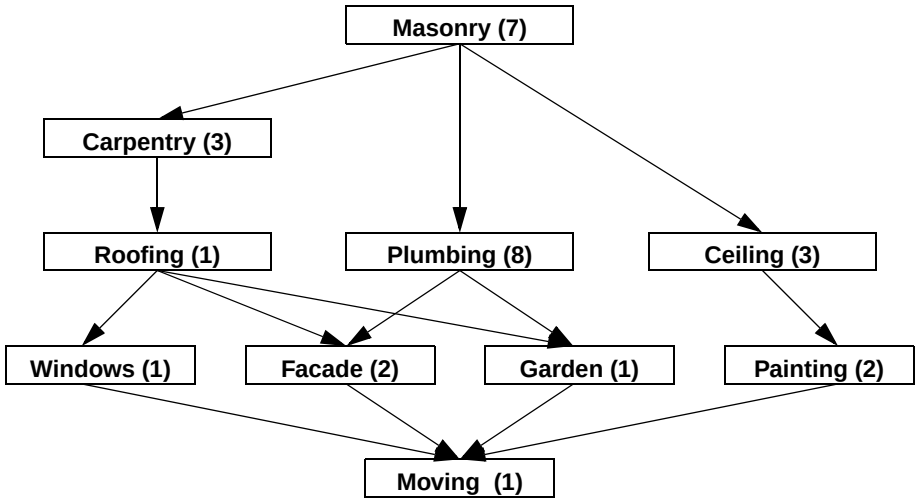


Figure 3.1 *House Construction Activities*

Create the Model

In Scheduler, activities and resources are created in an environment, an instance of the Concert Technology class `IloEnv`. Within that environment, a model of the scheduling problem is created, and this model will contain all the constraints necessary to define the problem. The following code creates the initial `model` in the environment `env`.

```
IloModel DefineModel(const IloEnv env, IloNumVar& makespan)
{
  IloModel model(env);
```

Define Activities

Activities are defined by the class `IloActivity`. The following constructor creates a new instance of `IloActivity` and adds it to the set of activities managed in the environment `env`. The activity has a processing time of `processingTime`, and the activity name is set to `name`. Other constructors of activities are available and will be discussed later.

```
IloActivity(IloEnv& env,
            IloNum processingTime,
            const char* name = 0);
```

The following code creates the activities for our first example: `masonry`, `carpentry`, `plumbing`, `ceiling`, `roofing`, `painting`, `windows`, `facade`, `garden`, and `moving`, in the context of the environment `env`.

```
/* CREATE THE ACTIVITIES. */
IloActivity masonry(env,   7, “masonry   “);
IloActivity carpentry(env, 3, “carpentry “);
IloActivity plumbing(env,  8, “plumbing  “);
IloActivity ceiling(env,   3, “ceiling   “);
IloActivity roofing(env,   1, “roofing   “);
IloActivity painting(env,  2, “painting  “);
IloActivity windows(env,   1, “windows   “);
IloActivity facade(env,    2, “facade    “);
IloActivity garden(env,    1, “garden    “);
IloActivity moving(env,    1, “moving    “);
```

Add Temporal Constraints

In this example, the only temporal constraints required are precedence constraints. Certain activities can only start after other activities have been completed. To add these constraints, we use the predefined member function `startsAfterEnd` of the class `IloActivity`.

```
/* ADD THE TEMPORAL CONSTRAINTS. */
model.add(carpentry.startsAfterEnd(masonry));
model.add(plumbing.startsAfterEnd(masonry));
model.add(ceiling.startsAfterEnd(masonry));
model.add(roofing.startsAfterEnd(carpentry));
model.add(painting.startsAfterEnd(ceiling));
model.add(windows.startsAfterEnd(roofing));
model.add(facade.startsAfterEnd(roofing));
model.add(facade.startsAfterEnd(plumbing));
model.add(garden.startsAfterEnd(roofing));
model.add(garden.startsAfterEnd(plumbing));
model.add(moving.startsAfterEnd(windows));
model.add(moving.startsAfterEnd(facade));
model.add(moving.startsAfterEnd(garden));
model.add(moving.startsAfterEnd(painting));
```

In the code above, `model` is the instance of `IloModel` that contains the functions used to declare the model. The method `add` specifies that the given parameters, expressions, or constraints must be verified by a solution.

Define the Objective

The makespan is the time that elapses between the beginning of the first activity (`masonry`) and the end of the last activity (`moving`). The objective of most of these examples is to

minimize the makespan. The following code sets `makespan` equal to the end time of the `moving` activity.

```
  /* SET THE MAKESPAN VARIABLE. */
  makespan = IloNumVar(env, 0, IloInfinity, ILOINT);
  model.add(moving.endsAt(makespan));

  return model;
}
```

Solve the Problem

To solve the problem, the model is extracted by an algorithm (`IloSolver`). That instance of `IloSolver` extracts the appropriate objects from the model and creates the corresponding objects needed by the algorithm to do the search.

The Solver function `IloInstantiate` creates the `goal` to instantiate the `makespan`. The solution search is then started using that `goal`. Constraint propagation is sufficient in this example to determine the earliest and latest start and end times of the activities.

```
    // Model
    IloEnv env;
    IloNumVar makespan;
    IloModel model = DefineModel(env, makespan);

    // Algorithm
    IloSolver solver(model);
    IloGoal goal = IloInstantiate(env, makespan);
```

Print the Solution

We use an instance of `IlcScheduler` to define a function to print the results. `IlcScheduler` is the repository of all the information used during the solve process. It allows retrieval of the search object (`Ilc`) that has been extracted from a scheduler extractable (`Ilo`). We use it here to retrieve the activities for printing, with an iterator (an instance of the class `IloIterator`) stepping through the activities.

```
void PrintSolution(const IloSolver solver)
{
  IlcScheduler scheduler(solver);
  IloEnv env = solver.getEnv();
  for(IloIterator<IloActivity> act(env);
      act.ok();
      ++act)
    env.out() << scheduler.getActivity(*act) << endl;
  solver.printInformation();
}
```

Complete Program and Output—Example 1

You can see the entire program `gsBasic.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

#if defined(ILO_SDXLOUTPUT)
#include “sdxloutput.h”
#endif

////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////
IloModel DefineModel(const IloEnv env, IloNumVar& makespan)
{
  IloModel model(env);
  /* CREATE THE ACTIVITIES. */
  IloActivity masonry(env,   7, “masonry   “);
  IloActivity carpentry(env, 3, “carpentry “);
  IloActivity plumbing(env,  8, “plumbing  “);
  IloActivity ceiling(env,   3, “ceiling   “);
  IloActivity roofing(env,   1, “roofing   “);
  IloActivity painting(env,  2, “painting  “);
  IloActivity windows(env,   1, “windows   “);
  IloActivity facade(env,    2, “facade    “);
  IloActivity garden(env,    1, “garden    “);
  IloActivity moving(env,    1, “moving    “);
  /* ADD THE TEMPORAL CONSTRAINTS. */
  model.add(carpentry.startsAfterEnd(masonry));
  model.add(plumbing.startsAfterEnd(masonry));
  model.add(ceiling.startsAfterEnd(masonry));
  model.add(roofing.startsAfterEnd(carpentry));
  model.add(painting.startsAfterEnd(ceiling));
  model.add(windows.startsAfterEnd(roofing));
  model.add(facade.startsAfterEnd(roofing));
  model.add(facade.startsAfterEnd(plumbing));
  model.add(garden.startsAfterEnd(roofing));
  model.add(garden.startsAfterEnd(plumbing));
  model.add(moving.startsAfterEnd(windows));
  model.add(moving.startsAfterEnd(facade));
  model.add(moving.startsAfterEnd(garden));
  model.add(moving.startsAfterEnd(painting));
  /* SET THE MAKESPAN VARIABLE. */
  makespan = IloNumVar(env, 0, IloInfinity, ILOINT);
  model.add(moving.endsAt(makespan));

  return model;
}

////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
```

```
//
////////////////////////////////////////////////////////////////////////////
void PrintSolution(const IloSolver solver)
{
  IlcScheduler scheduler(solver);
  IloEnv env = solver.getEnv();
  for(IloIterator<IloActivity> act(env);
      act.ok();
      ++act)
    env.out() << scheduler.getActivity(*act) << endl;
  solver.printInformation();
}

////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////

int main()
{
  try {
    // Model
    IloEnv env;
    IloNumVar makespan;
    IloModel model = DefineModel(env, makespan);

    // Algorithm
    IloSolver solver(model);
    IloGoal goal = IloInstantiate(env, makespan);
    if (solver.solve(goal)) {
      PrintSolution(solver);
#if defined(ILO_SDXLOUTPUT)
      IloSDXLOutput output(env);
      ofstream outFile("gsBasic.xml");
      output.write(IlcScheduler(solver), outFile, solver.getIntVar(makespan));
      outFile.close();
#endif
    }
    else
      solver.out() << "No Solution" << endl;

    env.end();

  } catch (IloException& exc) {
    cout << exc << endl;
  }

  return 0;
}

////////////////////////////////////////////////////////////////////////////
//
// RESULTS
//
////////////////////////////////////////////////////////////////////////////

/*
```

```
   masonry    [0 -- 7 --> 7]
   carpentry  [7..11 -- 3 --> 10..14]
   plumbing   [7 -- 8 --> 15]
   ceiling    [7..12 -- 3 --> 10..15]
   roofing    [10..14 -- 1 --> 11..15]
   painting   [10..15 -- 2 --> 12..17]
   windows    [11..16 -- 1 --> 12..17]
   facade     [15 -- 2 --> 17]
   garden     [15..16 -- 1 --> 16..17]
   moving     [17 -- 1 --> 18]
*/
```

In the standard output from a solution, the printed representation of each activity (instance of the class `IloActivity`) is its name (or the string "`IloActivity`" if the activity is not named) followed by information about its place in the schedule. This information is enclosed in square brackets. It consists of three items: start time, duration, and end time of the activity. Each item can be represented either as a single value or as an interval. If the item is represented as an interval, it appears as two integers separated by two dots.

For example:

- `masonry [0 -- 7 --> 7]` means the activity named `masonry` starts at 0, lasts 7 units of time, and ends at 7.
- `carpentry [7..11 -- 3 --> 10..14]` means that the activity named `carpentry` starts sometime between 7 and 11, lasts 3 units of time, and ends sometime between 10 and 14.

The start and end times of six activities (`garden`, `windows`, `painting`, `roofing`, `ceiling`, and `carpentry`) are not definitely set. In fact, any of these activities can execute from its latest start time to its latest end time (for example, from day 11 to day 14 in the case of `carpentry`) without compromising the makespan of the overall project.

Figure 3.2 provides a graphic display of the solution to the problem.

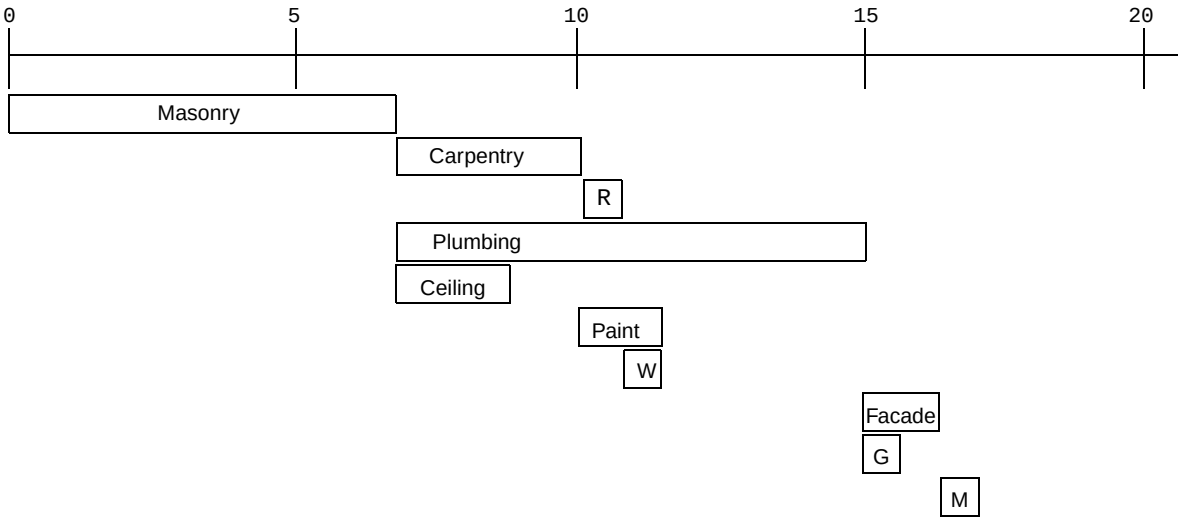


Figure 3.2 *Another View of the Solution*

Adding Resources and Resource Constraints

This second house building example shows how to express that activities require a resource in order to be processed. It has the same goal as the previous example: to schedule the tasks to finish the entire house as quickly as possible. More formally, it consists of the same ten activities, some of which must necessarily take place before others (expressed, as before, through precedence constraints), and one worker required by all activities (expressed through resource constraints).

As in the first problem, solving the problem consists of determining the earliest possible end time for the project. That is, assuming a fixed start time, the goal is to find the least possible amount of time that can elapse between the start of the first activity and the end of the last activity. However, now both precedence constraints and resource constraints must be considered. In addition, a search method must be developed, because simply propagating the constraints is not sufficient to find an optimal solution.

Describe the Problem—Example 2

The activities to perform, their durations, and the precedence constraints between those activities are identical to those presented earlier, as shown in Table 3.1 on page 52.

In addition, we have one worker who is required to perform all the activities. Because this worker can only perform one task at a time, we cannot schedule two activities at the same time, as we could in Example 1.

Define the Problem, Design a Model

In the context of this example, it is clear that the worker is a *unary resource* who can perform only one activity at a time and thus has a capacity of 1. Each activity requires the worker.

Create the Resource

The following code creates the worker as an object of the `IloUnaryResource` class.

```
/* CREATE THE RESOURCE. */
IloUnaryResource worker(env);
```

Add the Resource Constraints

To add resource constraints, we use the member function `IloActivity::requires` which states that the invoking activity requires the capacity of the given resource (`worker`) from the beginning to the end of the activity's processing time.

The general form for specifying that an activity `A` requires a resource `R` is `model.add(A.requires(R, capacity));` where `model` is an instance of the class `IloModel`, and `capacity` is either a constant or a constrained variable specifying the amount that the activity requires from the resource. Because the `worker` resource is a unary resource with a capacity of 1 and because the default capacity of the method `requires` is also 1, the capacity required by the activity does not have to be stated.

```
/* ADD THE RESOURCE CONSTRAINTS. */
model.add(masonry.requires(worker));
model.add(carpentry.requires(worker));
model.add(plumbing.requires(worker));
model.add(ceiling.requires(worker));
model.add(roofing.requires(worker));
model.add(painting.requires(worker));
model.add(windows.requires(worker));
model.add(facade.requires(worker));
model.add(garden.requires(worker));
model.add(moving.requires(worker));
```

Define the Objective

Our objective is still to determine the least possible makespan for the project. The Concert Technology function `IloMinimize` is used to set the objective of minimizing the `makespan` in the model.

```
/* SET THE MAKESPAN VARIABLE. */
makespan = IloNumVar(env, 0, IloInfinity, ILOINT);
model.add(moving.endsAt(makespan));

/* SET THE OBJECTIVE */
model.add(IloMinimize(env, makespan));
```

Solve the Problem

The previous problem was constrained only by regular temporal constraints and thus could be solved directly by propagation. In this slightly more complicated version, constraint propagation alone is not sufficient to solve the problem because of the constraints on the unary resource. `IloRankForward` returns a goal used to rank all the resource constraints on the resource `worker`, while minimizing the criterion `makespan`. (For more information on ranking, see Ranking Goals or the *ILOG Scheduler Reference Manual*.)

```
    IloGoal goal = IloRankForward(env, makespan);
    IloSolver solver(model);
    if (solver.solve(goal)) {
      PrintSolution(solver,makespan);

#if defined(ILO_SDXLOUTPUT)
      IloSDXLOutput output(env);
      ofstream outFile(“gsUnary.xml”);
      output.write(IlcScheduler(solver), outFile, solver.getIntVar(makespan));
      outFile.close();
#endif
    }
    else
      solver.out() << “No Solution” << endl;
```

Complete Program and Output—Example 2

You can see the entire program `gsUnary.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

#if defined(ILO_SDXLOUTPUT)
#include “sdxloutput.h”
#endif

////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////

IloModel DefineModel(IloEnv env, IloNumVar& makespan)
{
  IloModel model(env);

  /* CREATE THE ACTIVITIES. */
  IloActivity masonry(env,   7, “masonry   “);
  IloActivity carpentry(env, 3, “carpentry “);
  IloActivity plumbing(env,  8, “plumbing  “);
  IloActivity ceiling(env,   3, “ceiling   “);
  IloActivity roofing(env,   1, “roofing   “);
```

```
  IloActivity painting(env,  2, "painting  ");
  IloActivity windows(env,   1, "windows   ");
  IloActivity facade(env,    2, "facade    ");
  IloActivity garden(env,    1, "garden    ");
  IloActivity moving(env,    1, "moving    ");

  /* ADD THE TEMPORAL CONSTRAINTS. */
  model.add(carpentry.startsAfterEnd(masonry));
  model.add(plumbing.startsAfterEnd(masonry));
  model.add(ceiling.startsAfterEnd(masonry));
  model.add(roofing.startsAfterEnd(carpentry));
  model.add(painting.startsAfterEnd(ceiling));
  model.add(windows.startsAfterEnd(roofing));
  model.add(facade.startsAfterEnd(roofing));
  model.add(facade.startsAfterEnd(plumbing));
  model.add(garden.startsAfterEnd(roofing));
  model.add(garden.startsAfterEnd(plumbing));
  model.add(moving.startsAfterEnd(windows));
  model.add(moving.startsAfterEnd(facade));
  model.add(moving.startsAfterEnd(garden));
  model.add(moving.startsAfterEnd(painting));
  /* CREATE THE RESOURCE. */
  IloUnaryResource worker(env);
  /* ADD THE RESOURCE CONSTRAINTS. */
  model.add(masonry.requires(worker));
  model.add(carpentry.requires(worker));
  model.add(plumbing.requires(worker));
  model.add(ceiling.requires(worker));
  model.add(roofing.requires(worker));
  model.add(painting.requires(worker));
  model.add(windows.requires(worker));
  model.add(facade.requires(worker));
  model.add(garden.requires(worker));
  model.add(moving.requires(worker));
  /* SET THE MAKESPAN VARIABLE. */
  makespan = IloNumVar(env, 0, IloInfinity, ILOINT);
  model.add(moving.endsAt(makespan));

  /* SET THE OBJECTIVE */
  model.add(IloMinimize(env, makespan));
  return model;
}


////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////////////////

void PrintSolution(const IloSolver solver, const IloNumVar makespan)
{
  IlcScheduler scheduler(solver);
  IloEnv env = solver.getEnv();
  env.out() << "Solution with makespan "
            << solver.getIntVar(makespan).getMin() << endl;
  for(IloIterator<IloActivity> act(env);
      act.ok();
```

```
      ++act)
    env.out() << scheduler.getActivity(*act) << endl;
  solver.printInformation();
}

////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////

int main()
{
  try {
    IloEnv env;
    IloNumVar makespan;
    IloModel model = DefineModel(env, makespan);

    IloGoal goal = IloRankForward(env, makespan);
    IloSolver solver(model);
    if (solver.solve(goal)) {
      PrintSolution(solver,makespan);

#if defined(ILO_SDXLOUTPUT)
      IloSDXLOutput output(env);
      ofstream outFile("gsUnary.xml");
      output.write(IlcScheduler(solver), outFile, solver.getIntVar(makespan));
      outFile.close();
#endif
    }
    else
      solver.out() << "No Solution" << endl;
    env.end();

  } catch (IloException& exc) {
    cout << exc << endl;
  }
  return 0;
}

////////////////////////////////////////////////////////////////////////////
//
// RESULTS
//
////////////////////////////////////////////////////////////////////////////

/*
  Solution with makespan 29
  masonry    [0 -- 7 --> 7]
  carpentry  [15 -- 3 --> 18]
  plumbing   [7 -- 8 --> 15]
  ceiling    [18 -- 3 --> 21]
  roofing    [21 -- 1 --> 22]
  painting   [24 -- 2 --> 26]
  windows    [27 -- 1 --> 28]
  facade     [22 -- 2 --> 24]
  garden     [26 -- 1 --> 27]
  moving     [28 -- 1 --> 29]
```

```
*/
```

The start and end times of all activities are fixed in this example.

Figure 3.3 provides a graphic display of the solution to our problem.

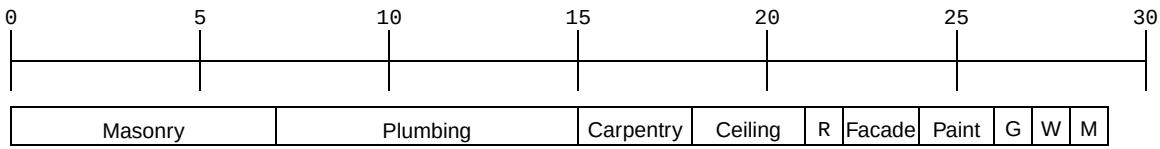


Figure 3.3 *Solution with Resource Constraints*

C H A P T E R

4

Using Discrete Resources

A *discrete resource* represents a resource of finite capacity, which is defined to be a positive integer number. The capacity of a discrete resource can vary with time. At any time *t*, the capacity represents the number of copies or instances of the resource that are available. Each activity in a schedule may require some amount of the capacity of a discrete resource.

The two examples in this chapter illustrate two ways in which discrete resources can be used. In the first example we use a discrete resource to model two identical workers, one of which is required by each of the activities in the problem. The second example shows how a discrete resource can be used to model a consumable resource.

Working with Discrete Resources

Once again, our example is based on the house building problem, with the goal of scheduling the tasks to finish the entire house as quickly as possible. It still consists of ten activities, but there is now an additional worker, identical in capability to the first worker.

As before, solving the problem consists of determining the earliest possible end time for the project. However, instead of a unary resource, we have a discrete resource with a capacity of 2.

Describe the Problem—Example 3

The activities to perform, their durations, and the precedence constraints between those activities, are identical to those presented in Using the Building Blocks, as shown in Table 3.1.

In this example we have two workers with equal ability to perform the activities.

Define the Problem, Design a Model

Because the workers in this problem are considered to be identical, we can represent them as a discrete resource with a capacity of 2. Each activity requires one of the workers.

Create the Resource

The following code creates the workers as an instance of `IloDiscreteResource` with a capacity of 2.

```
/* CREATE THE RESOURCE. */
IloDiscreteResource workers(env, 2);
```

Add the Resource Constraints

The resource constraints are once again declared using the method `IloActivity::requires`, which states that the invoking activity requires the capacity of the given resource (`workers`) from the beginning to the end of its execution. Because the default capacity of the method `requires` is 1, the capacity required by the activity does not have to be stated.

```
/* ADD THE RESOURCE CONSTRAINTS. */
model.add(masonry.requires(workers));
model.add(carpentry.requires(workers));
model.add(plumbing.requires(workers));
model.add(ceiling.requires(workers));
model.add(roofing.requires(workers));
model.add(painting.requires(workers));
model.add(windows.requires(workers));
model.add(facade.requires(workers));
model.add(garden.requires(workers));
model.add(moving.requires(workers));
```

Define the Objective

Our objective is still to minimize the makespan for the project. `IloMinimize` is used just as for the previous example.

Solve the Problem

This solution uses the function `IloSetTimesForward`, which creates and returns a goal that assigns a start time to all activities in the model, while minimizing the criterion `makespan`. For more information on `IloSetTimesForward`, see Search Goals or the *IBM ILOG Scheduler Reference Manual*.

```
// Algorithm
IloSolver solver(model);
IloGoal goal = IloSetTimesForward(env, makespan);
```

Complete Program and Output—Example 3

You can see the entire program `gsDisc.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

#if defined(ILO_SDXLOUTPUT)
#include “sdxloutput.h”
#endif

////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////

IloModel DefineModel(const IloEnv env, IloNumVar& makespan)
{
  IloModel model(env);

  /* CREATE THE ACTIVITIES. */
  IloActivity masonry(env,   7, “masonry   “);
  IloActivity carpentry(env, 3, “carpentry “);
  IloActivity plumbing(env,  8, “plumbing  “);
  IloActivity ceiling(env,   3, “ceiling   “);
  IloActivity roofing(env,   1, “roofing   “);
  IloActivity painting(env,  2, “painting  “);
  IloActivity windows(env,   1, “windows   “);
  IloActivity facade(env,    2, “facade    “);
  IloActivity garden(env,    1, “garden    “);
  IloActivity moving(env,    1, “moving    “);

  /* ADD THE TEMPORAL CONSTRAINTS. */
  model.add(carpentry.startsAfterEnd(masonry));
  model.add(plumbing.startsAfterEnd(masonry));
  model.add(ceiling.startsAfterEnd(masonry));
  model.add(roofing.startsAfterEnd(carpentry));
  model.add(painting.startsAfterEnd(ceiling));
  model.add(windows.startsAfterEnd(roofing));
```

```
  model.add(facade.startsAfterEnd(roofing));
  model.add(facade.startsAfterEnd(plumbing));
  model.add(garden.startsAfterEnd(roofing));
  model.add(garden.startsAfterEnd(plumbing));
  model.add(moving.startsAfterEnd(windows));
  model.add(moving.startsAfterEnd(facade));
  model.add(moving.startsAfterEnd(garden));
  model.add(moving.startsAfterEnd(painting));
  /* CREATE THE RESOURCE. */
  IloDiscreteResource workers(env, 2);
  /* ADD THE RESOURCE CONSTRAINTS. */
  model.add(masonry.requires(workers));
  model.add(carpentry.requires(workers));
  model.add(plumbing.requires(workers));
  model.add(ceiling.requires(workers));
  model.add(roofing.requires(workers));
  model.add(painting.requires(workers));
  model.add(windows.requires(workers));
  model.add(facade.requires(workers));
  model.add(garden.requires(workers));
  model.add(moving.requires(workers));
  /* SET THE MAKESPAN VARIABLE. */
  makespan = IloNumVar(env, 0, IloInfinity, ILOINT);
  model.add(moving.endsAt(makespan));
  /* SET THE OBJECTIVE */
  model.add(IloMinimize(env, makespan));
  return model;
}

////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////////////////

void PrintSolution(const IloSolver solver, const IloNumVar makespan)
{
  IlcScheduler scheduler(solver);
  IloEnv env = solver.getEnv();
  env.out() << "Solution with makespan "
            << solver.getIntVar(makespan).getMin() << endl;
  for(IloIterator<IloActivity> act(env);
      act.ok();
      ++act)
    env.out() << scheduler.getActivity(*act) << endl;
  solver.printInformation();
}

////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////

int main()
{
  try {
    // Model
```

```
    IloEnv env;
    IloNumVar makespan;
    IloModel model = DefineModel(env, makespan);

    // Algorithm
    IloSolver solver(model);
    IloGoal goal = IloSetTimesForward(env, makespan);

    if (solver.solve(goal)) {
      PrintSolution(solver,makespan);

#if defined(ILO_SDXLOUTPUT)
      IloSDXLOutput output(env);
      ofstream outFile("gsDisc.xml");
      output.write(IlcScheduler(solver), outFile, solver.getIntVar(makespan));
      outFile.close();
#endif

    }
    else
      solver.out() << "No Solution" << endl;

    env.end();

  } catch (IloException& exc) {
    cout << exc << endl;
  }
  return 0;
}

////////////////////////////////////////////////////////////////////////////
//
// RESULTS
//
////////////////////////////////////////////////////////////////////////////

/*
  Solution with makespan 19
  masonry    [0 -- 7 --> 7]
  carpentry  [7 -- 3 --> 10]
  plumbing   [7 -- 8 --> 15]
  ceiling    [11 -- 3 --> 14]
  roofing    [10 -- 1 --> 11]
  painting   [16 -- 2 --> 18]
  windows    [14 -- 1 --> 15]
  facade     [15 -- 2 --> 17]
  garden     [15 -- 1 --> 16]
  moving     [18 -- 1 --> 19]
*/
```

The start and end times of all activities are fixed and the total time to complete the project is 19 days. Worker1 works from day 0 to day 17, Worker2 works from day 7 to day 19. Notice that the tasks could be switched between the workers at days 7 and 15.

Figure 4.1 provides a graphic display of the solution to our problem.

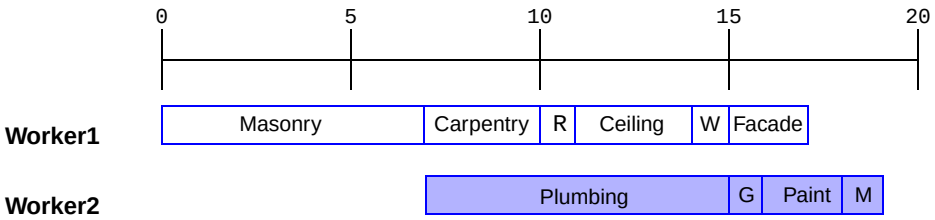


Figure 4.1 *Solution with Discrete Resource of Capacity 2*

Managing Consumable Resources

The aim of this second example is to introduce a consumable resource: the budget available for the execution of the project activities. In this example, a certain amount of money must be paid at the beginning of each activity. The budget required to build the entire house is not all available at the start of the project, but more money becomes available a given number of days afterwards. The goal is still to minimize the project end date, but both the precedence and budget constraints must now be taken into account, as shown in Figure 4.2.

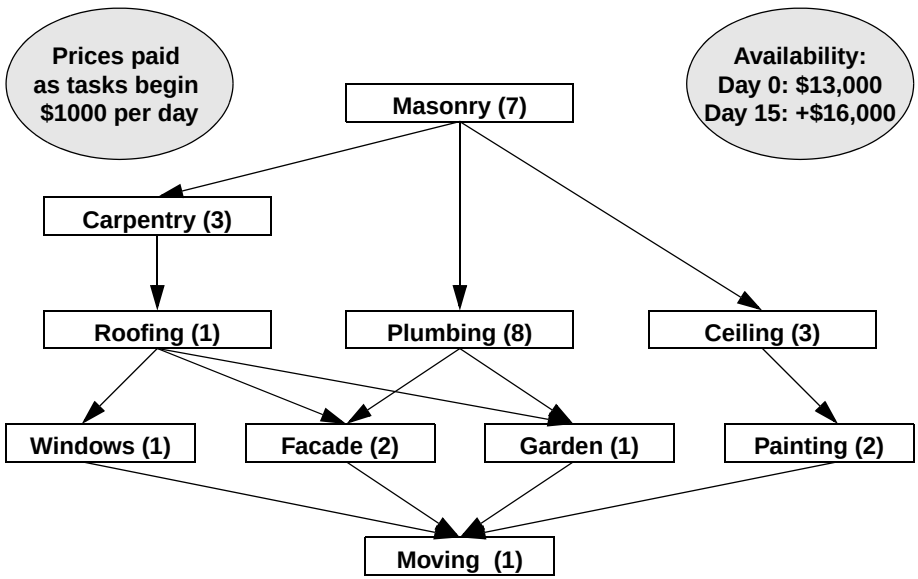


Figure 4.2 *Budget Constraints on the Problem*

In this example we also introduce the class `IloSchedulerSolution`. An instance of this class is used to store and share elements of the solution of a problem (for example, activity and resource content). This example shows how to create, manage, and use a solution object to print resource capacity and activity schedules.

Describe the Problem—Example 4

The activities to perform, their durations, and the precedence constraints between these activities are identical to those of Using the Building Blocks (shown in Table 3.1). As in Example 2, we have only one worker, an `IloUnaryResource`, available to perform the activities.

In addition, each activity requires the payment of an amount of money proportional to the duration of the activity. This amount, to be paid at the beginning of the activity, is set to $1,000 per day per activity, so that the total budget for the project is $29,000 (that is, $1,000 times the sum of the durations of the project activities). The sum is available in two portions: $13,000 is available at the start of the project, and $16,000 is available 15 days afterwards. As in Example 3, our goal is to minimize the project end date, but both the precedence and budget constraints must be taken into account.

Define the Problem, Design a Model

As indicated before, one of the most important methodological principles of constraint programming is to distinguish the definition and the resolution of the problem. We start from the problem definition used in Example 2. The activities, temporal constraints, worker resource, worker resource constraints, and objective are all created in the same way as in that example. So now we can tackle the representation of the budget and budget constraints.

In the context of this example, it is clear that money is a *discrete resource*, available in limited quantity, and consumed by the activities in the schedule—in other words, required to start each activity and not recovered when the activity ends.

The method `consumes(resource, capacity)` constrains an activity to consume `capacity` of `resource`. The consumed capacity, which can be a non-negative number or a Solver non-negative number variable, is non-recoverable. Zero capacity is allowed, in which case the resource constraint does not change the availability of the resource.

Create the Budget Resource

The following lines create an instance of `IloDiscreteResource` representing the budget. They set the overall capacity of this resource to $29,000. The member function `setCapacityMax` of the class `IloDiscreteResource` is used to specify that only $13,000 is available from day 0 up to day 15. As its arguments, this function takes three parameters: two points in time, `timeMin` and `timeMax` (defining an interval `[timeMin`

timeMax), here [0 15)), and a number specifying the maximal capacity available over this interval ($13,000).

```
  /* CREATE THE CONSUMPTION RESOURCE. */
  budget = IloDiscreteResource(env, 29000, "Budget");
  budget.setCapacityMax(0, 15, 13000);
```

Create the Budget Resource Constraints

To add the budget resource constraints to the model, we use the predefined member function consumes. The durations of the activities are defined in the array dur using IloNum.

```
IloNum dur [] = { 7, 3, 8, 3, 1, 2, 1, 2, 1, 1};

  /* ... */
  /* ADD THE CONSUMPTION RESOURCE CONSTRAINTS. */
  model.add(masonry.consumes(budget,   1000*dur[0]));
  model.add(carpentry.consumes(budget, 1000*dur[1]));
  model.add(plumbing.consumes(budget,  1000*dur[2]));
  model.add(ceiling.consumes(budget,   1000*dur[3]));
  model.add(roofing.consumes(budget,   1000*dur[4]));
  model.add(painting.consumes(budget,  1000*dur[5]));
  model.add(windows.consumes(budget,   1000*dur[6]));
  model.add(facade.consumes(budget,    1000*dur[7]));
  model.add(garden.consumes(budget,    1000*dur[8]));
  model.add(moving.consumes(budget,    1000*dur[9]));

  /* SET THE OBJECTIVE */
  model.add(IloMinimize(env, makespan));

  /* REGISTER VARIABLES TO BE STORED IN THE SOLUTION */
  solution.getSolution().add(makespan);
  solution.add(budget);

  solution.add(masonry);
  solution.add(carpentry);
  solution.add(plumbing);
  solution.add(ceiling);
  solution.add(roofing);
  solution.add(painting);
  solution.add(windows);
  solution.add(facade);
  solution.add(garden);
  solution.add(moving);

  return model;
}
```

Store the Solution

An instance of the class IloSchedulerSolution (named solution) is used to store activities, resources and other components of a solution. These components are retrieved

later in this example for printing. See the *IBM ILOG Scheduler Reference Manual* for more information on `IloSchedulerSolution`.

```
  /* REGISTER VARIABLES TO BE STORED IN THE SOLUTION */
  solution.add(makespan);
  solution.add(budget);

  solution.add(masonry);
  solution.add(carpentry);
  solution.add(plumbing);
  solution.add(ceiling);
  solution.add(roofing);
  solution.add(painting);
  solution.add(windows);
  solution.add(facade);
  solution.add(garden);
  solution.add(moving);

  return model;
}
```

Solve the Problem

This problem is solved with `IloSetTimesForward` while minimizing `makespan`, as for the previous example. The addition of `IloSchedulerSolution` to the model makes the solution data available to the program.

```
  /* SET THE MAKESPAN VARIABLE. */
  makespan = IloNumVar(env, 0, IloInfinity, ILOINT);
  model.add(moving.endsAt(makespan));

  /* ... */

    IloEnv env;
    IloNumVar makespan;
    IloDiscreteResource budget;
    IloSchedulerSolution solution(env);
    IloModel model = DefineModel(env, makespan, budget, solution);

    IloSolver solver(model);
    IloGoal goal = IloSetTimesForward(env, makespan);
```

Print the Solution

The function `PrintSolution` allows retrieval and printing of the budget resource capacity from `solution`, the instance of the class `IloSchedulerSolution`.

```
void PrintSolution(const IloSolver& solver,
                   const IloSchedulerSolution solution,
                   const IloNumVar makespan,
                   const IloDiscreteResource budget)
{
  solver.out() << “Solution with makespan “
               << solution.getMin(makespan) << endl << endl;
```

```
  for (IloSchedulerSolution::ActivityIterator iter(solution);
       iter.ok(); ++iter)
  {
    IloActivity act = *iter;

    solver.out() << act.getName();

    solver.out() << " [" << solution.getStartMin(act);
    if (solution.getStartMin(act) != solution.getStartMax(act))
      solver.out() << ".." << solution.getStartMax(act);
    solver.out() << " -- "
                 << solution.getProcessingTimeMin(act);

    solver.out() << " --> " << solution.getEndMin(act);
    if (solution.getEndMin(act) != solution.getEndMax(act))
      solver.out() << ".." << solution.getEndMax(act);
    solver.out() << "]" << endl;
  }
  solver.out() << endl;

  solver.out() << budget << endl;
  solver.out() << endl;

  for (IloNumToNumStepFunctionCursor iter1(solution.getLevelMin(budget));
       iter1.ok(); ++iter1) {
    solver.out() << "date "
                 << iter1.getSegmentMin()
                 << " amount "
                 << iter1.getValue() << endl;
  }
}
```

Complete Program and Output—Example 4

You can see the entire program `gsBudget.cpp` here or online in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

#if defined(ILO_SDXLOUTPUT)
#include "sdxloutput.h"
#endif

////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////
IloNum dur [] = { 7, 3, 8, 3, 1, 2, 1, 2, 1, 1};

IloModel DefineModel(const IloEnv env,
                     IloNumVar& makespan,
                     IloDiscreteResource& budget,
                     IloSchedulerSolution& solution)
{
```

```
IloModel model(env);

/* CREATE THE ACTIVITIES. */
IloActivity masonry(env,   dur[0], “masonry   “);
IloActivity carpentry(env, dur[1], “carpentry “);
IloActivity plumbing(env,  dur[2], “plumbing  “);
IloActivity ceiling(env,   dur[3], “ceiling   “);
IloActivity roofing(env,   dur[4], “roofing   “);
IloActivity painting(env,  dur[5], “painting  “);
IloActivity windows(env,   dur[6], “windows   “);
IloActivity facade(env,    dur[7], “facade    “);
IloActivity garden(env,    dur[8], “garden    “);
IloActivity moving(env,    dur[9], “moving    “);

/* ADD THE TEMPORAL CONSTRAINTS. */
model.add(carpentry.startsAfterEnd(masonry));
model.add(plumbing.startsAfterEnd(masonry));
model.add(ceiling.startsAfterEnd(masonry));
model.add(roofing.startsAfterEnd(carpentry));
model.add(painting.startsAfterEnd(ceiling));
model.add(windows.startsAfterEnd(roofing));
model.add(facade.startsAfterEnd(roofing));
model.add(facade.startsAfterEnd(plumbing));
model.add(garden.startsAfterEnd(roofing));
model.add(garden.startsAfterEnd(plumbing));
model.add(moving.startsAfterEnd(windows));
model.add(moving.startsAfterEnd(facade));
model.add(moving.startsAfterEnd(garden));
model.add(moving.startsAfterEnd(painting));

/* CREATE THE WORKER RESOURCE. */
IloUnaryResource worker(env);

/* ADD THE WORKER RESOURCE CONSTRAINTS. */
model.add(masonry.requires(worker));
model.add(carpentry.requires(worker));
model.add(plumbing.requires(worker));
model.add(ceiling.requires(worker));
model.add(roofing.requires(worker));
model.add(painting.requires(worker));
model.add(windows.requires(worker));
model.add(facade.requires(worker));
model.add(garden.requires(worker));
model.add(moving.requires(worker));
/* SET THE MAKESPAN VARIABLE. */
makespan = IloNumVar(env, 0, IloInfinity, ILOINT);
model.add(moving.endsAt(makespan));
/* CREATE THE CONSUMPTION RESOURCE. */
budget = IloDiscreteResource(env, 29000, “Budget”);
budget.setCapacityMax(0, 15, 13000);
/* ADD THE CONSUMPTION RESOURCE CONSTRAINTS. */
model.add(masonry.consumes(budget,   1000*dur[0]));
model.add(carpentry.consumes(budget, 1000*dur[1]));
model.add(plumbing.consumes(budget,  1000*dur[2]));
model.add(ceiling.consumes(budget,   1000*dur[3]));
model.add(roofing.consumes(budget,   1000*dur[4]));
model.add(painting.consumes(budget,  1000*dur[5]));
model.add(windows.consumes(budget,   1000*dur[6]));
```

```
  model.add(facade.consumes(budget,    1000*dur[7]));
  model.add(garden.consumes(budget,    1000*dur[8]));
  model.add(moving.consumes(budget,    1000*dur[9]));

  /* SET THE OBJECTIVE */
  model.add(IloMinimize(env, makespan));

  /* REGISTER VARIABLES TO BE STORED IN THE SOLUTION */
  solution.getSolution().add(makespan);
  solution.add(budget);

  solution.add(masonry);
  solution.add(carpentry);
  solution.add(plumbing);
  solution.add(ceiling);
  solution.add(roofing);
  solution.add(painting);
  solution.add(windows);
  solution.add(facade);
  solution.add(garden);
  solution.add(moving);

  return model;
}

////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////////////////

void PrintSolution(const IloSolver& solver,
                   const IloSchedulerSolution solution,
                   const IloNumVar makespan,
                   const IloDiscreteResource budget)
{
  solver.out() << "Solution with makespan "
               << solution.getSolution().getMin(makespan) << endl << endl;

  for (IloSchedulerSolution::ActivityIterator iter(solution);
       iter.ok(); ++iter)
  {
    IloActivity act = *iter;

    solver.out() << act.getName();

    solver.out() << " [" << solution.getStartMin(act);
    if (solution.getStartMin(act) != solution.getStartMax(act))
      solver.out() << ".." << solution.getStartMax(act);
    solver.out() << " -- "
                 << solution.getProcessingTimeMin(act);

    solver.out() << " --> " << solution.getEndMin(act);
    if (solution.getEndMin(act) != solution.getEndMax(act))
      solver.out() << ".." << solution.getEndMax(act);
    solver.out() << "]" << endl;
  }
  solver.out() << endl;
```

```
  solver.out() << budget << endl;
  solver.out() << endl;

  for (IloNumToNumStepFunctionCursor iter1(solution.getLevelMin(budget));
       iter1.ok(); ++iter1) {
    solver.out() << "date "
                 << iter1.getSegmentMin()
                 << " amount "
                 << iter1.getValue() << endl;
  }
}

////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////

int main()
{
  try {
    IloEnv env;
    IloNumVar makespan;
    IloDiscreteResource budget;
    IloSchedulerSolution solution(env);
    IloModel model = DefineModel(env, makespan, budget, solution);

    IloSolver solver(model);
    IloGoal goal = IloSetTimesForward(env, makespan);
    if (solver.solve(goal)) {
      solution.store( IlcScheduler(solver) );
      PrintSolution(solver,solution, makespan, budget);
#if defined(ILO_SDXLOUTPUT)
      IloSDXLOutput output(env);
      ofstream outFile("gsBudget.xml");
      output.write(IlcScheduler(solver), outFile, solver.getIntVar(makespan));
      outFile.close();
#endif

    }
    else
      solver.out() << "No Solution" << endl;

    solution.end();
    env.end();

  } catch (IloException& exc) {
    cout << exc << endl;
  }
  return 0;
}

////////////////////////////////////////////////////////////////////////////
//
// RESULTS
//
////////////////////////////////////////////////////////////////////////////
```

```
/*
  Solution with makespan 31

  masonry    [0 -- 7 --> 7]
  carpentry  [7 -- 3 --> 10]
  plumbing   [16 -- 8 --> 24]
  ceiling    [10 -- 3 --> 13]
  roofing    [15 -- 1 --> 16]
  painting   [28 -- 2 --> 30]
  windows    [27 -- 1 --> 28]
  facade     [25 -- 2 --> 27]
  garden     [24 -- 1 --> 25]
  moving     [30 -- 1 --> 31]

  Budget [29000]

  date 0 amount 7000
  date 7 amount 10000
  date 10 amount 13000
  date 15 amount 14000
  date 16 amount 22000
  date 24 amount 23000
  date 25 amount 25000
  date 27 amount 26000
  date 28 amount 28000
  date 30 amount 29000
*/
```

The start and end times of all activities are fixed and the total time to complete the project is 31 days. Notice that there is no activity on days 14 and 15, because the original budget release has been used up and the additional budget has not yet become available. Figure 4.3 provides a graphic display of the solution to our problem.

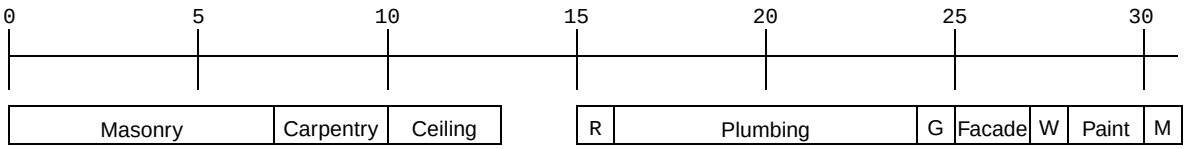


Figure 4.3 *Solution with Consumable Resource*

Figure 4.4 shows the amount of available and used budget capacity over the schedule.

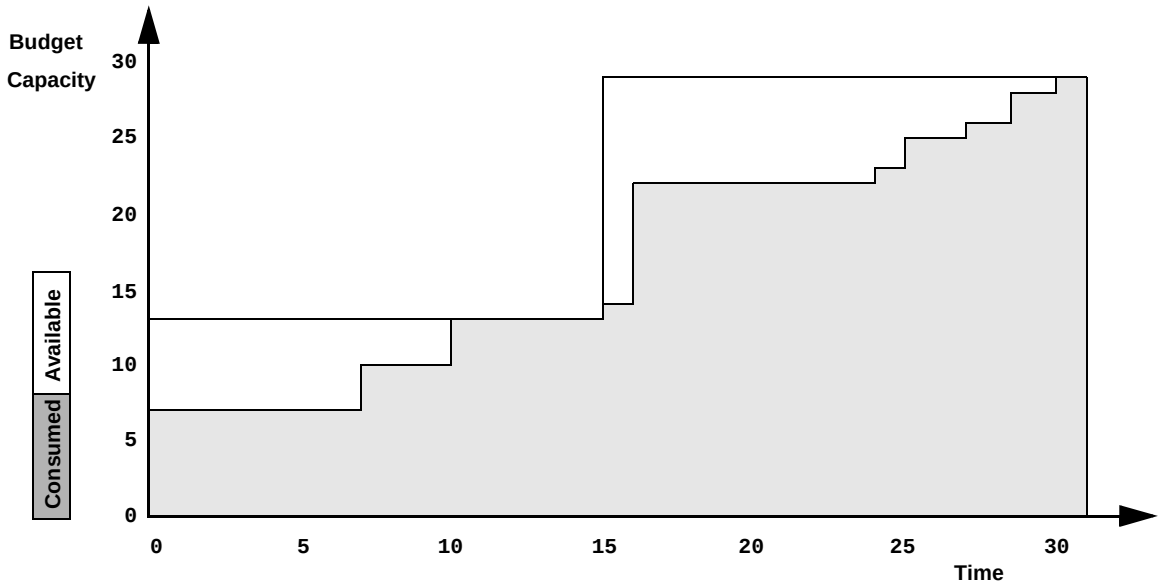


Figure 4.4 *Budget Capacity Availability Over Time*

C H A P T E R

5

Adding Transition Times

Given an activity A_1 that precedes an activity A_2, the *transition time* between A_1 and A_2 is an amount of time that must elapse between the end of A_1 and the beginning of A_2.

The example in this chapter shows how to model a problem that involves transition times between activities. The transition time represents a required delay between tasks; for example, the time required to clean up, to set up a piece of equipment, or time spent waiting for paint to dry or concrete to set.

Describe the Problem—Example 5

In this example, we have one worker who is required to build three houses. The tasks in each house have different durations corresponding to the house design (Duration Design 1, 2, or 3 in Table 5.1). There is also a transition time between tasks that is based on the activity transition type. In this example, every activity has its own transition type. The actual transition times are defined later.

Table 5.1 *Multiple House Construction Activities*

Activity	Transition Type	Duration Design1	Duration Design2	Duration Design3	Preceding Activities
masonry	1	7	12	15	
carpentry	2	3	5	3	masonry
plumbing	3	8	10	10	masonry
ceiling	4	3	5	6	masonry
roofing	5	1	2	2	carpentry
painting	6	2	5	3	ceiling
windows	7	1	2	2	roofing
facade	8	2	3	3	roofing, plumbing
garden	9	1	2	2	roofing, plumbing
moving	0	1	1	1	windows, facade, garden, painting

In addition, the second house cannot be started before day 5 and must be finished by day 80.

As in all our previous examples, our objective is to minimize the makespan of the problem.

Define the Problem, Design a Model

This problem has several elements which are different from the preceding examples. We must define transition times and create a transition table; we must define activity types and set them for the activities; we must define a function for building the different houses. The additional complexity of the problem will also require a slightly different search strategy.

Setting the Makespan Variable and Calculating the Horizon

Previously, the makespan was set to the end of the last activity (moving). In this example, with three houses being built, the makespan is defined as an `IloNumVar` equal to or greater than each moving activity, so that the house with the longest build time increments the makespan appropriately.

We are going to set a *horizon* in this example, or latest potential ending point in time. When calculating the horizon, we must guarantee that all activities will be completed by that time.

We set the horizon in this example because it helps define the outward time boundary for building the three houses, thereby limiting the potential solution search space.

We have only one worker, so we can start calculating the horizon by adding the durations for all activities of all three houses: 29 + 47 + 47 = 123. We are defining the minimum transition time between differing activities to be three; therefore, if all activities follow activities of a different type, the total minimum transition time would be 87. The time required to build all three houses would then be 123 + 87 = 210. Since it's unlikely that all the transition time will be required, we can safely set the horizon to 200.

```
/* CREATE THE MAKESPAN VARIABLE. */
IloNum horizon = 200;
makespan = IloNumVar(env, 0, horizon, IloNumVar::Int);
/* ... */
model.add(moving.endsBefore(makespan));
```

Define a Function for Building the Houses

The function `MakeHouse` describes the problem of building a house.

```
void MakeHouse(IloModel model,
               const IloNum* dur,
               const IloNum startMin,
               const IloNum endMax,
               const IloUnaryResource worker,
               const IloNumVar makespan)
{
```

The following lines of code define the three houses to be built. Each house design uses a different array to define the durations of the activities (a different `durationsDesign`). The earliest start and latest end dates for the first and third houses are the origin and horizon of the schedule, but the second house has an earliest start date of 5 and a latest end date of 80.

```
/* CREATE THE ACTIVITIES AND CONSTRAINTS FOR THE HOUSES. */
MakeHouse(model, durationsDesign1, 0, horizon, worker, makespan);
MakeHouse(model, durationsDesign2, 5, 80, worker, makespan);
MakeHouse(model, durationsDesign3, 0, horizon, worker, makespan);
```

Set Activity Durations

The following lines create the arrays of the activity durations.

```
/* THREE DESIGNS OF HOUSES EACH HAVING DIFFERENT TASK DURATIONS */

IloNum durationsDesign1 [] = { 7, 3,  8, 3, 1, 2, 1, 2, 1, 1};
IloNum durationsDesign2 [] = {12, 5, 10, 5, 2, 5, 2, 3, 2, 1};
IloNum durationsDesign3 [] = {15, 3, 10, 6, 2, 3, 2, 3, 2, 1};
```

Create Activities

The following code creates the ten activities `masonry`, `carpentry`, `plumbing`, `ceiling`, `roofing`, `painting`, `windows`, `facade`, `garden`, and `moving`. The argument `dur` accesses the `durationsDesign` array. The third argument, the integer 0 through 9, is the transition type. The transition type is used in the next section to define the transition time.

```
/* CREATE THE ACTIVITIES. */
IloActivity masonry(env,   dur[0], 1, “masonry   “);
IloActivity carpentry(env, dur[1], 2, “carpentry “);
IloActivity plumbing(env,  dur[2], 3, “plumbing  “);
IloActivity ceiling(env,   dur[3], 4, “ceiling   “);
IloActivity roofing(env,   dur[4], 5, “roofing   “);
IloActivity painting(env,  dur[5], 6, “painting  “);
IloActivity windows(env,   dur[6], 7, “windows   “);
IloActivity facade(env,    dur[7], 8, “facade    “);
IloActivity garden(env,    dur[8], 9, “garden    “);
IloActivity moving(env,    dur[9], 0, “moving    “);
```

Define Transition Times

The function `GetTransitionTime` is used to calculate the transition time between two activities of different transition type. The transition time is defined to be the greater of either 3, or half the absolute value of the difference between the two transition types. For example, the transition time between a masonry activity (type = 1), and a garden activity (type = 9), is 4.

```
IloNum GetTransitionTime(IloInt type1, IloInt type2) {
  if (!type1 || !type2 || type1==type2)
    return 0;
  return IloMax(3L, IloAbs((type1 - type2)/2L));
}
```

Create the Table for the Transition Times

The class `IloTransitionParam` lets you build transition times and costs to apply to a resource. An instance of this class is a table of non-negative numbers, indexed by the transition type of the activities.

The `IloTrue` parameter indicates that the table is symmetric—that is, that the transition time is the same whether *Act1* precedes *Act2* or *Act2* precedes *Act1*. The transition time is

then passed to the resource with an instance of the class `IloTranstionTime` (in the next code section).

```
const IloInt numberOfTypes = 10;

  /* CREATE THE TRANSITION TABLE */

  IloTransitionParam ttParam(env, numberOfTypes, IloTrue);
  IloInt i, j;
  for(i = 0; i < numberOfTypes; ++i) {
    for(j = 0; j <= i; ++j) {
      ttParam.setValue(i, j, GetTransitionTime(i, j));
    }
  }
```

Create the Resource

The following lines create the worker as an object of the `IloUnaryResource` class. The transition time object is bound to the resource to ensure that the transition times are taken into consideration. The enforcement level `IloMediumHigh` means that Scheduler will spend more effort at enforcing the constraint than it would by default.

```
  /* CREATE THE RESOURCE. */
  IloUnaryResource worker(env);
  IloTransitionTime tt(worker, ttParam);
  worker.setCapacityEnforcement(IloMediumHigh);
```

Set Temporal Constraints

The temporal constraints in this problem are the same as in all previous examples. However, we also add two constraints on the minimum start date and maximum end date.

```
  /* SET STARTMIN AND ENDMAX. */
  model.add(masonry.startsAfter(startMin));
  model.add(moving.endsBefore(endMax));
```

Solve the Problem

The function `IloRankForward` returns a goal that ranks all resource constraints while minimizing `makespan`. By default, `IloRankForward` uses the resource selector `IloSelResMinGlobalSlack` to select the next resource to be ranked, and the resource constraint selector `IloSelFirstRCMinStartMax` to select the next resource constraint to be ranked first. `IloSelResMinGlobalSlack` selects a resource on which not all activities are ordered and that has minimal slack. (Slack is the difference between the demand for a

resource and its availability over a specific period of time. For more information, see `IlcDiscreteResource` in the *IBM ILOG Scheduler Reference Manual*.)

```
IloEnv env;
IloNumVar makespan;
IloModel model = DefineModel(env, makespan);

IloSolver solver(model);

IloGoal goal = IloRankForward(env, makespan);
```

Complete Program and Output—Example 5

You can see the entire program `gsTTime.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

#if defined(ILO_SDXLOUTPUT)
#include “sdxloutput.h”
#endif

////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////
/* THREE DESIGNS OF HOUSES EACH HAVING DIFFERENT TASK DURATIONS */

IloNum durationsDesign1 [] = { 7, 3,  8, 3, 1, 2, 1, 2, 1, 1};
IloNum durationsDesign2 [] = {12, 5, 10, 5, 2, 5, 2, 3, 2, 1};
IloNum durationsDesign3 [] = {15, 3, 10, 6, 2, 3, 2, 3, 2, 1};
const IloInt numberOfTypes = 10;
void MakeHouse(IloModel model,
               const IloNum* dur,
               const IloNum startMin,
               const IloNum endMax,
               const IloUnaryResource worker,
               const IloNumVar makespan)
{
  IloEnv env = model.getEnv();

  /* CREATE THE ACTIVITIES. */
  IloActivity masonry(env,   dur[0], 1, “masonry   “);
  IloActivity carpentry(env, dur[1], 2, “carpentry “);
  IloActivity plumbing(env,  dur[2], 3, “plumbing  “);
  IloActivity ceiling(env,   dur[3], 4, “ceiling   “);
  IloActivity roofing(env,   dur[4], 5, “roofing   “);
  IloActivity painting(env,  dur[5], 6, “painting  “);
  IloActivity windows(env,   dur[6], 7, “windows   “);
  IloActivity facade(env,    dur[7], 8, “facade    “);
```

```
  IloActivity garden(env,    dur[8], 9, "garden    ");
  IloActivity moving(env,    dur[9], 0, "moving    ");
  /* SET STARTMIN AND ENDMAX. */
  model.add(masonry.startsAfter(startMin));
  model.add(moving.endsBefore(endMax));
  /* ADD THE TEMPORAL CONSTRAINTS. */
  model.add(carpentry.startsAfterEnd(masonry));
  model.add(plumbing.startsAfterEnd(masonry));
  model.add(ceiling.startsAfterEnd(masonry));
  model.add(roofing.startsAfterEnd(carpentry));
  model.add(painting.startsAfterEnd(ceiling));
  model.add(windows.startsAfterEnd(roofing));
  model.add(facade.startsAfterEnd(roofing));
  model.add(facade.startsAfterEnd(plumbing));
  model.add(garden.startsAfterEnd(roofing));
  model.add(garden.startsAfterEnd(plumbing));
  model.add(moving.startsAfterEnd(windows));
  model.add(moving.startsAfterEnd(facade));
  model.add(moving.startsAfterEnd(garden));
  model.add(moving.startsAfterEnd(painting));
  model.add(moving.endsBefore(makespan));
  /* ADD THE RESOURCE CONSTRAINTS. */
  model.add(carpentry.requires(worker));
  model.add(ceiling.requires(worker));
  model.add(roofing.requires(worker));
  model.add(windows.requires(worker));
  model.add(facade.requires(worker));
  model.add(masonry.requires(worker));
  model.add(plumbing.requires(worker));
  model.add(garden.requires(worker));
  model.add(painting.requires(worker));
  model.add(moving.requires(worker));
}

IloNum GetTransitionTime(IloInt type1, IloInt type2) {
  if (!type1 || !type2 || type1==type2)
    return 0;
  return IloMax(3L, IloAbs((type1 - type2)/2L));
}

IloModel DefineModel(const IloEnv env, IloNumVar& makespan)
{
  IloModel model(env);
  /* CREATE THE MAKESPAN VARIABLE. */
  IloNum horizon = 200;
  makespan = IloNumVar(env, 0, horizon, IloNumVar::Int);
  /* CREATE THE TRANSITION TABLE */

  IloTransitionParam ttParam(env, numberOfTypes, IloTrue);
  IloInt i, j;
  for(i = 0; i < numberOfTypes; ++i) {
    for(j = 0; j <= i; ++j) {
      ttParam.setValue(i, j, GetTransitionTime(i, j));
    }
  }
  /* CREATE THE RESOURCE. */
  IloUnaryResource worker(env);
  IloTransitionTime tt(worker, ttParam);
```

```
  worker.setCapacityEnforcement(IloMediumHigh);
  /* CREATE THE ACTIVITIES AND CONSTRAINTS FOR THE HOUSES. */
  MakeHouse(model, durationsDesign1, 0, horizon, worker, makespan);
  MakeHouse(model, durationsDesign2, 5, 80, worker, makespan);
  MakeHouse(model, durationsDesign3, 0, horizon, worker, makespan);

  /* RETURN THE CREATED MODEL. */
  return model;
}

////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////////////////

void PrintSolution(const IloSolver solver, const IloNumVar makespan)
{
  IlcScheduler scheduler(solver);
  IloEnv env = solver.getEnv();
  env.out() << "Solution with makespan "
            << solver.getIntVar(makespan).getMin() << endl;
  for(IloIterator<IloActivity> act(env);
      act.ok();
      ++act)
    env.out() << scheduler.getActivity(*act) << endl;
  solver.printInformation();
}

////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////

int main()
{
  try {
    IloEnv env;
    IloNumVar makespan;
    IloModel model = DefineModel(env, makespan);

    IloSolver solver(model);
    IloGoal goal = IloRankForward(env, makespan);
    if (solver.solve(goal)) {
      PrintSolution(solver,makespan);

#if defined(ILO_SDXLOUTPUT)
      IloSDXLOutput output(env);
      ofstream outFile("gsTTime.xml");
      output.write(IlcScheduler(solver), outFile, solver.getIntVar(makespan));
      outFile.close();
#endif
    }
    else
      solver.out() << "No solution !" << endl;

    env.end();
```

```
  } catch (IloException& exc) {
    cout << exc << endl;
  }
  return 0;
}

////////////////////////////////////////////////////////////////////////////
//
// RESULTS
//
////////////////////////////////////////////////////////////////////////////

/*
  Solution with makespan 173
  masonry    [0 -- 7 --> 7]
  carpentry  [134 -- 3 --> 137]
  plumbing   [108 -- 8 --> 116]
  ceiling    [125 -- 3 --> 128]
  roofing    [142 -- 1 --> 143]
  painting   [149 -- 2 --> 151]
  windows    [170 -- 1 --> 171]
  facade     [157 -- 2 --> 159]
  garden     [164 -- 1 --> 165]
  moving     [172 -- 1 --> 173]
  masonry    [7 -- 12 --> 19]
  carpentry  [43 -- 5 --> 48]
  plumbing   [22 -- 10 --> 32]
  ceiling    [35 -- 5 --> 40]
  roofing    [59 -- 2 --> 61]
  painting   [51 -- 5 --> 56]
  windows    [75 -- 2 --> 77]
  facade     [64 -- 3 --> 67]
  garden     [70 -- 2 --> 72]
  moving     [77..79 -- 1 --> 78..80]
  masonry    [80 -- 15 --> 95]
  carpentry  [131 -- 3 --> 134]
  plumbing   [98 -- 10 --> 108]
  ceiling    [119 -- 6 --> 125]
  roofing    [140 -- 2 --> 142]
  painting   [146 -- 3 --> 149]
  windows    [168 -- 2 --> 170]
  facade     [154 -- 3 --> 157]
  garden     [162 -- 2 --> 164]
  moving     [171 -- 1 --> 172]
*/
```

The start and end times of the activities are fixed, except for the moving activity of house Design2, which can start at any time between day 77 and 79 and end between day 78 and 80. The total time to complete the project is 173 days. Notice that our requirement that house Design2 be finished by day 80 has resulted in a longer makespan than would have occurred if we had been able to schedule all activities of the same type together.

Figure 5.1 provides a graphic display of the solution to our problem. The sole activity with variable start and end times is positioned at its earliest start time; its latest end time is indicated with an arrow.

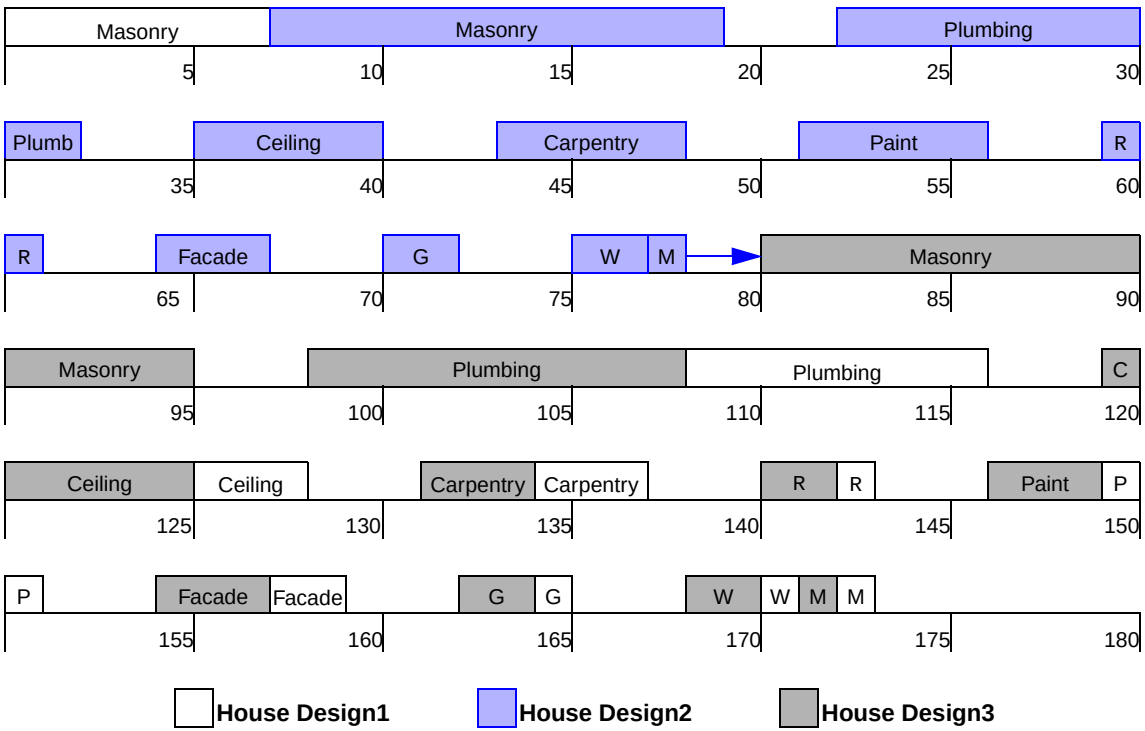


Figure 5.1 *Building Houses with Transition Times*

C H A P T E R

6

Adding Breaks

A resource *break* is a time interval during which the resource is not available for any activity. For example, breaks could include weekend or vacation time for workers, or maintenance time for machines. Breaks have a start time, an end time, and a duration, which can be either positive or null.

The example in this chapter shows how to model a problem that includes weekend breaks for all workers. The example includes some activities which cannot be interrupted by breaks, which means they must be started and finished in an interval between breaks.

Describe the Problem—Example 6

This example presents a slightly different problem. In it we have four workers who are required to build five houses. Each of the workers has different abilities: one performs `masonry`, the second performs `plumbing`, the third performs the `carpentry`, `ceiling`, `roofing`, `windows`, and `facade` activities, and the final worker performs `painting`, `garden`, and `moving` activities. The workers do not work on weekends (that is, work is suspended).

The five houses have the three designs, and the corresponding activity durations, shown in Table 5.1 in Adding Transition Times. Three of the activities, `roofing`, `painting`, and `moving`, cannot be suspended by breaks. Furthermore, there are time constraints on the third

and fourth houses that state that work on those houses cannot be started before day 7 and must be finished by day 79.

Once again, our objective is to minimize the makespan of the problem.

Define the Problem, Design a Model

In order to model the weekend breaks, we need to define a list of breaks, and add those breaks to our resources.

There are four workers in the problem, but since their abilities are not equal they cannot be defined as a discrete resource. Since each worker can only perform one activity at a time, the workers must be defined as unary resources.

We also need to define the activities that can and cannot be suspended by breaks.

As in Example 5 in Adding Transition Times, we define a function for building the different houses and use a ranking goal in our search for a solution.

Calculate the Horizon

We have four workers, which allows several activities to be performed at the same time, so it is difficult to calculate a horizon for this problem.

One worker performs all the masonry, which must be completed before any other activities can be performed on a house. We could calculate the maximum possible time requirement for the masonry worker (79) and add the maximum possible time requirements for the remaining activities on house 5 (25) to get a potential horizon of 104 days.

We could also calculate the maximum possible time requirement for the worker who performs the carpentry, ceiling, roofing, facade, and windows tasks (including a maximum delayed start of 21 days) to get a potential horizon of 137. We set the horizon at 150 to allow for activities, like moving, that must be performed after he completes his work.

```
/* CREATE THE MAKESPAN VARIABLE. */
IloNum horizon = 150;
makespan = IloNumVar(env, 0, horizon, IloNumVar::Int);
```

Define a Function for Building the Houses

The function `MakeHouse` describes the process of building a house.

```
void MakeHouse(IloModel model,
               const IloNum* dur,
               const IloNum startMin,
               const IloNum endMax,
               const IloUnaryResourceArray workers,
               const IloNumVar makespan)
```

The following lines of code define the five houses to be built. The earliest start and latest end dates for the first, second, and fifth houses are the origin and horizon of the schedule, but the third and fourth houses have an earliest start date of 7 and a latest end date of 79.

```
/* CREATE THE ACTIVITIES AND CONSTRAINTS FOR THE HOUSES. */
MakeHouse(model, durationsDesign1, 0, horizon, workers, makespan);
MakeHouse(model, durationsDesign1, 0, horizon, workers, makespan);
MakeHouse(model, durationsDesign2, 7, 79, workers, makespan);
MakeHouse(model, durationsDesign2, 7, 79, workers, makespan);
MakeHouse(model, durationsDesign3, 0, horizon, workers, makespan);
```

Create the Activities

By default, activities are created as non-breakable. The function `setBreakable` is used to define breakable activities (i.e., to define activities that can be suspended by breaks). `setBreakable` is used on all activities in this example except `roofing`, `painting`, and `moving`. As described previously, the duration is specified using the index of the duration design array.

```
/* CREATE THE ACTIVITIES. */
IloActivity masonry(env,   dur[0], “masonry   “);
masonry.setBreakable();

IloActivity carpentry(env, dur[1], “carpentry “);
carpentry.setBreakable();

IloActivity plumbing(env,  dur[2], “plumbing  “);
plumbing.setBreakable();

IloActivity ceiling(env,   dur[3], “ceiling   “);
ceiling.setBreakable();

IloActivity roofing(env,   dur[4], “roofing   “);
IloActivity painting(env,  dur[5], “painting  “);

IloActivity windows(env,   dur[6], “windows   “);
windows.setBreakable();

IloActivity facade(env,    dur[7], “facade    “);
facade.setBreakable();
```

```
IloActivity garden(env,    dur[8], “garden    “);
garden.setBreakable();

IloActivity moving(env,    dur[9], “moving    “);
```

Create the Breaks

The weekend breaks are created as an instance of the class `IloIntervalList`, and added to environment `env` with a range that spans the entire schedule (zero through `horizon`). The member function `addPeriodicInterval` defines when the break starts (5), how long it lasts (2), and the period interval (7). This break list is an example of *parameter sharing* within Concert Technology; several resources are sharing the same break list.

```
/* CREATE A SHARED BREAK LIST FOR ALL WORKERS */
IloIntervalList breakParam = schedEnv.getBreakListParam();
breakParam.addPeriodicInterval(5, 2, 7, horizon);
```

Set the Enforcement Level

The member function `IloResourceParam ::setCapacityEnforcement` allows you to set the enforcement level on resource usage. The enforcement level `IloMediumHigh` means that Scheduler will spend more effort at enforcing this constraint than it would by default.

```
/* SET ENFORCEMENT LEVEL */
IloSchedulerEnv schedEnv(env);
IloResourceParam resParam = schedEnv.getResourceParam();
resParam.setCapacityEnforcement(IloMediumHigh);
```

Create the Resources

The four workers in this example are created as an array of an instance of `IloUnaryResource`.

```
/* CREATE THE RESOURCES. */
IloInt nbOfWorkers = 4;
IloUnaryResourceArray workers(env,nbOfWorkers);
for (IloInt k = 0; k < nbOfWorkers; k++)
  workers[k] = IloUnaryResource(env);
```

Add the Resource Constraints

The `requires` resource constraint of activities are added to the model using the function `IloModel::add`. The workers are specified using an index on the resource array `workers`.

```
/* POST THE RESOURCE CONSTRAINTS. */
model.add(carpentry.requires(workers[0]));
model.add(ceiling.requires(workers[0]));
model.add(roofing.requires(workers[0]));
model.add(windows.requires(workers[0]));
model.add(facade.requires(workers[0]));

model.add(masonry.requires(workers[1]));

model.add(plumbing.requires(workers[2]));

model.add(garden.requires(workers[3]));
model.add(painting.requires(workers[3]));
model.add(moving.requires(workers[3]));
```

Define the Objective

The Concert Technology function `IloMinimize` is used to set the objective of minimizing the makespan.

```
/* SET THE OBJECTIVE */
model.add(IloMinimize(env, makespan));
```

Solve the Problem

To solve the problem, we use `IloRankForward`, as described previously in Solve the Problem.

```
// Model
IloEnv env;
IloNumVar makespan;
IloModel model = DefineModel(env, makespan);

// Algorithm
IloSolver solver(model);
IloGoal goal = IloRankForward(env, makespan);
```

Complete Program and Output—Example 6

You can see the entire program `gsBreak.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

#if defined(ILO_SDXLOUTPUT)
#include “sdxloutput.h”
#endif

////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////

/* THREE DESIGNS OF HOUSES EACH HAVING DIFFERENT TASK DURATIONS */

IloNum durationsDesign1 [] = { 7, 3,  8, 3, 1, 2, 1, 2, 1, 1};
IloNum durationsDesign2 [] = {12, 5, 10, 5, 2, 5, 2, 3, 2, 1};
IloNum durationsDesign3 [] = {15, 3, 10, 6, 2, 3, 2, 3, 2, 1};

void MakeHouse(IloModel model,
               const IloNum* dur,
               const IloNum startMin,
               const IloNum endMax,
               const IloUnaryResourceArray workers,
               const IloNumVar makespan)
{
  IloEnv env = model.getEnv();
  /* CREATE THE ACTIVITIES. */
  IloActivity masonry(env,   dur[0], “masonry   “);
  masonry.setBreakable();

  IloActivity carpentry(env, dur[1], “carpentry “);
  carpentry.setBreakable();

  IloActivity plumbing(env,  dur[2], “plumbing  “);
  plumbing.setBreakable();

  IloActivity ceiling(env,   dur[3], “ceiling   “);
  ceiling.setBreakable();

  IloActivity roofing(env,   dur[4], “roofing   “);
  IloActivity painting(env,  dur[5], “painting  “);

  IloActivity windows(env,   dur[6], “windows   “);
  windows.setBreakable();

  IloActivity facade(env,    dur[7], “facade    “);
  facade.setBreakable();

  IloActivity garden(env,    dur[8], “garden    “);
```

```
  garden.setBreakable();

  IloActivity moving(env,    dur[9], "moving    ");
  /* SET EARLIEST START */
  model.add(masonry.startsAfter(startMin));
  model.add(moving.endsBefore(endMax));

  /* POST THE TEMPORAL CONSTRAINTS. */
  model.add(carpentry.startsAfterEnd(masonry));
  model.add(plumbing.startsAfterEnd(masonry));
  model.add(ceiling.startsAfterEnd(masonry));
  model.add(roofing.startsAfterEnd(carpentry));
  model.add(painting.startsAfterEnd(ceiling));
  model.add(windows.startsAfterEnd(roofing));
  model.add(facade.startsAfterEnd(roofing));
  model.add(facade.startsAfterEnd(plumbing));
  model.add(garden.startsAfterEnd(roofing));
  model.add(garden.startsAfterEnd(plumbing));
  model.add(moving.startsAfterEnd(windows));
  model.add(moving.startsAfterEnd(facade));
  model.add(moving.startsAfterEnd(garden));
  model.add(moving.startsAfterEnd(painting));

  /* SET MAKESPAN */
  model.add(moving.endsBefore(makespan));
  /* POST THE RESOURCE CONSTRAINTS. */
  model.add(carpentry.requires(workers[0]));
  model.add(ceiling.requires(workers[0]));
  model.add(roofing.requires(workers[0]));
  model.add(windows.requires(workers[0]));
  model.add(facade.requires(workers[0]));

  model.add(masonry.requires(workers[1]));

  model.add(plumbing.requires(workers[2]));

  model.add(garden.requires(workers[3]));
  model.add(painting.requires(workers[3]));
  model.add(moving.requires(workers[3]));
}

IloModel DefineModel(const IloEnv env, IloNumVar& makespan)
{
  IloModel model(env);

  /* CREATE THE MAKESPAN VARIABLE. */
  IloNum horizon = 150;
  makespan = IloNumVar(env, 0, horizon, IloNumVar::Int);
  /* SET ENFORCEMENT LEVEL */
  IloSchedulerEnv schedEnv(env);
  IloResourceParam resParam = schedEnv.getResourceParam();
  resParam.setCapacityEnforcement(IloMediumHigh);
  /* CREATE A SHARED BREAK LIST FOR ALL WORKERS */
  IloIntervalList breakParam = schedEnv.getBreakListParam();
  breakParam.addPeriodicInterval(5, 2, 7, horizon);
  /* CREATE THE RESOURCES. */
  IloInt nbOfWorkers = 4;
  IloUnaryResourceArray workers(env,nbOfWorkers);
```

```
  for (IloInt k = 0; k < nbOfWorkers; k++)
    workers[k] = IloUnaryResource(env);
  /* CREATE THE ACTIVITIES AND CONSTRAINTS FOR THE HOUSES. */
  MakeHouse(model, durationsDesign1, 0, horizon, workers, makespan);
  MakeHouse(model, durationsDesign1, 0, horizon, workers, makespan);
  MakeHouse(model, durationsDesign2, 7, 79, workers, makespan);
  MakeHouse(model, durationsDesign2, 7, 79, workers, makespan);
  MakeHouse(model, durationsDesign3, 0, horizon, workers, makespan);
  /* SET THE OBJECTIVE */
  model.add(IloMinimize(env, makespan));
  return model;
}

////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////////////////

void PrintSolution(const IloSolver solver, const IloNumVar makespan)
{
  IlcScheduler scheduler(solver);
  IloEnv env = solver.getEnv();
  env.out() << "Solution with makespan "
            << solver.getIntVar(makespan).getMin() << endl;
  for(IloIterator<IloActivity> act(env);
      act.ok();
      ++act)
    env.out() << scheduler.getActivity(*act) << endl;
  solver.printInformation();
}

////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////

int main()
{
  try {
    // Model
    IloEnv env;
    IloNumVar makespan;
    IloModel model = DefineModel(env, makespan);

    // Algorithm
    IloSolver solver(model);
    IloGoal goal = IloRankForward(env, makespan);

    if (solver.solve(goal)) {
      PrintSolution(solver,makespan);
#if defined(ILO_SDXLOUTPUT)
      IloSDXLOutput output(env);
      ofstream outFile("gsBreak.xml");
      output.write(IlcScheduler(solver), outFile, solver.getIntVar(makespan));
      outFile.close();
#endif
```

```
    }
    else
      solver.out() << "No solution !" << endl;

    env.end();

  } catch (IloException& exc) {
    cout << exc << endl;
  }
  return 0;
}

////////////////////////////////////////////////////////////////////////////
//
// RESULTS
//
////////////////////////////////////////////////////////////////////////////

/*
  Solution with makespan 113
  masonry    [64..78 -- (7) 7..11 --> 73..87]
  carpentry  [87 -- (3) 5 --> 92]
  plumbing   [78..95 -- (8) 8..12 --> 88..107]
  ceiling    [94 -- (3) 5 --> 99]
  roofing    [102 -- 1 --> 103]
  painting   [99..106 -- 2 --> 101..108]
  windows    [109 -- (1) 1 --> 110]
  facade     [107 -- (2) 2 --> 109]
  garden     [105..108 -- (1) 1 --> 106..109]
  moving     [112 -- 1 --> 113]
  masonry    [0..1 -- (7) 9 --> 9..10]
  carpentry  [9..14 -- (3) 3..5 --> 12..17]
  plumbing   [9..14 -- (8) 8..12 --> 19..24]
  ceiling    [14..17 -- (3) 3..5 --> 17..22]
  roofing    [17..22 -- 1 --> 18..23]
  painting   [17..50 -- 2 --> 19..52]
  windows    [18..23 -- (1) 1 --> 19..24]
  facade     [21..24 -- (2) 2 --> 23..26]
  garden     [21..52 -- (1) 1 --> 22..53]
  moving     [23..53 -- 1 --> 24..54]
  masonry    [25..28 -- (12) 16..18 --> 43..44]
  carpentry  [44 -- (5) 7 --> 51]
  plumbing   [43..51 -- (10) 10..14 --> 57..65]
  ceiling    [51 -- (5) 7 --> 58]
  roofing    [58 -- 2 --> 60]
  painting   [70 -- 5 --> 75]
  windows    [72 -- (2) 2 --> 74]
  facade     [65 -- (3) 3 --> 68]
  garden     [60..66 -- (2) 2..4 --> 64..68]
  moving     [78 -- 1 --> 79]
  masonry    [9..10 -- (12) 16 --> 25..26]
  carpentry  [32..35 -- (5) 5..7 --> 39..40]
  plumbing   [25..37 -- (10) 10..14 --> 39..51]
  ceiling    [25..28 -- (5) 5..7 --> 32..33]
  roofing    [42 -- 2 --> 44]
  painting   [35..56 -- 5 --> 40..61]
  windows    [70 -- (2) 2 --> 72]
```

```
  facade     [60 -- (3) 5 --> 65]
  garden     [44..64 -- (2) 2..4 --> 46..66]
  moving     [77 -- 1 --> 78]
  masonry    [43..53 -- (15) 15..21 --> 64..74]
  carpentry  [84 -- (3) 3 --> 87]
  plumbing   [64..81 -- (10) 10..14 --> 78..95]
  ceiling    [74 -- (6) 8 --> 82]
  roofing    [92 -- 2 --> 94]
  painting   [84..99 -- 3 --> 87..102]
  windows    [105 -- (2) 2 --> 107]
  facade     [99 -- (3) 3 --> 102]
  garden     [94..102 -- (2) 2..4 --> 96..106]
  moving     [107..109 -- 1 --> 108..110]
*/
```

The start and end times of some activities are fixed, while others have variable start and end dates. The total time to complete the project is 113 days.

The results are interpreted as follows:

`moving [107..109 -- 1 --> 108..110]` The moving activity has an earliest start time of 107, a latest start time of 109, a duration of 1, an earliest completion time of 108, and a latest completion time of 110.

`garden [94..102 -- (2) 2..4 --> 96..106]` The garden activity has an earliest start time of 94, a latest start time of 102, a processing time of 2, a minimal duration of 2, a maximal duration of 4, an earliest completion time of 96, and a latest completion time of 106.

`facade [60 -- (3) 5 --> 65]` The facade activity has a start time of 60, a processing time of 3, a duration of 5, and a completion time of 65.

Figure 6.1 provides a graphic display of the solution to our problem. Activities with variable start and end times are positioned at their earliest start times.

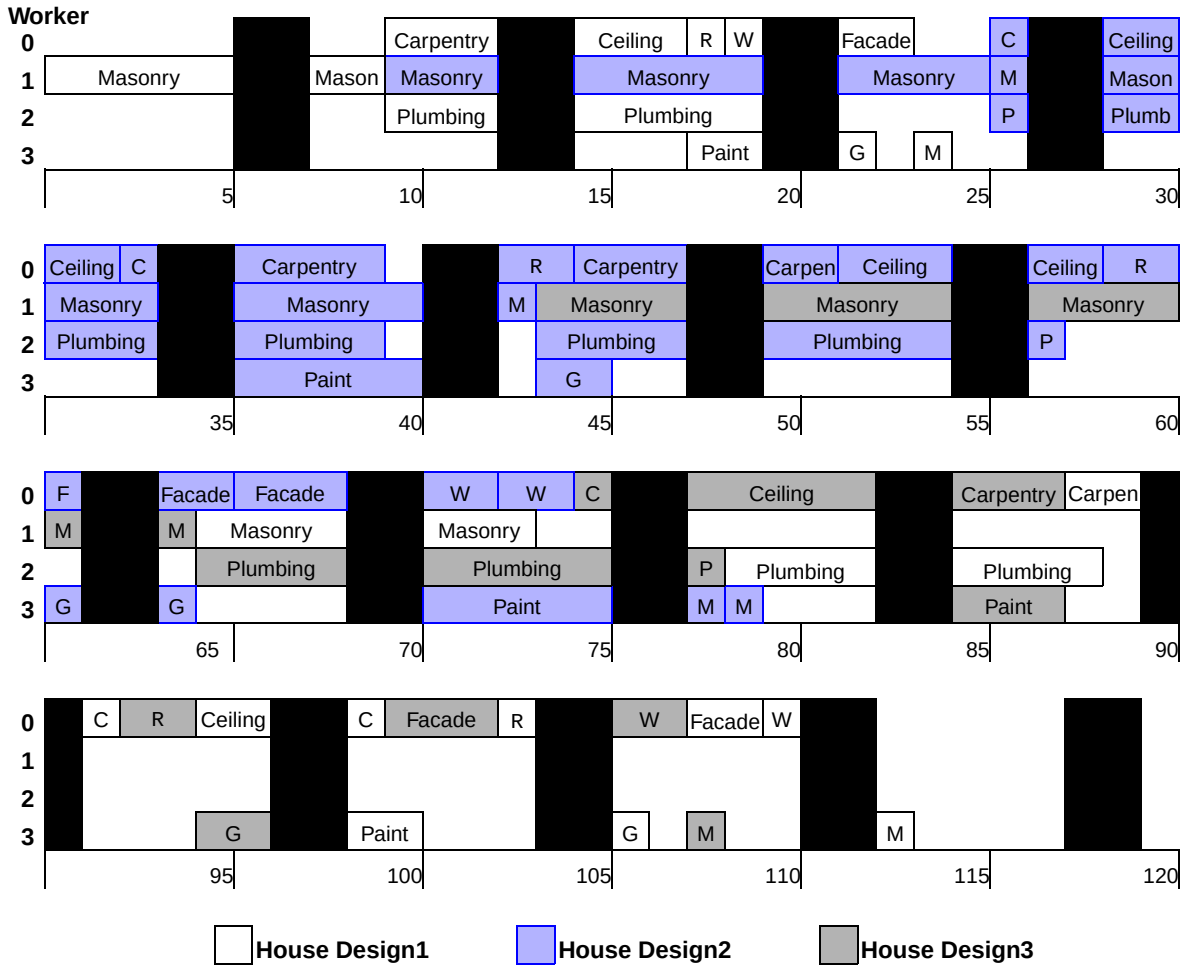


Figure 6.1 *Building Houses with Breaks*

C H A P T E R

7

Adding Integral and Functional Constraints

Integral and functional constraints are used to express natural dependencies, such as “the total cost incurred to execute an activity is the sum of the daily expenses over its duration,” or “prices are cut by 30% for a car rental of at least one week, and by 50% for a rental of at least two weeks.” The example in this chapter presents a finer model of weekend breaks (compared to Adding Breaks); an integral constraint is used to reduce efficiency when processing an activity during weekends. In addition, some dependencies between duration and cost variables are modeled using stepwise functions to better achieve the complex relationships needed in the model.

Describe the Problem—Example 7

Starting with the simple model of Using the Building Blocks, we now wish to take into account the worker's part-time days. Our worker does not work on Wednesday and Saturday afternoons, so we need to specify that these days, if worked, only contribute 50% towards the processing time of an activity.

As before, Sunday is a full day off, so its contribution is 0%. In addition, there are two other days during the schedule when the worker is not available (Tuesday of the second week and Monday of the third week).

Each activity has a variable cost that depends on the activity duration. The longer the activity, the longer we must hire the worker and his tools. Our objective is to minimize the

total cost of the house, that is, to supply the worker, tools, and materials at the lowest possible cost for the length of time necessary. Additionally, although it is not an objective, we would like the house to be built within 50 days, even if it increases the final cost. We will use the `IloGenerate` and `IloSetTimesForward` goals in our search for a solution.

Define the Problem, Design a Model

Minimizing the makespan of the problem is not the objective, and we set a horizon spanning ten weeks of work, which leaves enough room for performing the activities. However, as we wish the house to be finished in less than 50 days, we must limit the end time of the `moving` activity.

```
model.add( moving.endsBefore(MaxMakespan) );
```

We will set `MaxMakespan` equal to 50, thus specifying that even if minimizing the makespan is not the real objective, it does count as a trade-off in the quality of the solution.

As in Chapter 3, the worker is modeled by a unary resource, and building the house consists in a set of activities constrained by temporal precedences.

Worker's efficiency

We use a granular function in order to model the worker's efficiency. The granularity specified when constructing this `IloGranularFunction` is 100, so this is then the default initial value over the definition interval `[0, horizon)`. We set the part-time efficiency for Wednesday and Saturday (50%), as well as the break on Sunday (0%), repeatedly over the time horizon using the `setValue` member function.

```
/* MODEL WORKER'S EFFICIENCY */
IloNum day;
IloGranularFunction efficiency(env, 0, horizon, 100);
for(day=0; day+7<=horizon; day+=7) {
  efficiency.setValue(day+2, day+3, 50);  // part-time wednesday
  efficiency.setValue(day+5, day+6, 50);  // saturdays
  efficiency.setValue(day+6, day+7, 0);   // sundays
}
efficiency.setValue( 8, 9, 0);  // Tuesday of week #2 is a day off
efficiency.setValue(14,15, 0);  // Monday of week #3 is a day off
```

The exceptional days off are also modeled with a zero efficiency.

> ***Note:*** *We have chosen to express the efficiency with a percentage. Hence, the granularity of the* `IloGranularFunction` *is set to 100 at construction time. An alternative method of modeling would be to split the working days into mornings and afternoons, and use a granularity of 2 without loss of precision.*

We must specify that the worker works at this reduced efficiency for *every* activity he might be asked to process. Hence we add to the model the following integral constraint.

```
model.add( IloResourceIntegralConstraint(worker,
                                         IloProcessingTimeVariable,
                                         efficiency));
```

This constraint binds the processing time of every activity executed on the unary resource `worker` to be equal to the sum of the function over its duration interval `[start,end]`.

By default, an activity is non-breakable. We use the `IloActivity` member function `setBreakable` to declare an activity breakable.

Budget

After an activity has started, each worked day costs a decreasing amount of money. Days 0 to 2 are charged $1000 each, days 3 to 6 are charged $900 each, and all days including or after day 7 cost $700 each. The actual cost of an activity is then a function of its duration.

```
/* CREATE COST CURVE */
IloNum dailyCost = 0;
IloNum overallCost = 0;
IloGranularFunction costFunction(env, 0, horizon);
for(day=0; day<horizon; day+=1) {
  costFunction.setValue(day, day+1, overallCost);
  if (day<3) dailyCost = 1000.0;
  else if (day<7) dailyCost = 900.0;
  else dailyCost = 700.0;
  overallCost += dailyCost;
}
```

Once the granular function is built, we constrain each activity's external variable to be equal to the total cost incurred due to the activity's duration, and add this constraint to the model.

```
model.add( IloResourceFunctionalConstraint(worker,
                                           IloExternalVariable,
                                           costFunction,
                                           IloDurationVariable) );
```

Define the Objective

We introduce a new variable constrained with `IloSum` to be equal to the total sum of each individual activity's cost. The Concert Technology function `IloMinimize` is used to set the objective of minimizing this total cost.

```
/* SET THE OBJECTIVE: MINIMIZE THE TOTAL COST VARIABLE */
totalCost = IloNumVar(env, 0, maxBudget, ILOINT);
model.add( totalCost == IloSum(costs) );
model.add( IloMinimize(env, totalCost) );
```

Solve the Problem

To solve the problem, we use a combination of the Concert Technology `IloGenerate` goal and Scheduler's `IloSetTimesForward`.

```
IloEnv env;
IloNumVar totalCost;
IloNumVarArray costs;
IloSchedulerSolution solution(env);

IloModel model = DefineModel(env, maxMakespan, totalCost, costs, solution);

IloSolver solver(model);
IloGoal goal = IloGenerate(env, costs) &&
               IloSetTimesForward(env);
```

`IloGenerate` will first instantiate the costs to their minimal possible value. Then, `IloSetTimesForward` will schedule the activities at a convenient start time, given their duration.

Complete Program and Output—Example 7

You can see the entire program `gsEff.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////
```

```
const IloNum dur[] = { 7, 3, 8, 3, 1, 2, 1, 2, 1, 1};

const IloInt NumberOfActivities = 10;
const IloNum horizon = 10*7; /* MAXIMAL SCHEDULE OF TEN WEEKS */
const IloNum maxBudget = 50000;
const IloNum maxMakespan = 50;

IloModel DefineModel(IloEnv env,
                     IloNum MaxMakespan,
                     IloNumVar &totalCost,
                     IloNumVarArray& costs,
                     IloSchedulerSolution solution)
{
  IloModel model(env);

  IloSchedulerEnv schedEnv(env);
  schedEnv.getBreakListParam().keepOpen();
  schedEnv.setHorizon(horizon);

  /* CREATE THE ACTIVITIES. */
  IloActivity activities[NumberOfActivities];
  IloActivity masonry(env,   dur[0], "masonry   ");
  IloActivity carpentry(env, dur[1], "carpentry ");
  IloActivity plumbing(env,  dur[2], "plumbing  ");
  IloActivity ceiling(env,   dur[3], "ceiling   ");
  IloActivity roofing(env,   dur[4], "roofing   ");
  IloActivity painting(env,  dur[5], "painting  ");
  IloActivity windows(env,   dur[6], "windows   ");
  IloActivity facade(env,    dur[7], "facade    ");
  IloActivity garden(env,    dur[8], "garden    ");
  IloActivity moving(env,    dur[9], "moving    ");

  /* STORE THE ACTIVITIES */
  activities[0] = masonry;
  activities[1] = carpentry;
  activities[2] = plumbing;
  activities[3] = ceiling;
  activities[4] = roofing;
  activities[5] = painting;
  activities[6] = windows;
  activities[7] = facade;
  activities[8] = garden;
  activities[9] = moving;

  /* ADD THE TEMPORAL CONSTRAINTS. */
  model.add(carpentry.startsAfterEnd(masonry));
  model.add(plumbing.startsAfterEnd(masonry));
  model.add(ceiling.startsAfterEnd(masonry));
  model.add(roofing.startsAfterEnd(carpentry));
  model.add(painting.startsAfterEnd(ceiling));
  model.add(windows.startsAfterEnd(roofing));
  model.add(facade.startsAfterEnd(roofing));
  model.add(facade.startsAfterEnd(plumbing));
  model.add(garden.startsAfterEnd(roofing));
  model.add(garden.startsAfterEnd(plumbing));
  model.add(moving.startsAfterEnd(windows));
  model.add(moving.startsAfterEnd(facade));
```

```
  model.add(moving.startsAfterEnd(garden));
  model.add(moving.startsAfterEnd(painting));

  model.add( moving.endsBefore(MaxMakespan) );

  /* CREATE THE WORKER RESOURCE. */
  IloUnaryResource worker(env);
#ifndef NO_CALENDAR
  worker.ignoreCalendarConstraints();
#endif // NO_CALENDAR
  /* MODEL WORKER'S EFFICIENCY */
  IloNum day;
  IloGranularFunction efficiency(env, 0, horizon, 100);
  for(day=0; day+7<=horizon; day+=7) {
    efficiency.setValue(day+2, day+3, 50);  // part-time wednesday
    efficiency.setValue(day+5, day+6, 50);  // saturdays
    efficiency.setValue(day+6, day+7, 0);   // sundays
  }
  efficiency.setValue( 8, 9, 0);  // Tuesday of week #2 is a day off
  efficiency.setValue(14,15, 0);  // Monday of week #3 is a day off


  model.add( IloResourceIntegralConstraint(worker,
                                           IloProcessingTimeVariable,
                                           efficiency));

  /* CREATE COST CURVE */
  IloNum dailyCost = 0;
  IloNum overallCost = 0;
  IloGranularFunction costFunction(env, 0, horizon);
  for(day=0; day<horizon; day+=1) {
    costFunction.setValue(day, day+1, overallCost);
    if (day<3) dailyCost = 1000.0;
    else if (day<7) dailyCost = 900.0;
    else dailyCost = 700.0;
    overallCost += dailyCost;
  }

  model.add( IloResourceFunctionalConstraint(worker,
                                             IloExternalVariable,
                                             costFunction,
                                             IloDurationVariable) );

  costs = IloNumVarArray(env);
  for(IloInt i=0; i<NumberOfActivities; ++i)
  {
    /* SET ACTIVITY AS BREAKABLE */
    activities[i].setBreakable(IloTrue);

    /* ADD THE WORKER RESOURCE CONSTRAINTS. */
    model.add( activities[i].requires(worker) );

    /* CREATE THE INDIVIDUAL COST VARIABLE */
    IloNumVar cost(env, 1, maxBudget, ILOINT);
    costs.add( cost );
    activities[i].setExternalVariable( cost );
    solution.getSolution().add(cost);
```

```
    /* REGISTER ACTIVITIES TO BE STORED IN THE SOLUTION */
    solution.add( activities[i] );
  }

  /* SET THE OBJECTIVE: MINIMIZE THE TOTAL COST VARIABLE */
  totalCost = IloNumVar(env, 0, maxBudget, ILOINT);
  model.add( totalCost == IloSum(costs) );
  model.add( IloMinimize(env, totalCost) );

  /* REGISTER VARIABLES TO BE STORED IN THE SOLUTION */
  solution.getSolution().add(totalCost);

  return model;
}

////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////////////////

void PrintSolution(const IloSolver& solver,
                   const IloSchedulerSolution solution,
                   const IloNumVar totalCost,
                   IloNum MaxMakespan)
{
  solver.out() << "Solution with cost "
               << solution.getSolution().getMin(totalCost)
               << " and maximum makespan " << MaxMakespan
               << endl << endl;

  for (IloSchedulerSolution::ActivityIterator iter(solution);
       iter.ok(); ++iter)
  {
    IloActivity act = *iter;
    solver.out() << act.getName();
    solver.out() << " [" << solution.getStartMin(act);
    solver.out() << " -- " << solution.getProcessingTimeMin(act);
    solver.out() << " (d=" << solution.getDurationMin(act);
    solver.out() << ") --> " << solution.getEndMin(act);
    solver.out() << " ] cost:" << solution.getExternalVariableMin(act);
    solver.out() << endl;
  }
  solver.out() << endl;
}

////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////

int main()
{
  try {


    IloEnv env;
```

```
    IloNumVar totalCost;
    IloNumVarArray costs;
    IloSchedulerSolution solution(env);

    IloModel model = DefineModel(env, maxMakespan, totalCost, costs, solution);

    IloSolver solver(model);
    IloGoal goal = IloGenerate(env, costs) &&
                   IloSetTimesForward(env);

    if (solver.solve(goal)) {
      solution.store( IlcScheduler(solver) );
      PrintSolution(solver, solution, totalCost, maxMakespan);
    }
    else
      solver.out() << “No Solution.” << endl;

    solution.end();
    env.end();
  } catch (IloException& exc) {
    cout << exc << endl;
  }
  return 0;
}

////////////////////////////////////////////////////////////////////////////
//
// RESULTS
//
////////////////////////////////////////////////////////////////////////////

/*
Solution with cost 34600 and maximum makespan 50

masonry    [0 -- 7 (d=11) --> 11 ] cost:9400
carpentry  [21 -- 3 (d=4) --> 25 ] cost:3900
plumbing   [30 -- 8 (d=11) --> 41 ] cost:9400
ceiling    [15 -- 3 (d=4) --> 19 ] cost:3900
roofing    [25 -- 1 (d=1) --> 26 ] cost:1000
painting   [28 -- 2 (d=2) --> 30 ] cost:2000
windows    [43 -- 1 (d=1) --> 44 ] cost:1000
facade     [45 -- 2 (d=2) --> 47 ] cost:2000
garden     [42 -- 1 (d=1) --> 43 ] cost:1000
moving     [49 -- 1 (d=1) --> 50 ] cost:1000
*/
```

Figure 7.1 displays the solution to our problem.

One sees that there is not much slack to finish the house earlier with the selected sequence.

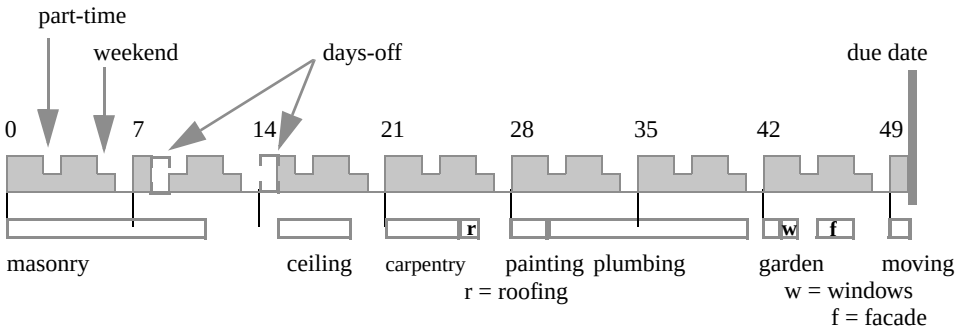


Figure 7.1 *Solution for Example 7*

Having a due date of 50 days has an impact on the total cost. The long activities (masonry and plumbing) are best not executed across weekends or days-off. However, with a due date of 50 days, there is not enough time to delay these long activities until more convenient time-slots are available. Increasing the due date leaves more slack time to avoid processing long activities during days off, thus reducing the total cost. Figure 7.2 shows the optimal cost found for various due dates ranging from 44 to 70 days. The figure shows that it is possible to finish in 47 days at the same cost of finishing in 50 days. With a due date tighter than 47 days, the cost increases fast as some activities are executed over weekends. Finishing before 44 days is impossible.

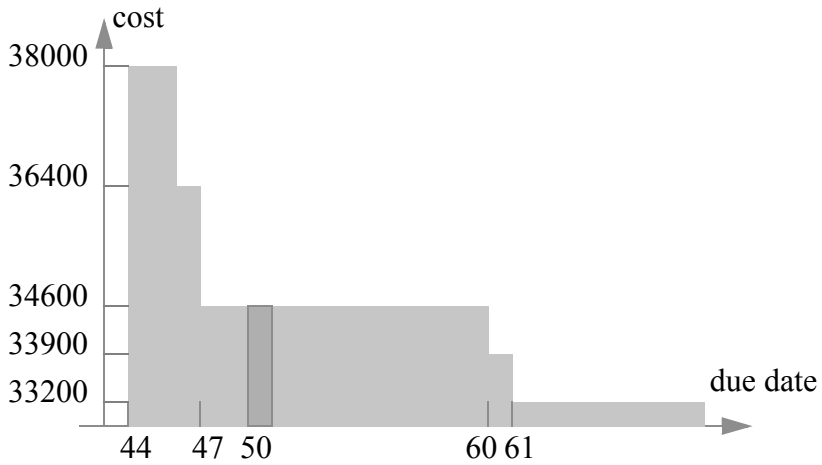


Figure 7.2 *Minimal cost as a function of the imposed due date*

C H A P T E R

8

Adding Alternative Resources

An *alternative resource set* contains two or more equivalent resources to which activities can be assigned. An alternative resource set, for example, may consist of a group of workers or a number of machines that can perform the same function or task. Scheduler provides the class `IloAltResSet` to represent these resources.

The example in this chapter shows how to model a problem that includes alternative resources. Some of the activities in this problem require a specific resource. Other activities require one of the resources in the alternative resource set, but do not specify which of those resources they require.

Describe the Problem—Example 8

In this example we have again four workers who are required to build five houses. However, instead of one worker who performs `masonry` and another who performs `plumbing`, we have two workers who are equally capable of performing the `masonry` and `plumbing`, and either of which can be selected to perform the activity.

The five houses have the three designs, and the corresponding activity durations, shown in Table 5.1 in Adding Transition Times. There are time constraints on the third and fourth houses that state that work on those houses cannot be started before day 5 and must be finished by day 57.

Once again, our objective is to minimize the makespan of the problem.

Define the Problem, Design a Model

There are four workers in the problem who must be defined as unary resources, since each worker can only perform one activity at a time. In addition, we must define an alternative resource set and assign our two interchangeable workers to the set.

As in Example 5 in Adding Transition Times, we must define a function for building the different houses. However, our search must be modified to include a process for choosing which of the alternative resources to use for each activity requiring one of the alternative resources.

Calculate the Horizon

As in Adding Breaks, it is difficult to calculate a horizon for this problem.

However, we can calculate the maximum possible time requirement for the worker who performs the carpentry, ceiling, roofing, facade, and windows tasks (including a maximum delayed start of 15 days) to get a potential horizon of 85. We set the horizon at 91 to allow for activities, like moving, that must be performed after he completes his work.

```
/* CREATE THE MAKESPAN VARIABLE */
IloNum horizon = 91;
makespan = IloNumVar(env, 0, horizon, IloNumVar::Int);
```

Define a Function for Building the Houses

The function `MakeHouse` describes the processing of building a house. A parameter specifies that workers from the alternative resource set can be used to build the house.

```
void MakeHouse(IloModel model,
               const IloNum* dur,
               const IloNum startMin,
               const IloNum endMax,
               const IloUnaryResourceArray workers,
               const IloAltResSet altres,
               const IloNumVar makespan)
```

The following lines of code define the five houses to be built. The earliest start and latest end dates for the first, second, and fifth houses are the origin and horizon of the schedule, but the third and fourth houses have an earliest start date of 5 and a latest end date of 57.

```
/* CREATE THE ACTIVITIES AND CONSTRAINTS FOR THE HOUSES. */
MakeHouse(model, durationsDesign1, 0, horizon, workers, altres, makespan);
MakeHouse(model, durationsDesign1, 0, horizon, workers, altres, makespan);
MakeHouse(model, durationsDesign2, 5, 57, workers, altres, makespan);
MakeHouse(model, durationsDesign2, 5, 57, workers, altres, makespan);
MakeHouse(model, durationsDesign3, 0, horizon, workers, altres, makespan);
```

Create the Resources

As in Example 6, we create the workers as an array of instances of `IloUnaryResource`. In this example we assign names to the workers, to make it easier to understand the solution.

```
const char* WorkerNames [] = {
  “joe”,
  “jim1”,
  “jim2”,
  “jack”};

  /* CREATE THE RESOURCES. */
  IloInt nbOfWorkers = 4;
  IloUnaryResourceArray workers(env, nbOfWorkers);
  for (IloInt k = 0; k < nbOfWorkers; k++)
    workers[k] = IloUnaryResource(env, WorkerNames[k]);
```

Create the Alternative Resource Set

To represent our two workers with equivalent abilities we use an instance of `IloAltResSet`. When an activity requires an instance of this class, the activity requires exactly one of the resources represented in that set.

The following code creates an alternative resource set in the environment `env`. The resource set contains the two workers, indexed as 1 and 2 in the unary resource array.

```
IloAltResSet altres(env, 2, workers[1], workers[2]);
```

Add the Resource Constraints

An activity may only require one single resource from a set of alternative resources during its execution, without specifying which one. We declare the resource constraints in the same manner as in previous examples, except that the activities `masonry` and `plumbing` require one of the workers in our alternative resource set (`workers[1]` or `workers[2]`).

```
/* POST THE RESOURCE CONSTRAINTS. */
```

```
model.add(carpentry.requires(workers[0]));
model.add(ceiling.requires(workers[0]));
model.add(roofing.requires(workers[0]));
model.add(windows.requires(workers[0]));
model.add(facade.requires(workers[0]));

model.add(masonry.requires(workers[1]));

model.add(plumbing.requires(workers[2]));

model.add(garden.requires(workers[3]));
model.add(painting.requires(workers[3]));
model.add(moving.requires(workers[3]));
```

Solve the Problem

A two-step approach is required to solve this problem, so we use && to link the two goals. In the first step, the goal `IloAssignAlternative` assigns a resource to all activities. In the second step, the goal `IloRankForward` ranks all resource constraints while minimizing `makespan`.

```
/* SET THE OBJECTIVE */
model.add(IloMinimize(env, makespan));
/* ... */
  IloEnv env;

  IloNumVar makespan;
  IloModel model = DefineModel(env, makespan);

  IloSolver solver(model);

  IloGoal goal = IloAssignAlternative(env) && IloRankForward(env, makespan);
```

Complete Program and Output—Example 8

You can see the entire program `gsAlt.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

#if defined(ILO_SDXLOUTPUT)
#include “sdxloutput.h”
#endif

////////////////////////////////////////////////////////////////////////////
//
```

```
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////
const char* WorkerNames [] = {
  “joe”,
  “jim1”,
  “jim2”,
  “jack”};
IloNum durationsDesign1 [] = { 7, 3,  8, 3, 1, 2, 1, 2, 1, 1};
IloNum durationsDesign2 [] = {12, 5, 10, 5, 2, 5, 2, 3, 2, 1};
IloNum durationsDesign3 [] = {15, 3, 10, 6, 2, 3, 2, 3, 2, 1};
void MakeHouse(IloModel model,
               const IloNum* dur,
               const IloNum startMin,
               const IloNum endMax,
               const IloUnaryResourceArray workers,
               const IloAltResSet altres,
               const IloNumVar makespan)
{
  IloEnv env = model.getEnv();

  /* CREATE THE ACTIVITIES. */
  IloActivity masonry(env,   dur[0], “masonry   “);
  IloActivity carpentry(env, dur[1], “carpentry “);
  IloActivity plumbing(env,  dur[2], “plumbing  “);
  IloActivity ceiling(env,   dur[3], “ceiling   “);
  IloActivity roofing(env,   dur[4], “roofing   “);
  IloActivity painting(env,  dur[5], “painting  “);
  IloActivity windows(env,   dur[6], “windows   “);
  IloActivity facade(env,    dur[7], “facade    “);
  IloActivity garden(env,    dur[8], “garden    “);
  IloActivity moving(env,    dur[9], “moving    “);

  /* SET STARTMIN AND ENDMAX. */
  model.add(masonry.startsAfter(startMin));
  model.add(moving.endsBefore(endMax));

  /* POST THE TEMPORAL CONSTRAINTS. */
  model.add(carpentry.startsAfterEnd(masonry));
  model.add(plumbing.startsAfterEnd(masonry));
  model.add(ceiling.startsAfterEnd(masonry));
  model.add(roofing.startsAfterEnd(carpentry));
  model.add(painting.startsAfterEnd(ceiling));
  model.add(windows.startsAfterEnd(roofing));
  model.add(facade.startsAfterEnd(roofing));
  model.add(facade.startsAfterEnd(plumbing));
  model.add(garden.startsAfterEnd(roofing));
  model.add(garden.startsAfterEnd(plumbing));
  model.add(moving.startsAfterEnd(windows));
  model.add(moving.startsAfterEnd(facade));
  model.add(moving.startsAfterEnd(garden));
  model.add(moving.startsAfterEnd(painting));

  model.add(moving.endsBefore(makespan));

  /* POST THE RESOURCE CONSTRAINTS. */
  model.add(carpentry.requires(workers[0]));
  model.add(ceiling.requires(workers[0]));
```

```
  model.add(roofing.requires(workers[0]));
  model.add(windows.requires(workers[0]));
  model.add(facade.requires(workers[0]));

  model.add(masonry.requires(altres));
  model.add(plumbing.requires(altres));

  model.add(garden.requires(workers[3]));
  model.add(painting.requires(workers[3]));
  model.add(moving.requires(workers[3]));
}

IloModel DefineModel(const IloEnv env, IloNumVar& makespan)
{
  IloModel model(env);

  /* CREATE THE MAKESPAN VARIABLE */
  IloNum horizon = 91;
  makespan = IloNumVar(env, 0, horizon, IloNumVar::Int);
  /* SET UP THE SHARED ResourceParam */
  IloSchedulerEnv schedEnv(env);
  schedEnv.getResourceParam().setCapacityEnforcement(IloMediumHigh);

  /* CREATE THE RESOURCES. */
  IloInt nbOfWorkers = 4;
  IloUnaryResourceArray workers(env, nbOfWorkers);
  for (IloInt k = 0; k < nbOfWorkers; k++)
    workers[k] = IloUnaryResource(env, WorkerNames[k]);
  IloAltResSet altres(env, 2, workers[1], workers[2]);

  /* CREATE THE ACTIVITIES AND CONSTRAINTS FOR THE HOUSES. */
  MakeHouse(model, durationsDesign1, 0, horizon, workers, altres, makespan);
  MakeHouse(model, durationsDesign1, 0, horizon, workers, altres, makespan);
  MakeHouse(model, durationsDesign2, 5, 57, workers, altres, makespan);
  MakeHouse(model, durationsDesign2, 5, 57, workers, altres, makespan);
  MakeHouse(model, durationsDesign3, 0, horizon, workers, altres, makespan);
  /* SET THE OBJECTIVE */
  model.add(IloMinimize(env, makespan));
  /* RETURN THE CREATED MODEL. */
  return model;
}

////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////////////////

void PrintSolution(const IloSolver solver, const IloNumVar makespan)
{
  IlcScheduler scheduler(solver);
  IloEnv env = solver.getEnv();
  env.out() << "Solution with makespan "
            << solver.getIntVar(makespan).getMin() << endl;

  for(IloIterator<IloResourceConstraint> rctIter(env);
      rctIter.ok(); ++rctIter) {
    IloResourceConstraint rct = *rctIter;
```

```
    if (scheduler.hasAlternative(rct)) {
      IlcAltResConstraint ilcAltRct = scheduler.getAltResConstraint(rct);
      env.out() << ilcAltRct.getActivity() << "\tuses "
                                          << ilcAltRct.getSelected() << endl;
    }
    else {
      IlcResourceConstraint ilcRct = scheduler.getResourceConstraint(rct);
      env.out() << ilcRct.getActivity() << "\tuses "
                                             << ilcRct.getResource() << endl;
    }
  }

  solver.printInformation();
}

////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////

int main()
{
  try {
    IloEnv env;

    IloNumVar makespan;
    IloModel model = DefineModel(env, makespan);

    IloSolver solver(model);
    IloGoal goal = IloAssignAlternative(env) && IloRankForward(env, makespan);

    if (solver.solve(goal)) {
      PrintSolution(solver,makespan);
#if defined(ILO_SDXLOUTPUT)
      IloSDXLOutput output(env);
      ofstream outFile("gsAlt.xml");
      output.write(IlcScheduler(solver), outFile);
      outFile.close();
#endif
    }
    else
      solver.out() << "No Solution" << endl;

    env.end();

  } catch (IloException& exc) {
    cout << exc << endl;
  }
  return 0;
}


////////////////////////////////////////////////////////////////////////////
//
// RESULTS
//
////////////////////////////////////////////////////////////////////////////
```

```
/*
   Solution with makespan 78
   carpentry  [10 -- 3 --> 13]        uses joe [1]
   ceiling    [16 -- 3 --> 19]        uses joe [1]
   roofing    [68 -- 1 --> 69]        uses joe [1]
   windows    [76 -- 1 --> 77]        uses joe [1]
   facade     [71 -- 2 --> 73]        uses joe [1]
   garden     [69..73 -- 1 --> 70..74]        uses jack [1]
   painting   [19..39 -- 2 --> 21..41]        uses jack [1]
   moving     [77 -- 1 --> 78]        uses jack [1]
   carpentry  [7 -- 3 --> 10]        uses joe [1]
   ceiling    [13 -- 3 --> 16]        uses joe [1]
   roofing    [53 -- 1 --> 54]        uses joe [1]
   windows    [75 -- 1 --> 76]        uses joe [1]
   facade     [73 -- 2 --> 75]        uses joe [1]
   garden     [72..75 -- 1 --> 73..76]        uses jack [1]
   painting   [16..37 -- 2 --> 18..39]        uses jack [1]
   moving     [76 -- 1 --> 77]        uses jack [1]
   carpentry  [34 -- 5 --> 39]        uses joe [1]
   ceiling    [29 -- 5 --> 34]        uses joe [1]
   roofing    [41 -- 2 --> 43]        uses joe [1]
   windows    [51 -- 2 --> 53]        uses joe [1]
   facade     [46 -- 3 --> 49]        uses joe [1]
   garden     [43..53 -- 2 --> 45..55]        uses jack [1]
   painting   [34..46 -- 5 --> 39..51]        uses jack [1]
   moving     [53..56 -- 1 --> 54..57]        uses jack [1]
   carpentry  [24 -- 5 --> 29]        uses joe [1]
   ceiling    [19 -- 5 --> 24]        uses joe [1]
   roofing    [39 -- 2 --> 41]        uses joe [1]
   windows    [49 -- 2 --> 51]        uses joe [1]
   facade     [43 -- 3 --> 46]        uses joe [1]
   garden     [41..51 -- 2 --> 43..53]        uses jack [1]
   painting   [24..41 -- 5 --> 29..46]        uses jack [1]
   moving     [51..55 -- 1 --> 52..56]        uses jack [1]
   carpentry  [60 -- 3 --> 63]        uses joe [1]
   ceiling    [54 -- 6 --> 60]        uses joe [1]
   roofing    [63 -- 2 --> 65]        uses joe [1]
   windows    [69 -- 2 --> 71]        uses joe [1]
   facade     [65 -- 3 --> 68]        uses joe [1]
   garden     [65..71 -- 2 --> 67..73]        uses jack [1]
   painting   [60..68 -- 3 --> 63..71]        uses jack [1]
   moving     [71..74 -- 1 --> 72..75]        uses jack [1]
   masonry    [0 -- 7 --> 7]        uses jim2 [1]
   plumbing   [19..63 -- 8 --> 27..71]        uses jim2 [1]
   masonry    [0 -- 7 --> 7]        uses jim1 [1]
   plumbing   [64..65 -- 8 --> 72..73]        uses jim1 [1]
   masonry    [7 -- 12 --> 19]        uses jim2 [1]
   plumbing   [19 -- 10 --> 29]        uses jim1 [1]
   masonry    [7 -- 12 --> 19]        uses jim1 [1]
   plumbing   [29 -- 10 --> 39]        uses jim1 [1]
   masonry    [39 -- 15 --> 54]        uses jim1 [1]
   plumbing   [54..55 -- 10 --> 64..65]        uses jim1 [1]
*/
```

The start and end times of some activities are fixed, while others have variable start and end dates. The total time to complete the project is 78 days.

The results can be interpreted as follows:

`ceiling [54 -- 6 --> 60] uses joe [1]` This ceiling activity has a start time of 54, a duration of 6, and a completion time of 60. It is performed by `joe` who has a capacity of 1.

`plumbing [54..55 -- 10 --> 64..65] uses jim1 [1]` This plumbing activity has an earliest start time of 54, a latest start time of 55, a processing time of 10, an earliest completion time of 64, and a latest completion time of 65. It is performed by `jim1` who has a capacity of 1.

Figure 8.1 provides a graphic display of the solution to our problem. Activities with variable start and end times are positioned at their earliest start times.

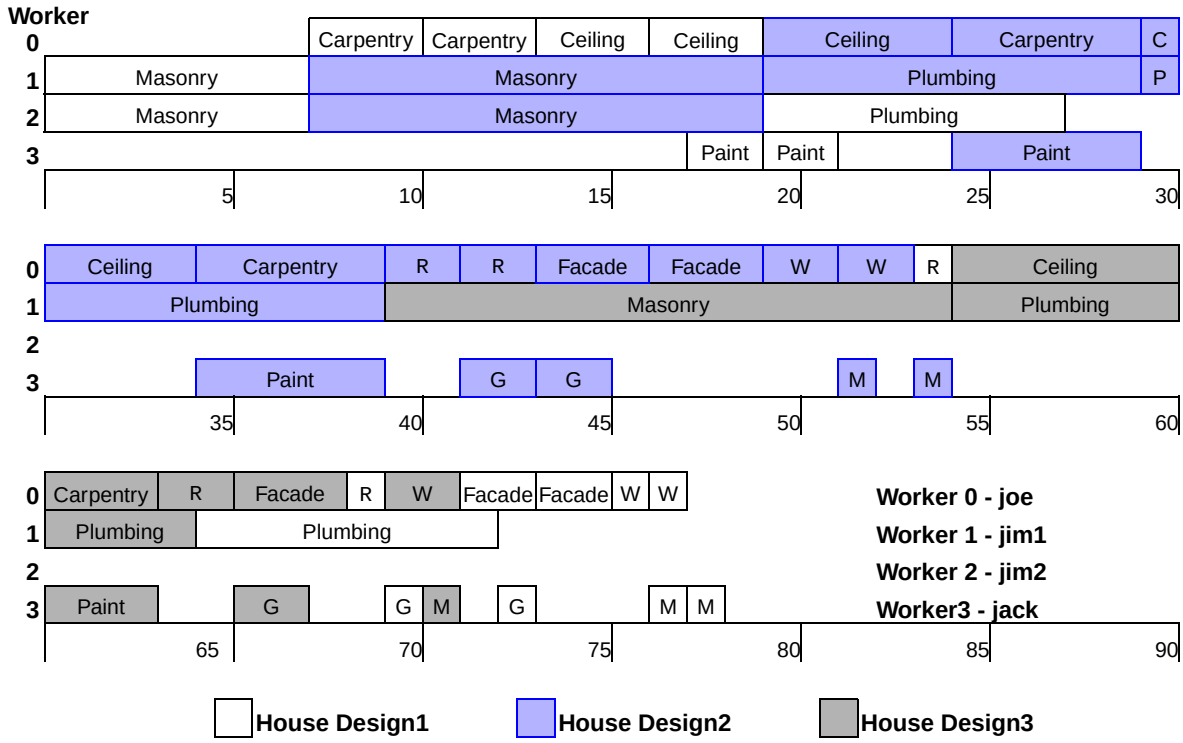


Figure 8.1 *Building Houses with Alternative Resources*

C H A P T E R

9

Using Reservoirs

A *reservoir* is a resource which can be dynamically replenished by producing activities. The class `IloReservoir` is used to model situations in which certain types of activities consume capacity from the reservoir, while others produce capacity. A reservoir must not overflow its maximal capacity nor underflow its minimal capacity.

The example in this chapter shows how to model a problem that includes activities that both produce and consume a reservoir.

A *continuous reservoir* is a reservoir that can be filled or emptied in a continuos and linear process between the start and the end of the activity. For more information about continuous reservoirs, see the class `IloContinuousReservoir` in the *IBM ILOG Scheduler Reference Manual*.

Describe the Problem—Example 9

In this example we again have four workers who are required to build five houses. These workers are specialized just like those in Example 6.

The five houses have the three designs, and the corresponding activity durations, shown in Table 5.1 in Adding Transition Times. However, in this problem, each activity costs $1,000 per day. Each activity also produces a profit of $1,000 when it is completed.

Once again, our objective is to minimize the makespan of the problem.

Define the Problem, Design a Model

The activities, temporal constraints, worker resources, and worker resource constraints are created as in Adding Breaks. Each activity in the problem consumes $1,000 per day. If it has a duration of t, it consumes t*1000 at start time. Each activity also produces a profit of $1,000, so it produces (t+1)*1000 upon completion. We use an instance of `IloReservoir` to model the budget resource.

Create the Budget Resource

The following code defines a reservoir with a maximum capacity of 100 and an initial level (content) of 11 (the unit of budget resource corresponds to $1,000).

```
/* CREATE THE BUDGET RESOURCE. */
IloReservoir budget(env, 100, 11);
```

Add the Budget Resource Constraints

The method `consumes(resource, capacity)` constrains an activity to consume `capacity` of `resource`. The consumed capacity, which can be a non-negative number or an `IloNumVar`, is non-recoverable. Zero capacity is allowed, in which case the resource constraint does not change the availability of the resource. To model the activity cost, we add the `consumes` constraint to each activity and set the capacity consumed to be equal to the activity duration.

The method `produces(reservoir, capacity)` constrains an activity to produce `capacity` of `reservoir`. The produced capacity can be a non-negative number or an `IloNumVar`. Zero capacity is allowed, in which case the resource constraint does not change the availability of the resource. To model the activity profit, we add the `produces` constraint to each activity and set the capacity consumed to be equal to the activity duration plus 1.

```
/* POST THE RESOURCE CONSTRAINTS ON THE BUDGET. */
model.add(masonry.consumes(budget, dur[0]));
model.add(carpentry.consumes(budget, dur[1]));
model.add(plumbing.consumes(budget, dur[2]));
model.add(ceiling.consumes(budget, dur[3]));
model.add(roofing.consumes(budget, dur[4]));
model.add(painting.consumes(budget, dur[5]));
model.add(windows.consumes(budget, dur[6]));
model.add(facade.consumes(budget, dur[7]));
model.add(garden.consumes(budget, dur[8]));
model.add(moving.consumes(budget, dur[9]));

model.add(masonry.produces(budget, (dur[0] + 1)));
model.add(carpentry.produces(budget, (dur[1] + 1)));
model.add(plumbing.produces(budget, (dur[2] + 1)));
model.add(ceiling.produces(budget, (dur[3] + 1)));
model.add(roofing.produces(budget, (dur[4] + 1)));
```

```
model.add(painting.produces(budget, (dur[5] + 1)));
model.add(windows.produces(budget, (dur[6] + 1)));
model.add(facade.produces(budget, (dur[7] + 1)));
model.add(garden.produces(budget, (dur[8] + 1)));
model.add(moving.produces(budget, (dur[9] + 1)));
```

Define a Function for Building the Houses

The function `MakeHouse` describes the processing of building a house. It includes a parameter for the budget resource.

```
void MakeHouse(IloModel model,
               const IloNum* dur,
               const IloNum startMin,
               const IloNum endMax,
               const IloUnaryResourceArray workers,
               const IloReservoir budget,
               const IloNumVar makespan)
```

The following lines of code define the five houses to be built. The earliest start and latest end dates are the origin and horizon of the schedule.

```
/* CREATE THE ACTIVITIES AND CONSTRAINTS FOR THE HOUSES. */
MakeHouse(model, durationsDesign1, 0, horizon, workers, budget, makespan);
MakeHouse(model, durationsDesign1, 0, horizon, workers, budget, makespan);
MakeHouse(model, durationsDesign2, 0, horizon, workers, budget, makespan);
MakeHouse(model, durationsDesign2, 0, horizon, workers, budget, makespan);
MakeHouse(model, durationsDesign3, 0, horizon, workers, budget, makespan);
```

Solve the Problem

To generate an optimal solution, and minimize `makespan`, we add the goal `IloSetTimesForward` to the model, set makespan as the objective variable with a call to `IloMinimize`, and search for a solution.

To select the activities, the predefined activity selector `IloSelFirstActMinEndMax` is used. This selector considers the earliest start and end times among the activities that can still be selected, and from the selectable activities having the minimal earliest start time, it chooses one having the maximal earliest end time

```
/* SET THE OBJECTIVE */
model.add(IloMinimize(env, makespan));
/* ... */
  IloSolver solver(model);

  IloGoal goal = IloSetTimesForward(env, makespan);
  if (solver.solve(goal)) {
    PrintSolution(solver,makespan);
```

```
#if defined(ILO_SDXLOUTPUT)
      IloSDXLOutput output(env);
      ofstream outFile("gsReserv.xml");
      output.write(IlcScheduler(solver), outFile, solver.getIntVar(makespan));
      outFile.close();
#endif
    }
    else
      solver.out() << "No Solution" << endl;
```

Complete Program and Output—Example 9

You can see the entire program `gsReserv.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

#if defined(ILO_SDXLOUTPUT)
#include "sdxloutput.h"
#endif

////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////

/* THREE DESIGNS OF HOUSES EACH HAVING DIFFERENT TASK DURATIONS */

IloNum durationsDesign1 [] = { 7, 3,  8, 3, 1, 2, 1, 2, 1, 1};
IloNum durationsDesign2 [] = {12, 5, 10, 5, 2, 5, 2, 3, 2, 1};
IloNum durationsDesign3 [] = {15, 3, 10, 6, 2, 3, 2, 3, 2, 1};
void MakeHouse(IloModel model,
               const IloNum* dur,
               const IloNum startMin,
               const IloNum endMax,
               const IloUnaryResourceArray workers,
               const IloReservoir budget,
               const IloNumVar makespan)
{
  IloEnv env = model.getEnv();

  /* CREATE THE ACTIVITIES. */
  IloActivity masonry(env,   dur[0], "masonry   ");
  IloActivity carpentry(env, dur[1], "carpentry ");
  IloActivity plumbing(env,  dur[2], "plumbing  ");
  IloActivity ceiling(env,   dur[3], "ceiling   ");
  IloActivity roofing(env,   dur[4], "roofing   ");
  IloActivity painting(env,  dur[5], "painting  ");
  IloActivity windows(env,   dur[6], "windows   ");
  IloActivity facade(env,    dur[7], "facade    ");
  IloActivity garden(env,    dur[8], "garden    ");
```

```
IloActivity moving(env,    dur[9], "moving    ");

/* SET STARTMIN AND ENDMAX. */
model.add(masonry.startsAfter(startMin));
model.add(moving.endsBefore(endMax));

/* POST THE TEMPORAL CONSTRAINTS. */
model.add(carpentry.startsAfterEnd(masonry));
model.add(plumbing.startsAfterEnd(masonry));
model.add(ceiling.startsAfterEnd(masonry));
model.add(roofing.startsAfterEnd(carpentry));
model.add(painting.startsAfterEnd(ceiling));
model.add(windows.startsAfterEnd(roofing));
model.add(facade.startsAfterEnd(roofing));
model.add(facade.startsAfterEnd(plumbing));
model.add(garden.startsAfterEnd(roofing));
model.add(garden.startsAfterEnd(plumbing));
model.add(moving.startsAfterEnd(windows));
model.add(moving.startsAfterEnd(facade));
model.add(moving.startsAfterEnd(garden));
model.add(moving.startsAfterEnd(painting));

model.add(moving.endsBefore(makespan));

/* POST THE RESOURCE CONSTRAINTS ON THE WORKERS. */
model.add(carpentry.requires(workers[0]));
model.add(ceiling.requires(workers[0]));
model.add(roofing.requires(workers[0]));
model.add(windows.requires(workers[0]));
model.add(facade.requires(workers[0]));

model.add(masonry.requires(workers[1]));

model.add(plumbing.requires(workers[2]));

model.add(garden.requires(workers[3]));
model.add(painting.requires(workers[3]));
model.add(moving.requires(workers[3]));
/* POST THE RESOURCE CONSTRAINTS ON THE BUDGET. */
model.add(masonry.consumes(budget, dur[0]));
model.add(carpentry.consumes(budget, dur[1]));
model.add(plumbing.consumes(budget, dur[2]));
model.add(ceiling.consumes(budget, dur[3]));
model.add(roofing.consumes(budget, dur[4]));
model.add(painting.consumes(budget, dur[5]));
model.add(windows.consumes(budget, dur[6]));
model.add(facade.consumes(budget, dur[7]));
model.add(garden.consumes(budget, dur[8]));
model.add(moving.consumes(budget, dur[9]));

model.add(masonry.produces(budget, (dur[0] + 1)));
model.add(carpentry.produces(budget, (dur[1] + 1)));
model.add(plumbing.produces(budget, (dur[2] + 1)));
model.add(ceiling.produces(budget, (dur[3] + 1)));
model.add(roofing.produces(budget, (dur[4] + 1)));
model.add(painting.produces(budget, (dur[5] + 1)));
model.add(windows.produces(budget, (dur[6] + 1)));
model.add(facade.produces(budget, (dur[7] + 1)));
```

```
  model.add(garden.produces(budget, (dur[8] + 1)));
  model.add(moving.produces(budget, (dur[9] + 1)));
}

IloModel DefineModel(const IloEnv env, IloNumVar& makespan)
{
  IloModel model(env);

  /* CREATE THE MAKESPAN VARIABLE. */
  IloNum horizon = 150;
  makespan = IloNumVar(env, 0, horizon, IloNumVar::Int);

  /* CREATE THE WORKERS. */
  IloInt nrOfWorkers = 4;
  IloUnaryResourceArray workers(env,nrOfWorkers);
  for (IloInt k = 0; k < nrOfWorkers; k++) {
    workers[k] = IloUnaryResource(env);
    workers[k].setCapacityEnforcement(IloMediumHigh);
  }
  /* CREATE THE BUDGET RESOURCE. */
  IloReservoir budget(env, 100, 11);
  /* CREATE THE ACTIVITIES AND CONSTRAINTS FOR THE HOUSES. */
  MakeHouse(model, durationsDesign1, 0, horizon, workers, budget, makespan);
  MakeHouse(model, durationsDesign1, 0, horizon, workers, budget, makespan);
  MakeHouse(model, durationsDesign2, 0, horizon, workers, budget, makespan);
  MakeHouse(model, durationsDesign2, 0, horizon, workers, budget, makespan);
  MakeHouse(model, durationsDesign3, 0, horizon, workers, budget, makespan);
  /* SET THE OBJECTIVE */
  model.add(IloMinimize(env, makespan));
  return model;
}

////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////////////////

void PrintSolution(const IloSolver solver, const IloNumVar makespan)
{
  IlcScheduler scheduler(solver);
  IloEnv env = solver.getEnv();
  env.out() << "Solution with makespan "
            << solver.getIntVar(makespan).getMin() << endl;
  for(IloIterator<IloActivity> act(env);
      act.ok();
      ++act)
    env.out() << scheduler.getActivity(*act) << endl;
  solver.printInformation();
}

////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////

int main()
```

```
{
  try {
    IloEnv env;
    IloNumVar makespan;
    IloModel model = DefineModel(env, makespan);

    IloSolver solver(model);

    IloGoal goal = IloSetTimesForward(env, makespan);
    if (solver.solve(goal)) {
      PrintSolution(solver,makespan);
#if defined(ILO_SDXLOUTPUT)
      IloSDXLOutput output(env);
      ofstream outFile("gsReserv.xml");
      output.write(IlcScheduler(solver), outFile, solver.getIntVar(makespan));
      outFile.close();
#endif
    }
    else
      solver.out() << "No Solution" << endl;

    env.end();

  } catch (IloException& exc) {
    cout << exc << endl;
  }

  return 0;
}

////////////////////////////////////////////////////////////////////////////
//
// RESULTS
//
////////////////////////////////////////////////////////////////////////////

/*
  Solution with makespan 81
  masonry    [7 -- 7 --> 14]
  carpentry  [17 -- 3 --> 20]
  plumbing   [31 -- 8 --> 39]
  ceiling    [14 -- 3 --> 17]
  roofing    [20 -- 1 --> 21]
  painting   [19 -- 2 --> 21]
  windows    [22 -- 1 --> 23]
  facade     [42 -- 2 --> 44]
  garden     [39 -- 1 --> 40]
  moving     [45 -- 1 --> 46]
  masonry    [0 -- 7 --> 7]
  carpentry  [7 -- 3 --> 10]
  plumbing   [21 -- 8 --> 29]
  ceiling    [11 -- 3 --> 14]
  roofing    [10 -- 1 --> 11]
  painting   [17 -- 2 --> 19]
  windows    [21 -- 1 --> 22]
  facade     [31 -- 2 --> 33]
  garden     [29 -- 1 --> 30]
  moving     [38 -- 1 --> 39]
```

```
  masonry    [44 -- 12 --> 56]
  carpentry  [62 -- 5 --> 67]
  plumbing   [59 -- 10 --> 69]
  ceiling    [57 -- 5 --> 62]
  roofing    [67 -- 2 --> 69]
  painting   [62 -- 5 --> 67]
  windows    [78 -- 2 --> 80]
  facade     [75 -- 3 --> 78]
  garden     [69 -- 2 --> 71]
  moving     [80 -- 1 --> 81]
  masonry    [14 -- 12 --> 26]
  carpentry  [33 -- 5 --> 38]
  plumbing   [39 -- 10 --> 49]
  ceiling    [26 -- 5 --> 31]
  roofing    [38 -- 2 --> 40]
  painting   [40 -- 5 --> 45]
  windows    [40 -- 2 --> 42]
  facade     [72 -- 3 --> 75]
  garden     [49 -- 2 --> 51]
  moving     [75 -- 1 --> 76]
  masonry    [29 -- 15 --> 44]
  carpentry  [50 -- 3 --> 53]
  plumbing   [49 -- 10 --> 59]
  ceiling    [44 -- 6 --> 50]
  roofing    [53 -- 2 --> 55]
  painting   [51 -- 3 --> 54]
  windows    [55 -- 2 --> 57]
  facade     [69 -- 3 --> 72]
  garden     [59 -- 2 --> 61]
  moving     [72 -- 1 --> 73]
*/
```

The start and end times of some activities are fixed and the total time to complete the project is 81 days.

Figure 9.1 provides a graphic display of the solution to our problem. The figure displays the worker schedules as three sections of 30 days each. The reservoir content is also displayed (black shading) for the first 60 days. After day 50 the content of the reservoir increases steadily. The final content of the reservoir is 61 (profit of 1 on 50 activities plus the starting level of 11).

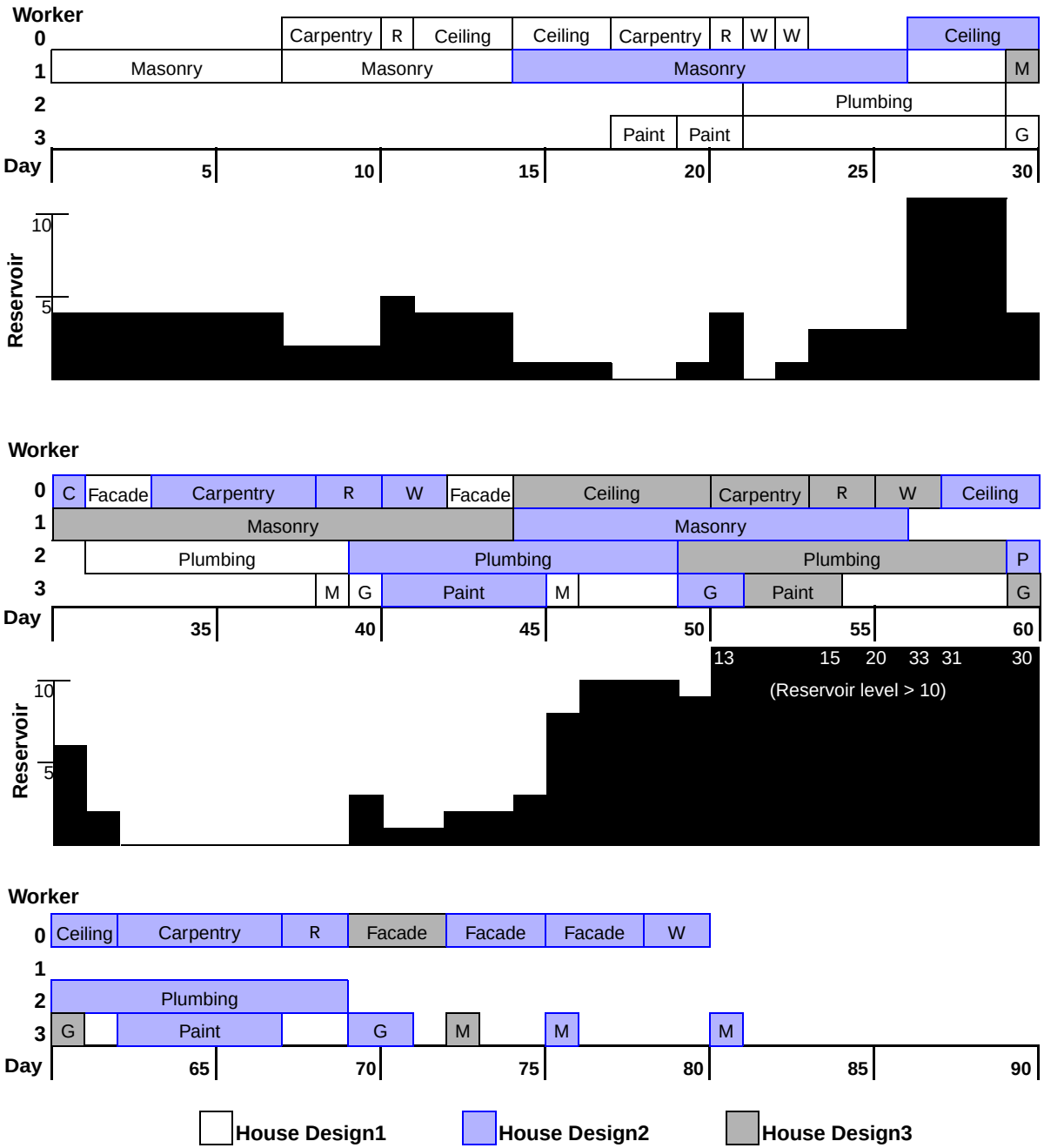


Figure 9.1 *Building Houses with Reservoirs*

C H A P T E R

10

Using the Trace Facilities

This example illustrates how the trace facilities in Scheduler can be used to get printed information about the propagations. We use the class `IlcSchedulerPrintTrace` for this purpose.

Describe the Problem—Example 10

The example is based on the one in Using Reservoirs. Only three houses will be built, one of each type. We will compute a first solution to this problem while displaying some information about the modifications to the activities.

We will use the parameter `traceLevel` to set the type and amount of information returned.

- If `traceLevel` is 0, then we want to trace all the modifications that pertain to the plumbing phase for each house.
- If `traceLevel` is 1, we want to know when the start variable of a set of activities is increased or becomes bound. The set of activities is defined as all those that require the fourth worker (that is, all the activities of type garden, painting or moving).
- If `traceLevel` is 2, we want to have the same type of information, but for all the activities in the problem.

We have to determine which activities to trace. This can be done in various ways, for example, by using a data structure to store the activities or resources that are to be traced. We have to create the trace objects and use filters to collect only the information desired.

Deciding which Objects to Trace

We will describe here a method that relies on information stored in the model itself: for each object that may be traced, we will use the method `setObject(IloAny)` (defined on the class `IloExtractable`) to "mark" the object as being traced.

To trace all the plumbing activities, we use:

```
plumbing.setObject((IloAny)1);
```

This line of code is in the `MakeHouse` function that builds all the activities corresponding to one house. In this way the three plumbing activities are marked.

To trace all the activities that require the fourth worker, we mark this resource the same way, in the function `DefineModel`.

```
workers[3].setObject((IloAny)1);
```

Of course, tracing all the activities in the model does not require any mark.

Creating a Trace Object

To get all the information about the modifications of traced activities, we create an instance of the class `IlcSchedulerPrintTrace` at the beginning of the search. We build a goal like this:

```
ILCGOAL1(SetTraceIlc, IlcInt, traceLevel) {
  IloSolver solver = getSolver();
  IlcScheduler scheduler(solver);
  IlcSchedulerPrintTrace trace(scheduler);
  solver.setTraceMode(IlcTrue);
```

We first create a trace object that is used to specify which objects must be traced, then instruct Solver to turn on its trace mode (otherwise the trace object would do nothing).

Tracing the Specified Objects

We use the following code, depending on the `traceLevel` parameter, to add marked objects to our instance of `IlcSchedulerPrintTrace`.

If `traceLevel` is 0, we iterate over all the activities to find those that are marked. They are added to the trace object with the method `trace(IlcActivity)`.

```
// Trace only some specific activities
for (IlcActivityIterator it(scheduler); it.ok(); ++it) {
  IlcActivity act = *it;
  if (act.getObject() == (IloAny)1) {
    trace.trace(act);
  }
}
```

If `traceLevel` is 1, we iterate over all the resources to find those that are marked. They are added to the trace object with the method `trace(IlcResource)`. Its effect is that all the activities that require the resource will be traced.

```
// We trace only the activities that require a specific resource
for (IlcResourceIterator it(scheduler); it.ok(); ++it) {
  IlcResource res = *it;
  if (res.getObject() == (IloAny)1) {
    trace.trace(res);
  }
}
```

And finally, if `traceLevel` is 2, we want to trace all the activities in the problem. This is achieved simply by using the method `traceAllActivities`.

```
// Tracing all activities in the problem
trace.traceAllActivities();
```

Using filters

The modification of a traced object results in two trace events: one just before the modification and one just after the modification. The class `IlcSchedulerPrintTrace` is a trace object that, by default, prints information about all trace events. Printing all trace events, especially in a large application, may produce too much data to be easily analyzed. To help you cope with this, Scheduler Engine provides trace filters, functions with the following signature:

```
typedef IlcBool (*IlcSchedulerTraceFilter)(IlcBool isBeginEvent,
    IlcSchedulerChange schedChange, IlcSolverChange solverChange);
```

Writing a function of type `IlcSchedulerTraceFilter` allows you to customize the trace information that is printed. After you attach the function to the print trace object (using `setFilter`), the function will be called for each trace event.

The following list describes the parameters of an `IlcSchedulerTraceFilter` function.

1. `isBeginEvent`: This argument will have the value `IlcTrue` if the function is called just before the modification of an object. When called just after the modification of an object, the value is `IlcFalse`.

2. `schedChange`: This argument specifies the type of Scheduler Engine object that is about to be (or just was) changed. For example, the values of the `IlcSchedulerChange` enum `IlcActivityStart`, `IlcPrecedenceConstraintDelay`, and `IlcResourceConstraintCapacity` correspond to events which modify, respectively, the start variable of an activity, the delay variable of a precedence constraint, and the capacity variable of a resource constraint.

3. `solverChange`: This argument, when it is applicable, specifies the type of Solver variable that is modifed and what the modification is. For example, the values of the `IlcSolverChange` enum `IlcIntExpSetMin` and `IlcIntSetVarRemovePossible`, correspond to events which, respectively, increase the minimum value of an integer variable and remove an element from a set variable.

Each time a trace event occurs (that is, just before and just after the modification of a traced object), this function will be called with the information corresponding to the event as argument. The event will be displayed if and only if the function returns `IlcTrue`.

The member functions `IlcSchedulerPrintTrace::setFilter` and `IlcSchedulerPrintTrace::resetFilter` allow you to define or reset such a filter function. If there is no filter definition (the default), all the events will be displayed.

For our example, with `traceLevel` equal to 1 or 2, we only want to print before the modification. If this event concerns an activity, we only want to print modifications that increase the minimum of the start variable or assign that variable. Here is the code of the filter function that does this.

```
IlcBool MyPrintTraceFilter(IlcBool isBeginEvent,
                           IlcSchedulerChange change,
                           IlcSolverChange solverChange) {
  // We are only interested in events that occur just
  // before a modification
  if (!isBeginEvent) {
    return IlcFalse;
  }
  switch (change) {
    // If the start variable of an activity is modified,
    // we only care if it becomes bound, or its min is
    // modified.
    //I.e. we do not care about the Latest Start Time
    case IlcActivityStart:
      return ((solverChange == IlcIntExpSetMin) ||
              (solverChange == IlcIntExpSetValue));
    // All other modifications of activities are skipped
    case IlcActivityEnd:
    case IlcActivityProcessingTime:
    case IlcActivityDuration:
    case IlcActivityDurationOfBreaks:
    case IlcActivityStartOverlap:
```

```
    case IlcActivityEndOverlap:
      return IlcFalse;
    // Other events are traced
    default:
      return IlcTrue;
  }
}
```

Having defined this function, we simply have to call `IlcSchedulerPrintTrace::setFilter` with the address of the function as parameter.

```
    // Setting a filter to get only some specified events
    trace.setFilter(MyPrintTraceFilter);
```

The following code then defines the goal to set the trace.

```
ILCGOAL1(SetTraceIlc, IlcInt, traceLevel) {
  IloSolver solver = getSolver();
  IlcScheduler scheduler(solver);
  IlcSchedulerPrintTrace trace(scheduler);
  solver.setTraceMode(IlcTrue);
  if (traceLevel == 0) {
    // Trace only some specific activities
    for (IlcActivityIterator it(scheduler); it.ok(); ++it) {
      IlcActivity act = *it;
      if (act.getObject() == (IloAny)1) {
        trace.trace(act);
      }
    }
  } else if (traceLevel == 1) {
    // We trace only the activities that require a specific resource
    for (IlcResourceIterator it(scheduler); it.ok(); ++it) {
      IlcResource res = *it;
      if (res.getObject() == (IloAny)1) {
        trace.trace(res);
      }
    }
  } else {
    // Tracing all activities in the problem
    trace.traceAllActivities();
  }
  if (traceLevel > 0) {
    // Setting a filter to get only some specified events
    trace.setFilter(MyPrintTraceFilter);
  }
  return 0;
}

ILOCPGOALWRAPPER1(SetTrace, solver, IloInt, traceLevel) {
  return SetTraceIlc(solver, traceLevel);
}
```

Solve the Problem

We have to ensure that this goal will be the first one executed when solving the problem. We simply write

```
IloGoal goal = SetTrace(env, traceLevel) &&
               IloSetTimesForward(env, makespan);
```

and then pass this goal as an argument to the method `solve(IloGoal)` of the class `IloSolver`.

Depending on the value of `traceLevel`, all the modifications that we are interested in will be printed during the search.

Complete Program and Output—Example 10

You can see the entire program `gsTrace.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

#if defined(ILO_SDXLOUTPUT)
#include “sdxloutput.h”
#endif

////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////

IloActivity MakeActivity(IloEnv env,
                         IloInt houseNumber,
                         IloNum duration,
                         const char* name) {
  char* s = new char[strlen(name)+3];
  sprintf(s, “%ld-%s”, houseNumber, name);
  IloActivity act(env, duration, s);
  delete [] s;
  return act;
}

/* THREE DESIGNS OF HOUSES EACH HAVING DIFFERENT TASK DURATIONS */

IloNum durationsDesign1 [] = { 7, 3,  8, 3, 1, 2, 1, 2, 1, 1};
IloNum durationsDesign2 [] = {12, 5, 10, 5, 2, 5, 2, 3, 2, 1};
IloNum durationsDesign3 [] = {15, 3, 10, 6, 2, 3, 2, 3, 2, 1};
void MakeHouse(IloModel model,
               IloInt houseNumber,
```

```
                const IloNum* dur,
                const IloNum startMin,
                const IloNum endMax,
                const IloUnaryResourceArray workers,
                const IloReservoir budget,
                const IloNumVar makespan)
{
  IloEnv env = model.getEnv();

  /* CREATE THE ACTIVITIES. */
  IloActivity masonry =
    MakeActivity(env, houseNumber, dur[0], "masonry   ");
  IloActivity carpentry =
    MakeActivity(env, houseNumber, dur[1], "carpentry ");
  IloActivity plumbing =
    MakeActivity(env, houseNumber, dur[2], "plumbing  ");

  plumbing.setObject((IloAny)1);

  IloActivity ceiling =
    MakeActivity(env, houseNumber, dur[3], "ceiling   ");
  IloActivity roofing =
    MakeActivity(env, houseNumber, dur[4], "roofing   ");
  IloActivity painting =
    MakeActivity(env, houseNumber, dur[5], "painting  ");
  IloActivity windows =
    MakeActivity(env, houseNumber, dur[6], "windows   ");
  IloActivity facade =
    MakeActivity(env, houseNumber, dur[7], "facade    ");
  IloActivity garden =
    MakeActivity(env, houseNumber, dur[8], "garden    ");
  IloActivity moving =
    MakeActivity(env, houseNumber, dur[9], "moving    ");

  /* SET STARTMIN AND ENDMAX. */
  model.add(masonry.startsAfter(startMin));
  model.add(moving.endsBefore(endMax));

  /* POST THE TEMPORAL CONSTRAINTS. */
  model.add(carpentry.startsAfterEnd(masonry));
  model.add(ceiling.startsAfterEnd(masonry));
  model.add(roofing.startsAfterEnd(carpentry));
  model.add(painting.startsAfterEnd(ceiling));
  model.add(windows.startsAfterEnd(roofing));
  model.add(facade.startsAfterEnd(roofing));
  model.add(facade.startsAfterEnd(plumbing));
  model.add(garden.startsAfterEnd(roofing));
  model.add(garden.startsAfterEnd(plumbing));
  model.add(moving.startsAfterEnd(windows));
  model.add(moving.startsAfterEnd(facade));
  model.add(moving.startsAfterEnd(garden));
  model.add(moving.startsAfterEnd(painting));

  IloPrecedenceConstraint p1(plumbing.startsAfterEnd(masonry));
  p1.setName("");
  model.add(p1);

  model.add(moving.endsBefore(makespan));
```

```
  /* POST THE RESOURCE CONSTRAINTS ON THE WORKERS. */
  model.add(carpentry.requires(workers[0]));
  model.add(ceiling.requires(workers[0]));
  model.add(roofing.requires(workers[0]));
  model.add(windows.requires(workers[0]));
  model.add(facade.requires(workers[0]));

  model.add(masonry.requires(workers[1]));

  model.add(plumbing.requires(workers[2]));

  model.add(garden.requires(workers[3]));
  model.add(painting.requires(workers[3]));
  model.add(moving.requires(workers[3]));

  /* POST THE RESOURCE CONSTRAINTS ON THE BUDGET. */
  model.add(masonry.consumes(budget, dur[0]));
  model.add(carpentry.consumes(budget, dur[1]));
  model.add(plumbing.consumes(budget, dur[2]));
  model.add(ceiling.consumes(budget, dur[3]));
  model.add(roofing.consumes(budget, dur[4]));
  model.add(painting.consumes(budget, dur[5]));
  model.add(windows.consumes(budget, dur[6]));
  model.add(facade.consumes(budget, dur[7]));
  model.add(garden.consumes(budget, dur[8]));
  model.add(moving.consumes(budget, dur[9]));

  model.add(masonry.produces(budget, (dur[0] + 1)));
  model.add(carpentry.produces(budget, (dur[1] + 1)));
  model.add(plumbing.produces(budget, (dur[2] + 1)));
  model.add(ceiling.produces(budget, (dur[3] + 1)));
  model.add(roofing.produces(budget, (dur[4] + 1)));
  model.add(painting.produces(budget, (dur[5] + 1)));
  model.add(windows.produces(budget, (dur[6] + 1)));
  model.add(facade.produces(budget, (dur[7] + 1)));
  model.add(garden.produces(budget, (dur[8] + 1)));
  model.add(moving.produces(budget, (dur[9] + 1)));
}

IloModel DefineModel(const IloEnv env, IloNumVar& makespan)
{
  IloModel model(env);

  /* CREATE THE MAKESPAN VARIABLE. */
  IloNum horizon = 150;
  makespan = IloNumVar(env, 0, horizon, IloNumVar::Int);

  /* CREATE THE WORKERS. */
  IloInt nrOfWorkers = 4;
  IloUnaryResourceArray workers(env,nrOfWorkers);
  for (IloInt k = 0; k < nrOfWorkers; k++) {
    workers[k] = IloUnaryResource(env);
    workers[k].setCapacityEnforcement(IloMediumHigh);
  }

  workers[3].setObject((IloAny)1);
```

```
  /* CREATE THE BUDGET RESOURCE. */
  IloReservoir budget(env, 100, 11);

  /* CREATE THE ACTIVITIES AND CONSTRAINTS FOR THE HOUSES. */
  MakeHouse(model, 0, durationsDesign1,
            0, horizon, workers, budget, makespan);
  MakeHouse(model, 1, durationsDesign2,
            0, horizon, workers, budget, makespan);
  MakeHouse(model, 2, durationsDesign3,
            0, horizon, workers, budget, makespan);

  return model;
}

////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////////////////

void PrintSolution(const IloSolver solver, const IloNumVar makespan)
{
  IlcScheduler scheduler(solver);
  IloEnv env = solver.getEnv();
  env.out() << "Solution with makespan "
            << solver.getIntVar(makespan).getMin() << endl;
  for(IloIterator<IloActivity> act(env);
      act.ok();
      ++act)
    env.out() << scheduler.getActivity(*act) << endl;
  solver.printInformation();
}

////////////////////////////////////////////////////////////////////////////
//
// GOAL TO SET THE TRACE
//
////////////////////////////////////////////////////////////////////////////

IlcBool MyPrintTraceFilter(IlcBool isBeginEvent,
                           IlcSchedulerChange change,
                           IlcSolverChange solverChange) {
  // We are only interested in events that occur just
  // before a modification
  if (!isBeginEvent) {
    return IlcFalse;
  }
  switch (change) {
    // If the start variable of an activity is modified,
    // we only care if it becomes bound, or its min is
    // modified.
    //I.e. we do not care about the Latest Start Time
    case IlcActivityStart:
      return ((solverChange == IlcIntExpSetMin) ||
              (solverChange == IlcIntExpSetValue));
    // All other modifications of activities are skipped
    case IlcActivityEnd:
    case IlcActivityProcessingTime:
```

```
    case IlcActivityDuration:
    case IlcActivityDurationOfBreaks:
    case IlcActivityStartOverlap:
    case IlcActivityEndOverlap:
      return IlcFalse;
    // Other events are traced
    default:
      return IlcTrue;
  }
}


ILCGOAL1(SetTraceIlc, IlcInt, traceLevel) {
  IloSolver solver = getSolver();
  IlcScheduler scheduler(solver);
  IlcSchedulerPrintTrace trace(scheduler);
  solver.setTraceMode(IlcTrue);
  if (traceLevel == 0) {
    // Trace only some specific activities
    for (IlcActivityIterator it(scheduler); it.ok(); ++it) {
      IlcActivity act = *it;
      if (act.getObject() == (IloAny)1) {
        trace.trace(act);
      }
    }
  } else if (traceLevel == 1) {
    // We trace only the activities that require a specific resource
    for (IlcResourceIterator it(scheduler); it.ok(); ++it) {
      IlcResource res = *it;
      if (res.getObject() == (IloAny)1) {
        trace.trace(res);
      }
    }
  } else {
    // Tracing all activities in the problem
    trace.traceAllActivities();
  }
  if (traceLevel > 0) {
    // Setting a filter to get only some specified events
    trace.setFilter(MyPrintTraceFilter);
  }
  return 0;
}

ILOCPGOALWRAPPER1(SetTrace, solver, IloInt, traceLevel) {
  return SetTraceIlc(solver, traceLevel);
}

////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////

int main(int argc, char** argv)
{
  try {
    IloEnv env;
```

```
    IloNumVar makespan;
    IloModel model = DefineModel(env, makespan);

    IloSolver solver(model);

    IloInt traceLevel = 0;
    if (argc == 2) {
      traceLevel = atol(argv[1]);
    }

    IloGoal goal = SetTrace(env, traceLevel) &&
                   IloSetTimesForward(env, makespan);
    if (solver.solve(goal)) {
      PrintSolution(solver,makespan);
#if defined(ILO_SDXLOUTPUT)
      IloSDXLOutput output(env);
      ofstream outFile("gsTrace.xml");
      output.write(IlcScheduler(solver), outFile, solver.getIntVar(makespan));
      outFile.close();
#endif

    }
    else
      solver.out() << "No Solution" << endl;

    env.end();

  } catch (IloException& exc) {
    cout << exc << endl;
  }

  return 0;
}
```

The whole output is not printed here, but with `traceLevel` equal to 0, it will begin with:

```
IlcOr   : 1
 >> begin IlcActivityStart(IlcIntExpSetMin) 2-plumbing
                                  [22..136 -- 10 --> 32..146] val: 25
    goal:IlcGoalScheduleActivity
    the active demon is the constraint: IlcPrecedenceConstraint(0x010A01C8)
                                  [2-plumbing   starts after end 2-masonry   ]
 << end   IlcActivityStart(IlcIntExpSetMin) 2-plumbing
                                  [25..136 -- 10 --> 32..146] val: 25
 >> begin IlcActivityEnd(IlcIntExpSetMin) 2-plumbing
                                  [25..136 -- 10 --> 32..146] val: 35
    goal:IlcGoalScheduleActivity
    propagation of demon, the constraint associated with the demon is:
                                  2-plumbing   [25..136 -- 10 --> 32..146]
 << end   IlcActivityEnd(IlcIntExpSetMin) 2-plumbing
                                  [25..136 -- 10 --> 35..146] val: 35
IlcOr   : 2
 >> begin IlcActivityStart(IlcIntExpSetMin) 2-plumbing   [
                                  25..136 -- 10 --> 35..146] val: 34
    goal:IlcGoalScheduleActivity
    the active demon is the constraint: IlcPrecedenceConstraint(0x010A01C8)
                                  [2-plumbing   starts after end 2-masonry   ]
```

We learn here that during the first choice point, the plumbing activity for the third house (numbered 2) has its earliest start time shifted from 22 to 25 due to a precedence constraint, and then its earliest end time shifted from 32 to 35 due to the internal constraint between all the variables of the activity. The choice made next (second choice point) leads to a shift of this same activity from 25 to 34 as earliest start time.

Part II

Advanced Concepts

The chapters in this part introduce some new Scheduler concepts and functionality and further discuss using Scheduler with IBM© ILOG© Solver and IBM ILOG Concert Technology. Examples using a variety of resources and constraints are presented. The examples in this part, although more complex than those in Part I, are still of interest to most Scheduler users.

C H A P T E R

11

Advanced Features and Concepts

Part I, explored the basic functions and features of Scheduler. Part II, explores advanced features and concepts which provide the flexibility to model a wide variety of scheduling problems. This chapter introduces some of these advanced features, which are then demonstrated in the examples of the following chapters.

Detailed descriptions of the classes and methods mentioned here can be found in the *IBM ILOG Scheduler Reference Manual*.

Parameter Sharing

Parameters represent characteristics or behaviors that can be attached to a scheduling object. For example, a scheduling object like a resource may have a maximal capacity that varies with time, or there may be break intervals during which activities executing on that resource may be suspended. Parameter classes represent these various characteristics, such as capacities, breaks, and transition times.

Concert Technology allows these characteristics to be shared between several scheduling objects. For example, several instances of resources may share the same break list (such as, all these resources have breaks on weekend), or several activities may share the same behavior with respect to breaks on resources. Sharing a characteristic saves memory in the model and allows easy modification of the value with only one function call for a group of

objects. For more information about parameters, see *Parameter Classes* in the *Overview* of the *IBM ILOG Scheduler Reference Manual*.

Advanced Resources

The examples in Part I, relied primarily on unary resources—instances of `IloUnaryResource` with a capacity of one that are either available at a requested time or otherwise occupied. There are more complex resources, which are often defined (or limited) by the *capacity*, *energy*, or the *state* that they provide.

Resources may be subject to different kinds of capacity limitations. For example, in one scheduling problem, we may be concerned primarily with which airplane is assigned to which flight in terms of number of passengers carried. In another context, we may be more concerned with how many hours each plane is in use between scheduled maintenance dates. The standard resource class in Scheduler is `IloDiscreteResource`, and it provides the ability to create capacity resources that are more complicated than unary resources.

One type of frequently encountered capacity limitation is the overall amount of *energy* (for example, work-hours) spent by a resource. The class `IloDiscreteEnergy` is available to create such resources. `IloDiscreteResource` and `IloDiscreteEnergy` are discussed further in Resources as Power versus Energy.

Some scheduling problems involve resources that can operate in different *states*. For example, in a bakery, the temperature of the oven must be warm for certain pastries but very hot for others, and these oven states and the time it takes to change states must be taken into account in a schedule. Similarly, in an auto body shop, several car parts may be painted at the same time if they are all of the same color, but not if they differ in color. In this case, the color in use in the paint room is the state of the resource. See State Resources for more information.

The following sections discuss some important considerations for the use of the resource classes provided by Scheduler.

Resources as Power versus Energy

The standard class of resource defined in Scheduler is `IloDiscreteResource`. Instances of that class can deal with resource capacity in two ways. You can specify that the resource capacity cannot exceed a given maximum at any point over a given interval of time; and you can specify that a resource capacity must reach a given minimum at every point over a given interval of time.

Another type of capacity limitation is the overall amount of energy (for example, work-hours) that is spent by a resource (for example, a group of nurses) over a given period (for example, a week). You may wish to limit this energy to a maximum or, in contrast, ensure that it reaches a given minimum. `IloDiscreteEnergy` is used to model energy resources.

Roughly, an instance of the class `IloDiscreteEnergy` does not represent a resource *per se*, but the amount of work performed by the resource over regular intervals of time. The *energy* of an instance of `IloDiscreteEnergy` is not measured as a number of machines or as a number of men and women, but as a number of machine-hours, human-hours, human-weeks, and so forth, spent in the performance of the scheduled activities.

Example

Let's assume we're creating a schedule for a group of `n` workers who might agree to work on week-ends, but who would not agree to work more than five days a week. The following function creates both an instance of `IloDiscreteResource` and an instance of `IloDiscreteEnergy` to represent that situation. The "theoretical" capacity of the `IloDiscreteResource` is defined to be `n`. In the case of the `IloDiscreteEnergy`, the code imposes a limitation on the energy of `(5 * n)` worker-days every week. Incidentally, it assumes that date 0 corresponds to the beginning of the first week.

```
void MakeDiscreteResourceAndEnergy(IloEnv env,
                                   IloInt numberOfWeeks,
                                   IloInt n) {
   IloDiscreteResource resource(env, n);
   IloDiscreteEnergy energy(env, 7, 5*n);
}
```

Sometimes you may encounter difficulties in deciding whether a resource must be represented by an instance of `IloDiscreteResource` or by an instance of `IloDiscreteEnergy`.

There's an easy question to ask yourself in order to make that decision: "If I multiply all dates and activity durations by a given factor, then are the measures of capacity demand multiplied by the same factor?"

If the answer to this question is "yes"—multiplying dates and durations does multiply the capacity by the same factor—then the measure of capacity is a measure of energy, and the resource must be represented by an instance of `IloDiscreteEnergy`.

If the answer is "no"—multiplying the dates and durations does not change the capacity by the same factor—then the measure of capacity is not a measure of energy, and the resource cannot be represented by an instance of `IloDiscreteEnergy`.

In the example above, defining a week as a period of 14 days instead of 7 does not change the number of workers. On the other hand, it's possible that each worker would agree to work up to 10 days out of 14. The number of workers does not change, but the number of available worker-days is proportional to the length of the week. In that situation, where the workers agree to work 10 days out of 14, you should use an instance of `IloDiscreteEnergy`.

State Resources

In Scheduler, the class `IloStateResource` is available to represent resources that can operate in different states. An instance of `IloStateResource` represents a resource of infinite capacity, the state of which can vary over time. An activity may require the resource to be in a particular state or to remain in one of a given set of states throughout its execution. Consequently, two activities that require inconsistent states cannot overlap.

The transition time between states might vary according to the previous state and target state. In the example of a pastry bakery shop requiring different oven temperatures for different pastries, it may take longer for a warm oven to heat to very high temperature than it would take for a moderately hot oven to heat to very high temperature. The way to define your own transition times for state resources is the same as for unary resources: that is, by providing an instance of the parameter class `IloTransitionTime` to the resource.

Different capacity enforcement levels are available to take into account the characteristics of the requirement and provision of a state resource.

- The default capacity enforcement level is `IloBasic`. It is interpreted to allow the set of possible states to vary over time: at any given time the resource may or may not be allowed to be in a given state.
- Higher levels of enforcement are interpreted to allow further constraint propagation with respect to the requiring activities: the earliest and latest start and end times of activities are updated to insure that the time intervals over which two activities that require incompatible states do not overlap.

In the context of an instance of `IloStateResource`, a state is defined as a pointer to any type of object (that is, a pointer of type `IloAny`).

The Scheduler extractor interprets the `IloBasic` capacity enforcement level with the use of the timetable constraint, and the class `IlcAnyTimetable` represents the evolution of the state of the resource over time. Higher levels of capacity enforcement use the timetable plus the disjunctive constraint. Here, we must emphasize that state resources are different from unary resources with respect to disjunctive constraints. Disjunctive constraints on unary resources tend to increase propagation, a distinct advantage in the search for a solution. However, in contrast to what happens with unary resources, the timetable representation for state resources may propagate more than the disjunctive representation. The use of a timetable may even be mandatory to represent some problems. The *IBM ILOG Scheduler Reference Manual* provides details about this phenomenon in the description of the class `IlcStateResource`. It is possible to use the disjunctive constraint without the timetable only by using the Scheduler Engine classes documented in *Part II* of the *IBM ILOG Scheduler Reference Manual*.

For state resources, you can define transition times only when the disjunctive constraint is posted. In the case where transition times are defined and the enforcement level is `IloBasic`, the disjunctive constraint is automatically added at extraction time.

Transition Costs

Between each pair of consecutive activities on a unary resource, some cost may be incurred to switch the resource from processing one activity to processing the next. These costs may be related to modifications to the resource, such as adjusting or purging a machine. The modifications may require manpower, material, and energy.

In Scheduler, *transition cost* is defined as the cost between two immediately successive activities scheduled on a unary resource. In addition, Scheduler lets you define a setup cost for the activity that starts the usage of the resource and a teardown cost for the activity that ends the usage of the resource.

Two types of transition times and costs can be modeled in Scheduler: constant and variable.

Constant transition costs only depend upon the precedence relationship between two activities. In that case, as shown in Adding Transition Times, Scheduler associates a transition type with each activity. The accessors are the functions `IloActivity::getTransitionType` and `IloActivity::setTransitionType`. These transition types allow you to define constant transition times and costs that are represented by the class `IloTransitionParam`. This class consists of a table of integers indexed by the transition type of activities.

The evaluation of variable transition times and costs depends upon current knowledge about the other variables and constraints involved. The macros `ILOTRANSITIONTIMEOBJECT` and `ILOTRANSITIONCOSTOBJECT` allow you to define variable transition times and costs in Scheduler. More precisely, these macros allow you to define and implement the extraction of an `IloTransitionTimeObject` or `IloTransitionCostObject` into your own `IlcTransitionTimeObject` or `IlcTransitionCostObject`, respectively. In the case of variable transition costs, the `Ilc` version of the transition cost object must define its minimal and maximal value given the current knowledge about the sequence.

Transition Types, Transition Times, and Timetables

In Scheduler, we can associate a *transition type* with an activity. This type, represented by a non-negative integer, allows the definition of transition times and costs between activities.

Basic Functions

First we look at basic functions for expressing transition times on typed activities.

Defining Activity Types

To specify the type of an activity, the following function is provided:

```
void IloActivity::setTransitionType(IloInt type);
```

where `type` must be a non-negative integer. By default the type of an activity is equal to zero.

We recommend using as few types as possible. If the number of different types is *N*, define the types between *0* and *N-1*.

Defining Transition Times

To define transition times between typed activities, define a square matrix of all transition times between types. The class `IloTransitionParam` has been designed for building such a matrix.

For instance, we can use the following code to define transition times between the types 0, 1 and 2.

```
IloTransitionParam table(env, 3);
```

The digit 3 represents the number of types and in consequence the types of activities must be between 0 and 2.

By default, transition times are assumed to be asymmetric; that is, the transition time between activities *A1* and *A2* depends on which activity, *A1* or *A2*, is first.

To create a symmetric table, an extra argument set to `IloTrue` must be passed to the constructor of `IloTransitionParam`, as follows.

```
IloTransitionParam table(env, 3, IloTrue);
```

To specify the transition times between two types, the function `setValue` is provided.

```
void IloTransitionParam::setValue(IloInt typeA,
                                  IloInt typeB,
                                  IloInt transTime);
```

Once an instance of `IloTransitionParam` has been defined, define an instance of `IloTransitionTime` by passing the `IloTransitionParam` and a resource (discrete, unary, discrete energy, or state resource) to the constructor. An instance of `IloTransitionTime` can be created, for example, from a function defining the time to be the sum of the processing time of the two activities *A1* and *A2* divided by 2.

The following code applies the `IloTransitionParam` instance `table` to the `IloUnaryResource` instance `worker`.

```
IloUnaryResource worker(env);
IloTransitionTime tTime(worker, table);
```

Choosing the Enforcement Level

The enforcement level allows specifying with how much effort a given constraint may be expressed on a given resource. For example, a break list or transition times may be expressed

on a given resource. One can specify the enforcement level of those constraints, thereby specifying how much effort the scheduler will spend at enforcing them.

`IloBasic` is the default enforcement level. `IloLow` and `IloMediumLow` represent enforcement levels lower than the default level `IloBasic`. This means that the scheduler will spend less effort at enforcing those constraints than it would do by default.

`IloMediumHigh`, `IloHigh` and `IloExtended` correspond to a scale of enforcement levels higher than the default level `IloBasic`. These levels specify that the scheduler will spend more effort at enforcing those constraints than it would do by default.

The higher enforcement levels typically cause more propagation of constraints; this results in fewer fails and fewer choice points, but more CPU time consumption in each search state. Also, the use of enforcement level should be chosen in accordance with the resource and how it is being used. See Recommendations for Capacity Enforcement.

The enforcement levels, when extracted, correspond to the use of different combinations of global constraints (for example, *disjunctive constraint* or *type timetable constraint*). You may use these constraints directly if so desired. See *Resource Enforcement as Global Constraint Declaration* in the *IBM ILOG Scheduler Reference Manual*.

The startsAfterEnd Constraint

To improve the performance of an application in which transition times are used, express temporal constraints using transition times with the member function `startsAfterEnd`, if possible.

In other words, if activity *A1* must start after the completion time of activity *A2* and there is a transition time *T* between *A2* and *A1*, we recommend expressing the transition time with the function `startsAfterEnd` like this:

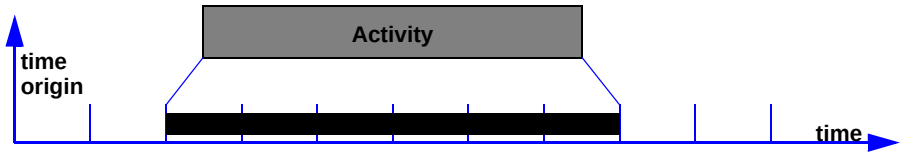
```
model.add(A1.startsAfterEnd(A2, T));
```

Passing the transition time in the `startsAfterEnd` function is redundant but can improve the performance of the application.

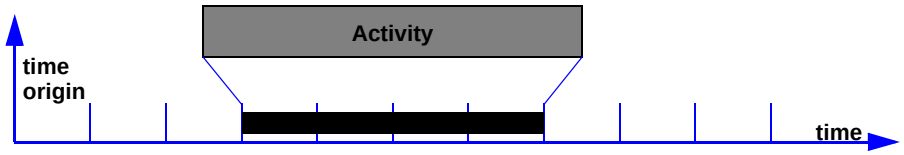
Rounding Mode on Resources

When the time step of a resource is greater than one, a decision must be made as to how to fit the start and end times of an activity into the time steps. For that purpose, an argument can be passed when creating a resource constraint to specify whether the rounding of the interval defining the activity should be inward or outward.

- Outward rounding insures that *at least* the required capacity is available. This type of rounding is most useful for a consumption hypothesis.



- Inward rounding insures that *at most* the required capacity is available. This type of rounding is most useful for a production hypothesis.



Recommendations for Capacity Enforcement

The capacity enforcement levels provide an easy way for you to explore the trade-off between the effort spent in each search state and the overall CPU time to solve the problem. In easy problems it is often better to spend little time at each search state and depend on the non-deterministic search to find a solution. In contrast, on harder problems, more effort at each search state in enforcing the capacity constraints often leads to significant overall savings in terms of CPU time in solving the problem.

In using any of the Concert Technology resource classes (for example, `IloUnaryResource`, `IloDiscreteResource`, etc.) the following guidelines will help to focus your exploration of the trade-off:

- Start with the `IloBasic` capacity enforcement level. This is the default level.
- Use `setCapacityMin` and `setCapacityMax` to define capacity constraints.
- Do not keep the resource open.
- Experiment with the use of higher and lower capacity enforcement levels.

There is a special case with unary resources (instances of the class `IloUnaryResource`) when the minimum capacity on the resource is always zero and the maximum capacity changes at only a few time points across the horizon. In such a case, you might try to introduce “fake activities” during time intervals where the maximum capacity must be zero,

rather than using the `setCapacityMax` function. In some cases, such a representation will provide better performance.

If you use the extracted Scheduler resource classes (for example, `IlcUnaryResource`, `IlcDiscreteResource`, etc.), then use the following tables for general recommendations about how to enforce capacity constraints. Unlike with Concert Technology, these recommendations depend on the class of resource. We give recommendations only for discrete resources, unary resources, discrete energy resources, reservoirs, and continuous reservoirs.

Table 11.1 provides recommendations for cases where *minimal* capacity constraints apply.

Table 11.1 *Propagating Resources—Minimal Capacity Constraints Defined*

Resource class	Recommendations
`IlcDiscreteResource`	1. Use a timetable.
	2. Use `setCapacityMin` and `setCapacityMax` to define capacity constraints.
	3. Close the resource.
	4. Try to add a disjunctive constraint (redundant in general).
	5. Try to use `setEdgeFinder` (maybe introduce “fake” activities instead of using `setCapacityMax`).
`IlcUnaryResource`	1. Use a timetable.
	2. Use `setCapacityMin` and `setCapacityMax` to define capacity constraints.
	3. Close the resource.
	4. Try to add the disjunctive constraint (redundant in general but necessary for taking transition times into account).
	5. Try to use `setEdgeFinder` (maybe use “fake” activities).
`IlcDiscreteEnergy`	1. Use a timetable.
	2. Use `setEnergyMin` and `setEnergyMax` to define energy constraints.
	3. Close the resource.

Table 11.1 *Propagating Resources—Minimal Capacity Constraints Defined*

Resource class	Recommendations
`IlcReservoir`	1. Use a timetable.
	2. Use `setLevelMin` and `setLevelMax` to define level constraints.
	3. Close the resource.
	4. Try to use the balance constraint if there are precedence constraints.
`IlcContinuousReservoir`	1. Use a timetable.
	2. Use `setLevelMin` and `setLevelMax` to define level constraints.
	3. Close the resource.

Table 11.2 provides recommendations in cases where *no* minimal capacity constraints apply. For instances of the class `IlcUnaryResource`, the recommendations depend on whether the *maximal* resource capacity varies over time.

Table 11.2 *Propagating Resources—No Minimal Capacity Constraints Defined*

Resource class	Max. capacity variations	Recommendations
`IlcDiscreteResource`	Yes/No	1. Use a timetable.
		2. Use `setCapacityMax` to define capacity constraints.
		3. Do not close the resource.
		4. Try to add the disjunctive constraint (redundant in general).
		5. Try to use `setEdgeFinder` (maybe introduce “fake” activities instead of using `setCapacityMax`).

Table 11.2 *Propagating Resources—No Minimal Capacity Constraints Defined (Continued)*

Resource class	Max. capacity variations	Recommendations
`IlcUnaryResource`	Yes (many)	1. Use a timetable.
		2. Use `setCapacityMax` to define capacity constraints.
		3. Do not close the resource.
		4. Try to add the disjunctive constraint (redundant in general but necessary for taking transition times into account).
		5. Try to use `setEdgeFinder.`
`IlcUnaryResource`	Yes (few)	1. Use a disjunctive constraint.
		2. Introduce “fake” activities.
		3. Close the resource.
		4. Try to use `setEdgeFinder.`
`IlcDiscreteEnergy`	Yes/No	1. Use a timetable.
		2. Use `setEnergyMax` to define energy constraints.
		3. Do not close the resource.
`IlcReservoir`	Yes/No	1. Use a timetable.
		2. Use `setLevelMax` to define capacity constraints.
		3. Close the resource.
		4. Try to use the balance constraint if there are precedence constraints.
`IlcContinuousReservoir`	Yes/No	1. Use a timetable.
		2. Use `setLevelMax` to define energy constraints.
		3. Do not close the resource.

Scheduling Algorithms

Before you start studying the examples in Part II, , we want to say a few words about scheduling algorithms. For the problems in Part I, , very simple algorithms are used to solve the problems. In fact, for the first problem, an exact algorithm exists for computing an optimal solution without backtracking. Later problems, in contrast, necessitate a search for optimal solutions.

The most important variables of a scheduling problem are, in most cases, those that represent the start times and the end times of activities. Picking a constrained variable and setting a choice point at each of its values is not an efficient method for exploring the search space because, in most cases, changing the start or the end time of an activity from a value `v` to a value `v-1` or `v+1` has little or no effect on the quality of a solution.

This part presents more efficient algorithms for exploring the search space. Indeed, Part III, focuses on search. Many of the algorithms presented in this part should be taken only as starting points. Often it is a good idea to think about problem-specific heuristics that can be used to solve your problem. Furthermore, when reusing an algorithm, you should check whether the algorithm really solves the problem. For example, the algorithms used in the chapters dealing exclusively with unary resources determine only an ordering of the activities on each of the resources. If you add constraints that require the exact placement of the activities in time, that ordering algorithm will not suffice.

You should also be careful when combining algorithms. Some algorithms are based on scheduling one resource at a time, whereas the algorithms used for scheduling discrete resources are based on scheduling forward in time. For a problem in which both unary and discrete resources are used, you could simply use the algorithm for the discrete resources, as a unary resource is just a discrete resource of capacity 1. If a simple ordering of the activities for the unary resources would suffice, you could also combine the two kinds of algorithms by first scheduling the unary resources, and after that, scheduling the discrete resources. If you do that, you have to pay close attention to separating those activities that use the unary resources and those that use the discrete resources. If such a separation is hard to carry out, then we recommend using the algorithm for discrete resources.

Again, we stress that you should take the examples in this part only as a starting point. The more you get acquainted with Solver and Scheduler and with the problem you are going to solve, the better you will be able to implement an algorithm that suits your problem.

C H A P T E R

12

Using Scheduler with Solver

The aim of this first example is to illustrate the combined use of Solver and Scheduler. Scheduler *activities* and *resources* are used with Solver *variables* and *constraints* to represent the problem, and a Solver *goal* is used to search for an optimal solution.

Describing the Problem

This example is organized around the problem of building a house—its foundation, masonry, roofing, painting, and so forth—with the obvious goal of scheduling the tasks to finish the entire house as quickly as possible. Some activities must necessarily take place before others, and that fact is expressed through *precedence constraints*. All activities require one worker, a unary resource, and that fact is expressed through resource constraints.

The house construction activities and their preceding activities are shown in Table 12.1.

Table 12.1 *House Construction Activities*

Activity	Duration	Preceding Activities
masonry	7	
carpentry	3	masonry
plumbing	8	masonry

Table 12.1 *House Construction Activities (Continued)*

Activity	Duration	Preceding Activities
ceiling	3	masonry
roofing	1	carpentry
painting	2	ceiling
windows	1	roofing
facade	2	roofing, plumbing
garden	1	roofing, plumbing
moving	1	windows, facade, garden, painting

This is the same example presented in Adding Resources and Resource Constraints Chapter 3. The problem is fully described and modeled there, so let's discuss the different solution technique used here.

Solve the Problem

The difference in this solution is that rather than using one of the predefined Concert Technology goals to implement search, we will implement our own goal to perform the search.

The following algorithm will be used: while some activity is unscheduled (that is, does not have a fixed start time), select the most urgent unscheduled activity (that is, the activity with the minimal latest start time) and schedule it as early as possible (that is, fix its start time to its earliest start time, as obtained with respect to the activities already scheduled).

We use an iterator (an instance of `IlcActivityIterator`) to step through the activities in the schedule. Scheduler guarantees the update, after each scheduling decision, of the earliest and latest start times of all the activities.

Note that the code of the goal and the activity selector are written using the Solver classes and the extracted Scheduler classes rather than Concert Technology classes. This is due to the fact that the goal and the activity selector are executed during the search for a solution which is controlled by the non-deterministic goal functionality of Solver. To make our activity selection, for example, we need the updated earliest start and latest end times of each activity given the current state of search. If we were to try to directly access these values from the `IloActivity` objects in the model, we would simply get the earliest start and latest end times from the model. These values are not updated during search, therefore we need to explicitly access the values from the `IlcActivity` objects.

`IlcBool`

```
SelFirstActStartMax(IlcActivity& selection, const IlcScheduler& scheduler)
{

  /* IF A NOT BOUNDED ACTIVITY EXISTS IN THE SCHEDULE, RETURN ILCTRUE
     AND SET SELECTION AS THE ACTIVITY WITH THE MINIMAL LATEST START TIME. */
  IlcInt startmax = IlcIntMax;
  IlcBool selected = IlcFalse;
  for (IlcActivityIterator iterator(scheduler);
       iterator.ok();
       ++iterator)
  {
    IlcActivity activity = *iterator;
    if (!activity.getStartVariable().isBound()
        && (activity.getStartMax() <= startmax))
    {
      startmax = activity.getStartMax();
      selection = activity;
      selected = IlcTrue;
    }
  }
  return selected;
}

ILCGOAL0(SolveIlc) {
  IloSolver solver = getSolver();
  IlcActivity chosenActivity;
  if (SelFirstActStartMax(chosenActivity, IlcScheduler(solver))) {
    chosenActivity.setStartTime(chosenActivity.getStartMin());
    return this;
  }
  else
    return 0;
}

ILOCPGOALWRAPPER0(Solve, solver) {
  return SolveIlc(solver);
}
```

At each step in the problem solving, the activity selector `SelFirstActStartMax` is used to select an activity with the minimal possible latest start time (result of the `activity.getStartMax()` statement) among those activities which are still unscheduled (that is, for which the Boolean statement `activity.getStartVariable().isBound()` is false). The chosen activity is then scheduled at its earliest possible start time. This choice automatically and transparently triggers constraint propagation, which results in the update of the earliest and latest start times to be used in the next iteration.

The iterative process terminates when all the activities have been scheduled, that is, when the activity selector `SelFirstActStartMax` returns `IlcFalse`.

Complete Program and Output

You can see the entire program `greedy.cpp` here or view it in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

#if defined(ILO_SDXLOUTPUT)
#include "sdxloutput.h"
#endif

////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////


IloModel DefineModel(IloEnv& env, IloNumVar& makespan)
{
  IloModel model(env);

  /* CREATE THE ACTIVITIES. */
  IloActivity masonry(env,   7, 0, "masonry   ");
  IloActivity carpentry(env, 3, 0, "carpentry ");
  IloActivity plumbing(env,  8, 0, "plumbing  ");
  IloActivity ceiling(env,   3, 0, "ceiling   ");
  IloActivity roofing(env,   1, 0, "roofing   ");
  IloActivity painting(env,  2, 0, "painting  ");
  IloActivity windows(env,   1, 0, "windows   ");
  IloActivity facade(env,    2, 0, "facade    ");
  IloActivity garden(env,    1, 0, "garden    ");
  IloActivity moving(env,    1, 0, "moving    ");

  /* ADD THE TEMPORAL CONSTRAINTS. */
  model.add(carpentry.startsAfterEnd(masonry));
  model.add(plumbing.startsAfterEnd(masonry));
  model.add(ceiling.startsAfterEnd(masonry));
  model.add(roofing.startsAfterEnd(carpentry));
  model.add(painting.startsAfterEnd(ceiling));
  model.add(windows.startsAfterEnd(roofing));
  model.add(facade.startsAfterEnd(roofing));
  model.add(facade.startsAfterEnd(plumbing));
  model.add(garden.startsAfterEnd(roofing));
  model.add(garden.startsAfterEnd(plumbing));
  model.add(moving.startsAfterEnd(windows));
  model.add(moving.startsAfterEnd(facade));
  model.add(moving.startsAfterEnd(garden));
  model.add(moving.startsAfterEnd(painting));


  /* CREATE THE RESOURCE. */
  IloUnaryResource worker(env);

  /* ADD THE RESOURCE CONSTRAINTS. */
  model.add(masonry.requires(worker));
  model.add(carpentry.requires(worker));
  model.add(plumbing.requires(worker));
  model.add(ceiling.requires(worker));
  model.add(roofing.requires(worker));
```

```
  model.add(painting.requires(worker));
  model.add(windows.requires(worker));
  model.add(facade.requires(worker));
  model.add(garden.requires(worker));
  model.add(moving.requires(worker));

  /* SET THE MAKESPAN VARIABLE. */
  makespan = IloNumVar(env, 0, IloInfinity, ILOINT);
  model.add(moving.endsAt(makespan));

  /* SET THE OBJECTIVE */
  model.add(IloMinimize(env, makespan));

  return model;
}

IlcBool
SelFirstActStartMax(IlcActivity& selection, const IlcScheduler& scheduler)
{

  /* IF A NOT BOUNDED ACTIVITY EXISTS IN THE SCHEDULE, RETURN ILCTRUE
     AND SET SELECTION AS THE ACTIVITY WITH THE MINIMAL LATEST START TIME. */
  IlcInt startmax = IlcIntMax;
  IlcBool selected = IlcFalse;
  for (IlcActivityIterator iterator(scheduler);
       iterator.ok();
       ++iterator)
  {
    IlcActivity activity = *iterator;
    if (!activity.getStartVariable().isBound()
        && (activity.getStartMax() <= startmax))
    {
      startmax = activity.getStartMax();
      selection = activity;
      selected = IlcTrue;
    }
  }
  return selected;
}

ILCGOAL0(SolveIlc) {
  IloSolver solver = getSolver();
  IlcActivity chosenActivity;
  if (SelFirstActStartMax(chosenActivity, IlcScheduler(solver))) {
    chosenActivity.setStartTime(chosenActivity.getStartMin());
    return this;
  }
  else
    return 0;
}

ILOCPGOALWRAPPER0(Solve, solver) {
  return SolveIlc(solver);
}

////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
```

```
//
////////////////////////////////////////////////////////////////////////////

void PrintSolution(const IloSolver& solver)
{
  IlcScheduler scheduler(solver);
  IloEnv env = solver.getEnv();
  for(IloIterator<IloActivity> act(env);
      act.ok();
      ++act)
    env.out() << scheduler.getActivity(*act) << endl;
  solver.printInformation();
}

////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////

int main()
{
  try {

    IloEnv env;
    IloNumVar makespan;
    IloModel model = DefineModel(env, makespan);

    IloSolver solver(model);
    IloGoal goal = Solve(env);

    if (solver.solve(goal)) {
      PrintSolution(solver);
#if defined(ILO_SDXLOUTPUT)
      IloSDXLOutput output(env);
      ofstream outFile("greedy.xml");
      output.write(IlcScheduler(solver), outFile, solver.getIntVar(makespan));
      outFile.close();
#endif
    }
    else
      solver.out() << "No Solution" << endl;

    env.end();

  } catch (IloException& exc) {
    cout << exc << endl;
  }
  return 0;
}

////////////////////////////////////////////////////////////////////////////
//
// RESULTS
//
////////////////////////////////////////////////////////////////////////////

/*
```

```
masonry    [0 -- 7 --> 7]
carpentry  [15 -- 3 --> 18]
plumbing   [7 -- 8 --> 15]
ceiling    [18 -- 3 --> 21]
roofing    [21 -- 1 --> 22]
painting   [22 -- 2 --> 24]
windows    [26 -- 1 --> 27]
facade     [24 -- 2 --> 26]
garden     [27 -- 1 --> 28]
moving     [28 -- 1 --> 29]
*/
```

The start and end times of all activities are fixed in this example.

Figure 12.1 provides a graphic display of the solution to our problem.

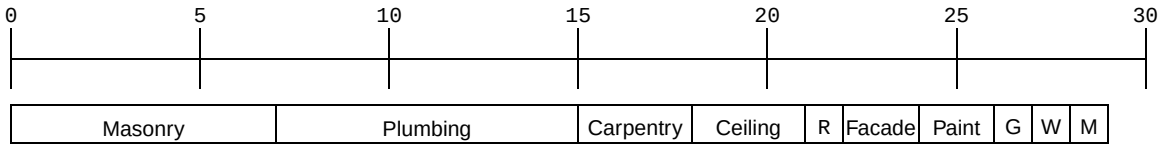


Figure 12.1 *Solution Using Solver Goal*

C H A P T E R

13

More Advanced Problem Modeling with Concert Technology

The aim of the following example is to demonstrate some advanced problem modeling techniques using Concert Technology. In addition to activities and resources, Solver *variables* and *constraints* are used to represent the problem. Non-deterministic functions from Solver are then used to search for an optimal solution.

The problem is to assign each activity to a start time and to a specific worker (chosen among a set of workers able to perform the activity) taking care of the fact that the same worker cannot perform two activities at the same time.

Two optimization criteria are considered: either minimize the *maximal* or the *average* amount of time spent by each worker on the site, given that a worker cannot leave the site between two activities. By default, we will assume that the maximal amount of time spent by each worker should be minimized. However, we'll use `-minimizeSum`, an optional argument to the `main` program, to specify that the sum (or, equivalently, the average value) of the amounts of time spent must be minimized.

Describing the Problem

The activities of the problem are the same as those presented in Chapter 12, but now we have three workers. Each activity will be performed by a member of the set of `{joe jack jim}`. The following table provides the duration and the possible workers of each activity.

Table 13.1 *House Construction Activity Assignments*

Activity	Duration	Possible Workers
masonry	7	joe or jack
carpentry	3	joe or jim
plumbing	8	jack
ceiling	3	joe or jim
roofing	1	joe or jim
painting	2	jack or jim
windows	1	joe or jim
facade	2	joe or jack
garden	1	joe or jack or jim
moving	1	joe or jim

The data is provided in the form of a matrix and three arrays.

- `ActivityNames`, containing the names of the ten activities,
- `ProcessingTimes`, containing the processing times of the ten activities,
- `WorkerNames`, containing the names of the three workers.

For each activity and each worker, the matrix `PossibleAssignments` specifies whether or not the worker is able to perform the activity. For the sake of clean software-engineering, only the `main` function refers to these global variables. When we pass the value of a global variable as an argument to another function, we derive the name of the argument from the name of the global variable, but we do not capitalize the first character of the argument name.

```
IloInt NumberOfActivities = 10;
IloInt NumberOfWorkers = 3;
const char* ActivityNames [] = {“masonry   “,
                                “carpentry “,
                                “plumbing  “,
                                “ceiling   “,
                                “roofing   “,
                                “painting  “,
                                “windows   “,
                                “facade    “,
```

```
                                “garden    “,
                                “moving    “};
const char* WorkerNames [] = {“joe”,
                              “jack”,
                              “jim”};
IloNum ProcessingTimes  [] = {7, 3, 8, 3,  1,  2,  1,  2,  1,  1};
const char* PossibleAssignments [] = {“joe”, “jack”,     0,
                                      “joe”,      0, “jim”,
                                      0, “jack”,     0,
                                      “joe”,      0, “jim”,
                                      “joe”,      0, “jim”,
                                      0, “jack”, “jim”,
                                      “joe”,      0, “jim”,
                                      “joe”, “jack”,     0,
                                      “joe”, “jack”, “jim”,
                                      “joe”,      0, “jim”};
```

As you can see from the matrix, either `joe` or `jack` can do masonry, but `jim` cannot. Only `jack` can do plumbing, but any of the three can work in the garden, and so forth.

Defining the Problem, Designing a Model

We’ll define our problem in four parts: the schedule itself, the workers, the optimization criteria, and the activities.

Time Horizon

In this example, we choose the time origin to be 0 and time horizon to be 18. For this kind of problem, nothing guarantees that a solution exists. In fact, for all we know at this point, some of the constraints may be incompatible with one another, precluding a solution to the problem.

Workers

There are two important facts to represent with respect to each worker. First, the worker can perform only one activity at a time: each individual worker is a *unary resource*. Second, the worker must be at the construction site in order to perform an activity. In other words, we need to preclude the worker’s performance of any activity when he or she is not present on the site.

In that sense, the worker is already “required” whenever he or she is not on the site. The use of `IloTimeExtent` in a `requires` statement makes it possible to represent this constraint.

The code that follows defines a new class, `Worker`. Each instance of the class `Worker` includes an instance of `IloUnaryResource` and an instance of `IloActivity`. The activity represents the presence of the worker on the site and is said to require the resource before its

start and after its end. This guarantees that the worker cannot perform any other activity before arrival at the site nor after departure from the site.

```
class Worker {
private:
  IloUnaryResource _resource;
  IloActivity      _limitedAttendanceOnSite;
public:
  Worker(IloModel, IloNum durMaxOnSite, const char* name);
  ~Worker();
  IloUnaryResource getResource() const {
    return _resource;
  }
  IloActivity getAttendance() const {
    return _limitedAttendanceOnSite;
  }
  IloNumVar getAttendanceProcessingTime() const {
    return _limitedAttendanceOnSite.getProcessingTimeVariable();
  }
  IloConstraint coverAttendance(IloActivity act);
};

Worker::Worker(IloModel model, IloNum durMaxOnSite, const char* name)
  :_resource(model.getEnv()),
   _limitedAttendanceOnSite(model.getEnv(),
                            IloNumVar(model.getEnv(), 0, durMaxOnSite,
IloNumVar::Int))
{
  _resource.setName(name);
  _resource.setCapacityEnforcement(IloMediumHigh);
  _limitedAttendanceOnSite.setName("Attendance");
  _resource.setObject(this);
  /* NEGATIVE STATEMENT: THE FACT THAT THE WORKER IS ON THE SITE FOR
     A LIMITED INTERVAL OF TIME MEANS THAT IT IS TO BE CONSIDERED
     "REQUIRED" BEFORE AND AFTER THIS INTERVAL OF TIME. */
  model.add(_limitedAttendanceOnSite.requires(_resource,
                                              1,
                                              IloBeforeStartAndAfterEnd));
}

Worker::~Worker()
{}

IloConstraint Worker::coverAttendance(IloActivity act){
  return (act.startsAfterStart(_limitedAttendanceOnSite) &&
          _limitedAttendanceOnSite.endsAfterEnd(act));
}
```

Optimization Criteria

The two optimization criteria are easy to define from the processing times of the "attendance activities." The `maxCriterion` is a variable constrained to be greater than or equal to all of these processing times. The `sumCriterion` is a variable defined as the sum of these variables.

Two parameters, `sumCriterion` and `maxCriterion`, can be passed by reference to the function `DefineProblem` and set when the instances of the `Worker` class are created. In our model, the workers are represented by an instance of the class `IloAltResSet`, defining the fact that the workers can be viewed as alternatives to each other. If an activity requires an instance of the class `IloAltResSet`, this means that it requires exactly one of the alternatives in that instance.

```
IloModel
DefineProblem(IloEnv env,
              IloInt numberOfActivities,
              IloInt numberOfWorkers,
              const char** activityNames,
              const char** workerNames,
              IloNum* processingTimes,
              const char** possibleAssignments,
              IloAltResSet& workersSet,
              IloNumVar& maxCriterion,
              IloNumVar& sumCriterion)
{

  /* CREATE THE MODEL. */
  IloNum origin = 0;
  IloNum horizon = 18;

  IloModel model(env);
  IloSchedulerEnv schedEnv(env);
  schedEnv.setOrigin(origin);
  schedEnv.setHorizon(horizon);

  /* CREATE THE WORKERS AND SET THE OPTIMIZATION CRITERIA. */
  IloUnaryResourceArray workers(env, numberOfWorkers);
  IloNumVarArray attendancies(env, numberOfWorkers);
  IloInt i;
  for (i = 0; i < numberOfWorkers; i++) {
    Worker* worker =
      new (env) Worker(model, horizon, workerNames[i]);
    attendancies[i] = worker->getAttendanceProcessingTime();
    workers[i]      = worker->getResource();
    workersSet.add(workers[i]);
  }

  sumCriterion = IloNumVar(env, 0, numberOfWorkers * horizon, IloNumVar::Int);
  maxCriterion = IloNumVar(env, 0, horizon, IloNumVar::Int);
  model.add(sumCriterion == IloSum(attendancies));
  model.add(maxCriterion == IloMax(attendancies));

  /* ... */
}
```

Activities

In a solution to the problem, each possible assignment of an activity to a worker may be enforced or not. We use the function `requires` to express the fact that an activity requires one of the alternative workers. The member function `setRejected` expresses the idea that a certain resource cannot execute a certain activity. The call to the member function `select`

expresses the fact that if some worker is selected for an activity, the activity has to occur during the “attendance on site” activity. In fact, this expresses the same idea as the time extent `IloBeforeStartAndAfterEnd`, and either of the two methods would suffice. We use the function `select` to demonstrate its use.

```
IloModel
DefineProblem(IloEnv env,
              IloInt numberOfActivities,
              IloInt numberOfWorkers,
              const char** activityNames,
              const char** workerNames,
              IloNum* processingTimes,
              const char** possibleAssignments,
              IloAltResSet& workersSet,
              IloNumVar& maxCriterion,
              IloNumVar& sumCriterion)
{

  /* ... */

  /* CREATE THE ACTIVITIES AND THE ASSIGNMENT VARIABLES. IT IS ASSUMED
     THAT THERE IS AT LEAST ONE POSSIBLE ASSIGNMENT PER ACTIVITY. */
  IloInt k = 0;
  IloActivityArray tasks(env,numberOfActivities);
  for (i = 0; i < numberOfActivities; i++) {
    IloActivity activity(env, processingTimes[i]);
    activity.setName(activityNames[i]);
    tasks[i] = activity;
    // RESOURCE ALLOCATION DEMAND
    IloResourceConstraint alternative = activity.requires(workersSet, 1);
    // REMOVE UNAVAILABLE WORKERS
    for (IloInt j = 0; j < numberOfWorkers; j++) {
      if (!possibleAssignments[k]) {
        alternative.setRejected(workers[j]);
      } else {
        model.add(IloIfThen(env,
                            alternative.select(workers[j]),
                            (((Worker*)
                              workers[j].getObject())-
>coverAttendance(activity))));
      }
      k++;
    }
    model.add(alternative);
  }

  /* ... */
}
```

Solving the Problem

Once we have represented the issues of the problem in those ways, the following code is sufficient to generate a solution that minimizes the given `criterion`.

```
    IloNumVar criterion = ((minimizeMax) ? maxCriterion : sumCriterion);
    model.add(IloMinimize(env, criterion));

    IloSolver solver(model);
    IloGoal goal = IloAssignAlternative(env) &&
      IloSetTimesForward(env,
                         criterion,
                         IloSelFirstActMinEndMin);

    if (solver.solve(goal)) {
      IlcInt maxCrit = solver.getIntVar(maxCriterion).getMin();
      IlcInt sumCrit = solver.getIntVar(sumCriterion).getMin();
      PrintSolution(solver, maxCrit, sumCrit);
#if defined(ILO_SDXLOUTPUT)
      IloSDXLOutput output(env);
      ofstream outFile("cassign.xml");
      output.write(IlcScheduler(solver), outFile);
      outFile.close();
#endif
    } else
      solver.out() << "No solution!" << endl;
    solver.printInformation();
    env.end();
  } catch (IloException& exc) {
    cout << exc << endl;
  }

  return 0;
```

The goal returned by the function `IloAssignAlternative` assigns a resource to all activities. The goal returned by `IloSetTimesForward` assigns a start time to all activities, starting with the activity that has the minimal earliest start time among those activities with the minimal earliest end times.

Complete Program and Output

You can see the entire program `cassign.cpp` here or online in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

#if defined(ILO_SDXLOUTPUT)
#include "sdxloutput.h"
#endif

IloInt NumberOfActivities = 10;
IloInt NumberOfWorkers = 3;
const char* ActivityNames [] = {"masonry   ",
                                "carpentry ",
                                "plumbing  ",
                                "ceiling   ",
                                "roofing   ",
                                "painting  ",
```

```
                                “windows   “,
                                “facade    “,
                                “garden    “,
                                “moving    “};
const char* WorkerNames [] = {“joe”,
                              “jack”,
                              “jim”};
IloNum ProcessingTimes  [] = {7, 3, 8, 3,  1,  2,  1,  2,  1,  1};
const char* PossibleAssignments [] = {“joe”, “jack”,     0,
                                      “joe”,      0, “jim”,
                                      0, “jack”,     0,
                                      “joe”,      0, “jim”,
                                      “joe”,      0, “jim”,
                                      0, “jack”, “jim”,
                                      “joe”,      0, “jim”,
                                      “joe”, “jack”,     0,
                                      “joe”, “jack”, “jim”,
                                      “joe”,      0, “jim”};

/* THE PROBLEM IS TO MINIMIZE
         EITHER (1) THE MAXIMAL
         OR     (2) THE AVERAGE
   AMOUNT OF TIME SPENT BY EACH WORKER ON THE CONSTRUCTION SITE GIVEN
   THAT A WORKER CANNOT LEAVE THE SITE BETWEEN TWO ACTIVITIES. */

////////////////////////////////////////////////////////////////////////////
//
// DEFINITION OF THE WORKER CLASS
//
////////////////////////////////////////////////////////////////////////////
class Worker {
private:
  IloUnaryResource _resource;
  IloActivity      _limitedAttendanceOnSite;
public:
  Worker(IloModel, IloNum durMaxOnSite, const char* name);
  ~Worker();
  IloUnaryResource getResource() const {
    return _resource;
  }
  IloActivity getAttendance() const {
    return _limitedAttendanceOnSite;
  }
  IloNumVar getAttendanceProcessingTime() const {
    return _limitedAttendanceOnSite.getProcessingTimeVariable();
  }
  IloConstraint coverAttendance(IloActivity act);
};

Worker::Worker(IloModel model, IloNum durMaxOnSite, const char* name)
  :_resource(model.getEnv()),
   _limitedAttendanceOnSite(model.getEnv(),
                            IloNumVar(model.getEnv(), 0, durMaxOnSite,
IloNumVar::Int))
{
  _resource.setName(name);
  _resource.setCapacityEnforcement(IloMediumHigh);
  _limitedAttendanceOnSite.setName(“Attendance”);
```

```
  _resource.setObject(this);
  /* NEGATIVE STATEMENT: THE FACT THAT THE WORKER IS ON THE SITE FOR
     A LIMITED INTERVAL OF TIME MEANS THAT IT IS TO BE CONSIDERED
     "REQUIRED" BEFORE AND AFTER THIS INTERVAL OF TIME. */
  model.add(_limitedAttendanceOnSite.requires(_resource,
                                              1,
                                              IloBeforeStartAndAfterEnd));
}

Worker::~Worker()
{}

IloConstraint Worker::coverAttendance(IloActivity act){
  return (act.startsAfterStart(_limitedAttendanceOnSite) &&
          _limitedAttendanceOnSite.endsAfterEnd(act));
}

////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////

void
AddTemporalConstraints(IloModel model,
                       IloActivityArray task,
                       IloNum origin)
{
  IloActivity masonry   = task[0];
  IloActivity carpentry = task[1];
  IloActivity plumbing  = task[2];
  IloActivity ceiling   = task[3];
  IloActivity roofing   = task[4];
  IloActivity painting  = task[5];
  IloActivity windows   = task[6];
  IloActivity facade    = task[7];
  IloActivity garden    = task[8];
  IloActivity moving    = task[9];

  /* POST THE TEMPORAL CONSTRAINTS. */
  masonry.setStartMax(origin);
  model.add(carpentry.startsAfterEnd(masonry));
  model.add(plumbing.startsAfterEnd(masonry));
  model.add(ceiling.startsAfterEnd(masonry));
  model.add(roofing.startsAfterEnd(carpentry));
  model.add(painting.startsAfterEnd(ceiling));
  model.add(windows.startsAfterEnd(roofing));
  model.add(facade.startsAfterEnd(roofing));
  model.add(facade.startsAfterEnd(plumbing));
  model.add(garden.startsAfterEnd(roofing));
  model.add(garden.startsAfterEnd(plumbing));
  model.add(moving.startsAfterEnd(windows));
  model.add(moving.startsAfterEnd(facade));
  model.add(moving.startsAfterEnd(garden));
  model.add(moving.startsAfterEnd(painting));
}
IloModel
DefineProblem(IloEnv env,
```

```
                IloInt numberOfActivities,
                IloInt numberOfWorkers,
                const char** activityNames,
                const char** workerNames,
                IloNum* processingTimes,
                const char** possibleAssignments,
                IloAltResSet& workersSet,
                IloNumVar& maxCriterion,
                IloNumVar& sumCriterion)
{
  /* CREATE THE MODEL. */
  IloNum origin = 0;
  IloNum horizon = 18;

  IloModel model(env);
  IloSchedulerEnv schedEnv(env);
  schedEnv.setOrigin(origin);
  schedEnv.setHorizon(horizon);

  /* CREATE THE WORKERS AND SET THE OPTIMIZATION CRITERIA. */
  IloUnaryResourceArray workers(env, numberOfWorkers);
  IloNumVarArray attendancies(env, numberOfWorkers);
  IloInt i;
  for (i = 0; i < numberOfWorkers; i++) {
    Worker* worker =
      new (env) Worker(model, horizon, workerNames[i]);
    attendancies[i] = worker->getAttendanceProcessingTime();
    workers[i]      = worker->getResource();
    workersSet.add(workers[i]);
  }

  sumCriterion = IloNumVar(env, 0, numberOfWorkers * horizon, IloNumVar::Int);
  maxCriterion = IloNumVar(env, 0, horizon, IloNumVar::Int);
  model.add(sumCriterion == IloSum(attendancies));
  model.add(maxCriterion == IloMax(attendancies));

  /* CREATE THE ACTIVITIES AND THE ASSIGNMENT VARIABLES. IT IS ASSUMED
     THAT THERE IS AT LEAST ONE POSSIBLE ASSIGNMENT PER ACTIVITY. */
  IloInt k = 0;
  IloActivityArray tasks(env,numberOfActivities);
  for (i = 0; i < numberOfActivities; i++) {
    IloActivity activity(env, processingTimes[i]);
    activity.setName(activityNames[i]);
    tasks[i] = activity;
    // RESOURCE ALLOCATION DEMAND
    IloResourceConstraint alternative = activity.requires(workersSet, 1);
    // REMOVE UNAVAILABLE WORKERS
    for (IloInt j = 0; j < numberOfWorkers; j++) {
      if (!possibleAssignments[k]) {
        alternative.setRejected(workers[j]);
      } else {
        model.add(IloIfThen(env,
                            alternative.select(workers[j]),
                            (((Worker*)
                              workers[j].getObject())-
>coverAttendance(activity))));
      }
      k++;
```

```
    }
    model.add(alternative);
  }
  AddTemporalConstraints(model, tasks, origin);
  /* RETURN THE CREATED MODEL. */
  return model;
}

////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////////////////

void
PrintSolution(IloSolver solver,
              IlcInt maxCriterionValue,
              IlcInt sumCriterionValue)
{
  IlcScheduler scheduler(solver);
  for (IlcAltResSetIterator resIte(scheduler); resIte.ok(); ++resIte) {
    for(IlcAltResConstraintIterator ite(*resIte); ite.ok(); ++ite) {
      if ((*ite).getNumberOfPossible() == 1)
        solver.out() << (*ite).getSelected() << "\t"
                     << (*ite).getActivity() << endl;
    }
  }
  solver.out() << "MAXIMAL ATTENDANCE ON SITE = " << maxCriterionValue << endl;
  solver.out() << "SUM OF ATTENDANCES ON SITE = " << sumCriterionValue << endl;
}

////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////

void
InitParameters(int argc, char** argv, IloBool& minimizeMax)
{
  if ((argc > 1) && (!strcmp(argv[1], "-minimizeSum")))
    minimizeMax = IloFalse;
}

int main(int argc, char** argv)
{
  try {
    IloBool minimizeMax = IlcTrue;
    InitParameters(argc, argv, minimizeMax);
    IloEnv env;
    IloNumVar maxCriterion, sumCriterion;
    IloAltResSet workersSet(env);
    IloModel model = DefineProblem(env,
                                   NumberOfActivities,
                                   NumberOfWorkers,
                                   ActivityNames,
                                   WorkerNames,
                                   ProcessingTimes,
```

```
                                    PossibleAssignments,
                                    workersSet,
                                    maxCriterion,
                                    sumCriterion);

    IloNumVar criterion = ((minimizeMax) ? maxCriterion : sumCriterion);
    model.add(IloMinimize(env, criterion));

    IloSolver solver(model);
    IloGoal goal = IloAssignAlternative(env) &&
      IloSetTimesForward(env,
                         criterion,
                         IloSelFirstActMinEndMin);

    if (solver.solve(goal)) {
      IlcInt maxCrit = solver.getIntVar(maxCriterion).getMin();
      IlcInt sumCrit = solver.getIntVar(sumCriterion).getMin();
      PrintSolution(solver, maxCrit, sumCrit);
#if defined(ILO_SDXLOUTPUT)
      IloSDXLOutput output(env);
      ofstream outFile("cassign.xml");
      output.write(IlcScheduler(solver), outFile);
      outFile.close();
#endif
    } else
      solver.out() << "No solution!" << endl;
    solver.printInformation();
    env.end();
  } catch (IloException& exc) {
    cout << exc << endl;
  }

  return 0;
}

////////////////////////////////////////////////////////////////////////////
//
// RESULTS
//
////////////////////////////////////////////////////////////////////////////

/* cassign -minimizeMax
   jim [1]        moving     [17 -- 1 --> 18]
   jim [1]        garden     [15 -- 1 --> 16]
   jack [1]       facade     [15 -- 2 --> 17]
   jim [1]        windows    [12 -- 1 --> 13]
   jim [1]        painting   [13 -- 2 --> 15]
   jim [1]        roofing    [11 -- 1 --> 12]
   joe [1]        ceiling    [7 -- 3 --> 10]
   jack [1]       plumbing   [7 -- 8 --> 15]
   jim [1]        carpentry  [8 -- 3 --> 11]
   joe [1]        masonry    [0 -- 7 --> 7]
   MAXIMAL ATTENDANCE ON SITE = 10
   SUM OF ATTENDANCES ON SITE = 30
*/

/* cassign -minimizeSum
   joe [1]        moving     [17 -- 1 --> 18]
```

```
   joe [1]        garden     [15 -- 1 --> 16]
   jack [1]       facade     [15 -- 2 --> 17]
   joe [1]        windows    [16 -- 1 --> 17]
   jim [1]        painting   [11 -- 2 --> 13]
   joe [1]        roofing    [14 -- 1 --> 15]
   joe [1]        ceiling    [8 -- 3 --> 11]
   jack [1]       plumbing   [7 -- 8 --> 15]
   joe [1]        carpentry  [11 -- 3 --> 14]
   jack [1]       masonry    [0 -- 7 --> 7]
   MAXIMAL ATTENDANCE ON SITE = 17
   SUM OF ATTENDANCES ON SITE = 29
*/
```

As you can see from the results, if we want to minimize the time that any one worker spends on site, we'll minimize maximal attendance on site, and get results such as the first one, where all workers spend 10 days.

In contrast, if we minimize the total number of days spent on site for all workers, we'll get a solution like the second one, where `jim` spends only 5 days, `jack` 17, and `joe` stays 7 days on site.

C H A P T E R

14

Scheduling with Unary Resources: the Bridge Problem

A *unary resource* is a resource with capacity one. Scheduling with unary resources is a very special case of scheduling. Indeed, the fact that any two activities requiring a common unary resource cannot overlap implies that these two activities must be put in order: one of the two activities must execute *before* the other simply because their common unary resource is a unit and thus cannot be used in two ways at a time.

More generally, if *n* activities *A1*, *A2*, ..., *An* require the same unary resource, and if none of these activities *Ai* can be started and interrupted for the benefit of another activity *Aj*, then in any problem solution, the *n* activities *A1*, *A2*, ..., *An* must be totally ordered.

Reciprocally, if a scheduling problem consists of activities subject only to *precedence constraints* (as discussed in *Part I, Getting Started with Scheduler)* and to *unary* resource constraints, then it is sufficient to order the activities that require a common unary resource to obtain a solution to the scheduling problem.

In other words, when faced with that kind of problem, you will simply use Scheduler to write an algorithm that puts the activities in order, and that algorithm will actually find a solution to the problem simply by putting the activities to schedule in order.

Indeed, as soon as decisions are made regarding the order in which activities sharing a resource must execute, these decisions apply as additional precedence constraints. When all the necessary ordering decisions have been made—whether by the nature of the problem or by an algorithm that you write for the purpose—the resource constraints are guaranteed to

hold, and the residual problem consists merely of *temporal constraints*, for which constraint propagation is known to be *complete*.

The earliest and latest start and end times of activities under the chosen order are fully determined: they are feasible start and end times.

For this reason, scheduling under temporal and unary resource constraints is often known as "sequencing," "disjunctive scheduling," or "disjunctive," conveying the idea that for each pair `{A B}` of activities that share a resource, either A must be performed before B, or B must be performed before A.

This chapter shows how Scheduler can be used to solve a disjunctive scheduling problem efficiently.

Describing the Problem

The problem is to schedule the construction of a five-segment bridge. To our knowledge, this problem was first published in Germany in a Ph.D. thesis at the University of Passau, and it is very commonly used as a benchmark in the constraint programming community. In this chapter, we offer a simple, efficient program for solving the problem.

The bridge is made of five segments. Five preformed bearers (or horizontal beams) must be placed on top of six vertical pillars: one bearer between any two successive pillars. In the notation generally used for this problem, the activity which consists of putting bearer number `k` on top of pillars number `k` and `k+1` is denoted `Tk`, and we'll follow the same convention, respecting the usual naming conventions in this widely used benchmark.

Constructing each pillar means that four activities must be completed: excavation (denoted `Ak` for pillar number `k`), formwork (`Sk`), concrete foundation (`Bk`), and masonry (`Mk`). In addition:

- The implementation of foundation piles `P1` and `P2` must occur between the excavations of the central pillars `A3` and `A4` and the corresponding formwork `S3` and `S4`.

  Excavations and foundations precede the corresponding formwork.

  Also, no more than 3 days can elapse between the end of a particular foundation (or excavation for the pillars with no foundation activity) and the beginning of the corresponding formwork.

- A period for the concrete to set, `ABk`, is required between the concrete foundation `Bk` and the corresponding masonry work `Mk`.

  Hence, for each pillar, `Sk` precedes `Bk`, `Bk` precedes `ABk`, `ABk` precedes `Mk`, and both `Mk` and `M(k+1)` precede `Tk`.

- The time between the completion of formwork `Sk` and the completion of the corresponding concrete foundation `Bk` is at most 4 days.

- The delivery of the preformed bearers (an activity denoted `L`) occurs exactly 30 days after the beginning of the construction project; it lasts two days. Positioning activities `Tk` cannot start before the end of the delivery activity `L`.
- On both sides of the bridge, filling activities `V1` and `V2` are required so that the bridge can be properly connected to the road.

  `V1` and `V2` cannot start before the end of the corresponding bearer-positioning activities `T1` and `T5`.
- Temporary housing is needed for the construction workers. This entails both building (an activity denoted `UE`) and later disassembling (an activity denoted `UA`) temporary housing.

  The erection of the temporary housing `UE` must begin at least six days before the formwork `Sk`.

  The removal of the temporary housing `UA` can start at most two days before the end of the last masonry work `Mk`.
- The project end, `PE`, is an activity of null duration, which "executes" when the positioning activities `Tk`, the filling activities `Vk`, and the disassembling activity `UA` are completed.

The following table lists the activities to be scheduled, the duration in days for each, and the unary resource to be used (if any is necessary).

Table 14.1 *Bridge Building Activities*

Name	Activity Description	Duration	Resource
A1	Excavation	4	Excavator
A2	Excavation	2	Excavator
A3	Excavation	2	Excavator
A4	Excavation	2	Excavator
A5	Excavation	2	Excavator
A6	Excavation	5	Excavator
P1	Foundation piles	20	Pile driver
P2	Foundation piles	13	Pile driver
S1	Formwork	8	Carpentry
S2	Formwork	4	Carpentry
S3	Formwork	4	Carpentry
S4	Formwork	4	Carpentry
S5	Formwork	4	Carpentry
S6	Formwork	10	Carpentry
B1	Concrete foundation	1	Concrete mixer

Table 14.1 *Bridge Building Activities (Continued)*

Name	Activity Description	Duration	Resource
B2	Concrete foundation	1	Concrete mixer
B3	Concrete foundation	1	Concrete mixer
B4	Concrete foundation	1	Concrete mixer
B5	Concrete foundation	1	Concrete mixer
B6	Concrete foundation	1	Concrete mixer
AB1	Concrete setting time	1	
AB2	Concrete setting time	1	
AB3	Concrete setting time	1	
AB4	Concrete setting time	1	
AB5	Concrete setting time	1	
AB6	Concrete setting time	1	
M1	Masonry work	16	Bricklaying
M2	Masonry work	8	Bricklaying
M3	Masonry work	8	Bricklaying
M4	Masonry work	8	Bricklaying
M5	Masonry work	8	Bricklaying
M6	Masonry work	20	Bricklaying
T1	Positioning	12	Crane
T2	Positioning	12	Crane
T3	Positioning	12	Crane
T4	Positioning	12	Crane
T5	Positioning	12	Crane
V1	Filling	15	Caterpillar
V2	Filling	10	Caterpillar
L	Delivery of the preformed bearers	2	
UE	Erection of the temporary housing	10	
UA	Removal of the temporary housing	10	
PE	End of project	0	

Defining the Problem, Designing a Model

In our definition of the problem, we need to represent the origin and horizon of the schedule itself, the activities, the temporal constraints, the unary resource constraints, and finally the optimization criterion.

Schedule

No time origin nor horizon is specified in the problem. Since the problem statement refers to no absolute date, the *time origin* can arbitrarily be set to 0.

For the *time horizon*, the most general strategy would consist of determining an upper bound for the overall project duration. As this computation is complex in the case under consideration, we adopt another possibility: we arbitrarily fix the horizon to a number that is obviously high for this problem.

However, it may prove inefficient to adopt a horizon that is too long as it increases the domains of many constrained variables and consequently enlarges the search. In a complete application, we should start from a reasonable value for the horizon, and then increase that value if no solution can be found with it.

Here, to keep the example simple, we'll assume that 365 days (a year) is a value given by the user—and that we can stick to it.

```
IloModel
DefineModel(IloEnv env, IloNumVar& makespan)
{
  IloModel model(env);
  IloSchedulerEnv schedEnv(env);
  schedEnv.getResourceParam().setCapacityEnforcement(IloMediumHigh);
  IloInt horizon = 365;
  schedEnv.setHorizon(horizon);
```

Activities

The constructor for the class `IloActivity` and the member functions `startsAfterStart`, `startsAfterEnd`, and `endsAfterStart` let us represent the activities and temporal constraints of the problem. The following function, `MakeActivity`, will associate a name with each activity.

```
IloActivity
MakeActivity(IloModel model,
             const char* name,
             IloInt duration)
{
  IloEnv env = model.getEnv();
  IloActivity activity(env, duration, 0, name);
  IloNumExpr startVar = activity.getStartExpr();
  IloNumExpr endVar   = activity.getEndExpr();
```

```
  NameVariable(startVar, name, "start");
  NameVariable(endVar,   name, "end");
  return activity;
}
```

Temporal Constraints

Most of the temporal constraints are simple precedence constraints of the kind already encountered in Part I, *Getting Started with Scheduler*. The following statements, for example, specify that the filling activities, `V1` and `V2`, cannot start before the end of the corresponding activities, `T1` and `T5`, to position the bearer.

```
/* POSITIONING OF BEARERS PRECEDE FILLING. */
model.add(V1.startsAfterEnd(T1));
model.add(V2.startsAfterEnd(T5));
```

Some of the temporal constraints specify minimal and maximal *delays* between start and end times of activities. For example, the following statements specify that no more than -3 days can elapse between the end of a particular foundation (or excavation for the pillars with no foundation activity) and the beginning of the corresponding formwork.

```
model.add(A1.endsAfterStart(S1, -3));
model.add(A2.endsAfterStart(S2, -3));
model.add(P1.endsAfterStart(S3, -3));
model.add(P2.endsAfterStart(S4, -3));
model.add(A5.endsAfterStart(S5, -3));
model.add(A6.endsAfterStart(S6, -3));
```

The negative delay `A1.endsAfterStart(S1, -3);` can be read as "`A1` ends after the start of `S1` minus three units of time." This states that no more than 3 days can elapse between the end of `A1` and the beginning of `S1`.

Another particular constraint is the one that states that the delivery of the preformed bearers occurs exactly 30 days after the beginning of the project. As we can arbitrarily decide to use 0 as a beginning date for the start of the project, this constraint translates into setting the start time of activity `L` to 30. This is done by simply setting a lower bound for the start time variable of `L`, like this: `L.getStartVariable().setLb(30)`.

Resource Constraints

The `requires` member function of the class `IloActivity` can be used to post the resource requirement constraints. Each resource is an instance of the class `IloUnaryResource`.

The following `MakeResource` function receives these arguments: the `model` defining the problem, a `name` to be given to the resource, and an array of `activities` requiring the resource. The argument `numberOfActivities` indicates the size of that array.

```
void
MakeResource(IloModel model,
             const char* name,
```

```
            IloInt numberOfActivities,
            IloActivityArray activities)
{
  IloEnv env = model.getEnv();
  IloUnaryResource resource(env, name);
  for (IloInt i = 0; i < numberOfActivities; i++)
    model.add(activities[i].requires(resource));
}
```

This function is then called once for each resource of the problem. For example, the following code creates the `CATERPILLAR` resource and the resource requirement constraints for the filling activities.

```
  IloActivityArray V(env,2);
  V[0]=V1; V[1]=V2;
  MakeResource(model, “CATERPILLAR”, 2, V);
```

Optimization Criterion

Generally, the definition of a problem includes the definition of the optimization criterion (or of the multiple criteria) to consider. The bridge construction problem specifies a unique optimization criterion—the makespan of the project—equal to the end time of the activity `PE` (project end).

The function `DefineModel` takes a reference to the constrained `makespan` variable and sets it to the constrained variable representing the end time of activity `PE`.

The enforcement level `IloMediumHigh` indicates that the Scheduler will spend extra effort in enforcing the resource constraints.

```
IloModel
DefineModel(IloEnv env, IloNumVar& makespan)
{
  IloModel model(env);
  IloSchedulerEnv schedEnv(env);
  schedEnv.getResourceParam().setCapacityEnforcement(IloMediumHigh);
  IloInt horizon = 365;
  schedEnv.setHorizon(horizon);

  /* CREATE ACTIVITIES: PROJECT END. THE makespan POINTER IS SET TO
     THE END-TIME VARIABLE OF THE PROJECT END. */
  IloActivity PE = MakeActivity(model, “PE”, 0);
  makespan = IloNumVar(env, 0, IloInfinity, ILOINT);
  model.add(PE.endsAt(makespan));


  /* ... */
  /* RETURN THE CREATED MODEL. */
  return model;
```

Solving the Problem

The most basic kind of decision that can be made in the process of disjunctive scheduling is the ordering of two activities, to arbitrate a potential resource conflict. The scheduling process could consequently consist of a very simple loop:

1. Among those activities that are not yet ordered, select two activities that require a common resource.
2. Select the most promising order for the chosen activities. Post the corresponding precedence constraint. Keep the reverse ordering decision as an alternative to be tried upon backtracking.
3. Iterate steps 1 and 2 until all pairs of activities which require a common resource are ordered.

The main advantage of that algorithm is that the ordering decisions are made one by one. After each decision, the added precedence constraint is propagated, and at the next iteration the results of the propagation can be used to select the next two activities to order and the "most promising" order for these activities.

The main disadvantage of this algorithm is the number of decisions to make. If *n* activities require a given resource *R*, the number of decisions to make to order these activities is potentially *n(n-1)/2*. Depending on the appropriateness—in the context of a given application—of the heuristics available to select the "most promising" order, it may prove more efficient to make more global decisions, such as deciding which activity to execute first among *n* unordered activities that require a given resource. The following algorithm is based on making such "more global" decisions. Its main advantage is that the number of decisions to make to order *n* activities does not exceed *(n-1)*. It works like this:

1. Select a resource among the resources required by unordered activities.
2. Select the activity to execute first among the unordered activities that require the chosen resource. Post the corresponding precedence constraints. Keep the other activities as alternatives to be tried upon backtracking.
3. Iterate step 2 until all the activities that require the *chosen* resource have been put in order.
4. Iterate steps 1 to 3 until all the activities that require a *common* resource have been put in order.

The algorithm we describe here is implemented by the predefined Scheduler goal `IloRankForward`. At each iteration, a resource is chosen and the activities which require the chosen resource are put in order. For this ordering at each iteration, a resource constraint (linking an activity to the resource under consideration) is chosen. Two possibilities are considered:

- either the chosen activity is scheduled before the other unscheduled activities;
- or the chosen activity is removed from the candidates to be scheduled before the other unscheduled activities. It is constrained *not to execute first*.

Allowing the resources to be closed (i.e., not using the member function `IloResource::keepOpen`) is advantageous in this situation. Since the resources are closed, when only one activity is not constrained *not to execute first*, it is immediately selected to execute first. This choice could not be made if the resource were not closed, as it would always be possible to introduce a brand new activity and execute it before all the activities unscheduled as yet. Furthermore, when the decision is made that an activity will not be executed first, the earliest start time of that activity can be updated to become the minimal earliest end time of the not yet ordered activities. This also would be impossible if the resource were not closed, as we cannot determine this minimal earliest end time if not all activities are known.

Choosing the Resource to Schedule

Very often in constraint programming, the efficiency of the decision-making process depends on the appropriateness of the order in which decisions are made. Here, it is very important to make two selections appropriately: to select the resource to make decisions about, and to select the activity to schedule first with respect to this resource.

Scheduling problems are generally such that resources are not equally loaded. Over some periods of time, some resources are more relied upon than others. These resources are often called *critical*: their limited availability is a factor which prevents the reduction of project cost or duration. It is, in general, very important to schedule critical resources first, in order to optimize the use of these resources without being bound by the schedule of other resources.

A standard way to measure the criticality of a resource consists of comparing the *demand* for the resource to its availability (*supply*) over a specific period of time. Here, for example, the period under consideration may run from the soonest of the earliest start times of the activities to order, to the last of the latest end times of these activities. As the capacity of the resource is 1 at all times, the supply over this time interval is the length of the interval. The demand over the interval is the sum of the durations of the activities to execute.

Slack is defined as the difference between the supply and the demand. The most critical resource is the resource with the least slack. Only the activities that are not yet ranked need to be considered in the computation of demand and supply, because once an activity is put in order, there are no more decisions to make about it.

The resource selector `IloSelResMinGlobalSlack` selects the resource on which not all activities are ordered that has minimal slack.

Choosing the Activity to Schedule First

Earliest and latest start or end times generally constitute appropriate data for deciding which activity to schedule first. The value `IloSelFirstRCMinStartMax` of the enumerated type `IloResourceConstraintSelector` indicates that the unranked resource constraint corresponding to the activity with the minimal earliest start time is to be selected. When two or more activities have the same earliest start time, the activity with the minimal latest start time is chosen.

Minimizing the Makespan

The default version of `IloRankForward` uses `IloSelResMinGlobalSlack` to select the next resource and `IloSelFirstRCMinStartMax` to select the next resource constraint. Using the defaults with the variable `makespan` set as the objective to minimize will generate an optimal solution to the problem.

```
    IloNumVar makespan;
    IloModel model = DefineModel(env, makespan);
    model.add(IloMinimize(env, makespan));

    IloSolver solver(model);
    IloGoal goal = IloRankForward(env, makespan);

    if (solver.solve(goal)) {
      PrintSolution(solver);
      solver.printInformation();

#if defined(ILO_SDXLOUTPUT)
      IloSDXLOutput output(env);
      ofstream outFile("bridge.xml");
      output.write(IlcScheduler(solver), outFile);
      outFile.close();
#endif
    }
    else
      solver.out() << "No solution!" << endl;
```

Complete Program and Output

You can see the entire program `bridge.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

#if defined(ILO_SDXLOUTPUT)
#include "sdxloutput.h"
#endif
```

```
////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////
void NameVariable(IloNumExpr& var,
                  const char* actName, const char* varName) {
  char* s = new char [strlen(actName) + strlen(varName) + 1];
  sprintf(s,"%s%s", actName, varName);
  var.setName(s);
  delete [] s;
}

IloActivity
MakeActivity(IloModel model,
             const char* name,
             IloInt duration)
{
  IloEnv env = model.getEnv();
  IloActivity activity(env, duration, 0, name);
  IloNumExpr startVar = activity.getStartExpr();
  IloNumExpr endVar   = activity.getEndExpr();
  NameVariable(startVar, name, "start");
  NameVariable(endVar,   name, "end");
  return activity;
}
void
MakeResource(IloModel model,
             const char* name,
             IloInt numberOfActivities,
             IloActivityArray activities)
{
  IloEnv env = model.getEnv();
  IloUnaryResource resource(env, name);
  for (IloInt i = 0; i < numberOfActivities; i++)
    model.add(activities[i].requires(resource));
}
IloModel
DefineModel(IloEnv env, IloNumVar& makespan)
{
  IloModel model(env);
  IloSchedulerEnv schedEnv(env);
  schedEnv.getResourceParam().setCapacityEnforcement(IloMediumHigh);
  IloInt horizon = 365;
  schedEnv.setHorizon(horizon);

  /* CREATE ACTIVITIES AND RESOURCES: EXCAVATIONS. */
  IloActivity A1 = MakeActivity(model, "A1", 4);
  IloActivity A2 = MakeActivity(model, "A2", 2);
  IloActivity A3 = MakeActivity(model, "A3", 2);
  IloActivity A4 = MakeActivity(model, "A4", 2);
  IloActivity A5 = MakeActivity(model, "A5", 2);
  IloActivity A6 = MakeActivity(model, "A6", 5);
  IloActivityArray A(env,6);
  A[0]=A1; A[1]=A2; A[2]=A3; A[3]=A4; A[4]=A5; A[5]=A6;
  MakeResource(model, "EXCAVATOR", 6, A);
```

```
/* CREATE ACTIVITIES AND RESOURCES: FOUNDATIONS. */
IloActivity P1 = MakeActivity(model, “P1”, 20);
IloActivity P2 = MakeActivity(model, “P2”, 13);
IloActivityArray P(env,2);
P[0]=P1; P[1]=P2;
MakeResource(model, “PILE DRIVER”, 2, P);

/* CREATE ACTIVITIES AND RESOURCES: FORMWORKS. */
IloActivity S1 = MakeActivity(model, “S1”, 8);
IloActivity S2 = MakeActivity(model, “S2”, 4);
IloActivity S3 = MakeActivity(model, “S3”, 4);
IloActivity S4 = MakeActivity(model, “S4”, 4);
IloActivity S5 = MakeActivity(model, “S5”, 4);
IloActivity S6 = MakeActivity(model, “S6”, 10);
IloActivityArray S(env,6);
S[0]=S1; S[1]=S2; S[2]=S3; S[3]=S4; S[4]=S5; S[5]=S6;
MakeResource(model, “CARPENTRY”, 6, S);

/* CREATE ACTIVITIES AND RESOURCES: CONCRETE FOUNDATIONS. */
IloActivity B1 = MakeActivity(model, “B1”, 1);
IloActivity B2 = MakeActivity(model, “B2”, 1);
IloActivity B3 = MakeActivity(model, “B3”, 1);
IloActivity B4 = MakeActivity(model, “B4”, 1);
IloActivity B5 = MakeActivity(model, “B5”, 1);
IloActivity B6 = MakeActivity(model, “B6”, 1);
IloActivityArray B(env,6);
B[0]=B1; B[1]=B2; B[2]=B3; B[3]=B4; B[4]=B5; B[5]=B6;
MakeResource(model, “CONCRETE MIXER”, 6, B);

/* CREATE ACTIVITIES AND RESOURCES: CONCRETE SETTING TIMES. */
IloActivity AB1 = MakeActivity(model, “AB1”, 1);
IloActivity AB2 = MakeActivity(model, “AB2”, 1);
IloActivity AB3 = MakeActivity(model, “AB3”, 1);
IloActivity AB4 = MakeActivity(model, “AB4”, 1);
IloActivity AB5 = MakeActivity(model, “AB5”, 1);
IloActivity AB6 = MakeActivity(model, “AB6”, 1);
IloActivityArray AB(env,6);
AB[0]=AB1; AB[1]=AB2; AB[2]=AB3; AB[3]=AB4; AB[4]=AB5; AB[5]=AB6;

/* CREATE ACTIVITIES AND RESOURCES: MASONRY. */
IloActivity M1 = MakeActivity(model, “M1”, 16);
IloActivity M2 = MakeActivity(model, “M2”, 8);
IloActivity M3 = MakeActivity(model, “M3”, 8);
IloActivity M4 = MakeActivity(model, “M4”, 8);
IloActivity M5 = MakeActivity(model, “M5”, 8);
IloActivity M6 = MakeActivity(model, “M6”, 20);
IloActivityArray M(env,6);
M[0]=M1; M[1]=M2; M[2]=M3; M[3]=M4; M[4]=M5; M[5]=M6;
MakeResource(model, “BRICKLAYING”, 6, M);

/* CREATE ACTIVITIES: POSITIONING. */
IloActivity T1 = MakeActivity(model, “T1”, 12);
IloActivity T2 = MakeActivity(model, “T2”, 12);
IloActivity T3 = MakeActivity(model, “T3”, 12);
IloActivity T4 = MakeActivity(model, “T4”, 12);
IloActivity T5 = MakeActivity(model, “T5”, 12);
IloActivityArray T(env,5);
T[0]=T1; T[1]=T2; T[2]=T3; T[3]=T4; T[4]=T5;
```

```
MakeResource(model, “CRANE”, 5, T);

/* CREATE ACTIVITIES: FILLING. */
IloActivity V1 = MakeActivity(model, “V1”, 15);
IloActivity V2 = MakeActivity(model, “V2”, 10);
IloActivityArray V(env,2);
V[0]=V1; V[1]=V2;
MakeResource(model, “CATERPILLAR”, 2, V);

/* CREATE ACTIVITIES: DELIVERY OF THE PREFORMED BEARERS. */
IloActivity L = MakeActivity(model, “L”,  2);

/* CREATE ACTIVITIES: REMOVAL OF THE TEMPORARY HOUSINGS. */
IloActivity UE = MakeActivity(model, “UE”, 10);
IloActivity UA = MakeActivity(model, “UA”, 10);
/* CREATE ACTIVITIES: PROJECT END. THE makespan POINTER IS SET TO
   THE END-TIME VARIABLE OF THE PROJECT END. */
IloActivity PE = MakeActivity(model, “PE”, 0);
makespan = IloNumVar(env, 0, IloInfinity, ILOINT);
model.add(PE.endsAt(makespan));


/* DELIVERY OF THE PREFORMED BEARERS OCCURS EXACTLY 30 DAYS AFTER
   THE BEGINNING OF THE PROJECT. */
L.setStartMin(30);

IloInt k;
for(k = 0; k < 5; k++) {
  /* POSITIONING STARTS AFTER THE DELIVERY OF THE PREFORMED BEARERS. */
  model.add(T[k].startsAfterEnd(L));
  /* MASONRY WORKS M[k] AND M[k + 1] PRECEDE POSITIONING T[k]. */
  model.add(T[k].startsAfterEnd(M[k]));
  model.add(T[k].startsAfterEnd(M[k + 1]));
}

for(k = 0; k < 6; k++) {
  /* FORMWORKS Sk PRECEDE CONCRETE FOUNDATIONS Bk. */
  model.add(B[k].startsAfterEnd(S[k]));
  /* CONCRETE FOUNDATIONS Bk PRECEDE CONCRETE SETTING TIMES ABk. */
  model.add(AB[k].startsAfterEnd(B[k]));
  /* CONCRETE SETTING TIMES ABk PRECEDE MASONRIES Mk. */
  model.add(M[k].startsAfterEnd(AB[k]));
  /* THE TIME BETWEEN THE COMPLETION OF A FORMWORK Sk AND THE
     COMPLETION OF ITS CORRESPONDING CONCRETE FOUNDATION Bk IS AT
     MOST 4 DAYS. */
  model.add(S[k].endsAfterEnd(B[k], -4));
  /* FORMWORKS Sk MUST BEGIN AT LEAST SIX DAYS AFTER THE BEGINNING
     OF ERECTION OF TEMPORARY HOUSING UE. */
  model.add(S[k].startsAfterStart(UE, 6));
  /* THE REMOVAL OF THE TEMPORARY HOUSING UA CAN START TWO DAYS
     BEFORE THE END OF THE LAST MASONRY WORK. */
  model.add(UA.startsAfterEnd(M[k], -2));
}

/* EXCAVATIONS PRECEDE FOUNDATIONS. */
model.add(P1.startsAfterEnd(A3));
model.add(P2.startsAfterEnd(A4));
```

```
  /* EXCAVATIONS AND FOUNDATIONS PRECEDE FORMWORKS. */
  model.add(S1.startsAfterEnd(A1));
  model.add(S2.startsAfterEnd(A2));
  model.add(S3.startsAfterEnd(P1));
  model.add(S4.startsAfterEnd(P2));
  model.add(S5.startsAfterEnd(A5));
  model.add(S6.startsAfterEnd(A6));
  /* POSITIONING OF BEARERS PRECEDE FILLING. */
  model.add(V1.startsAfterEnd(T1));
  model.add(V2.startsAfterEnd(T5));

  /* THERE ARE AT MOST THREE DAYS BEFORE THE END OF A PARTICULAR
     EXCAVATION (OR FOUNDATION PILES) AND THE BEGINNING OF THE
     CORRESPONDING FORMWORK. */
  model.add(A1.endsAfterStart(S1, -3));
  model.add(A2.endsAfterStart(S2, -3));
  model.add(P1.endsAfterStart(S3, -3));
  model.add(P2.endsAfterStart(S4, -3));
  model.add(A5.endsAfterStart(S5, -3));
  model.add(A6.endsAfterStart(S6, -3));

  /* PROJECT END. */
  model.add(PE.startsAfterEnd(V1));
  model.add(PE.startsAfterEnd(T2));
  model.add(PE.startsAfterEnd(T3));
  model.add(PE.startsAfterEnd(T4));
  model.add(PE.startsAfterEnd(V2));
  model.add(PE.startsAfterEnd(UA));

  /* RETURN THE CREATED MODEL. */
  return model;
}

////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////////////////

void
PrintSolution(const IloSolver& solver)
{
  IlcScheduler scheduler(solver);
  IloEnv env = solver.getEnv();
  for(IloIterator<IloActivity> ite(env); ite.ok(); ++ite)
    solver.out() << scheduler.getActivity(*ite) << endl;
}

////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////

int main()
{
  try {
    IloEnv env;
```

```
    // Model
    IloNumVar makespan;
    IloModel model = DefineModel(env, makespan);
    model.add(IloMinimize(env, makespan));

    IloSolver solver(model);
    IloGoal goal = IloRankForward(env, makespan);

    if (solver.solve(goal)) {
      PrintSolution(solver);
      solver.printInformation();

#if defined(ILO_SDXLOUTPUT)
      IloSDXLOutput output(env);
      ofstream outFile("bridge.xml");
      output.write(IlcScheduler(solver), outFile);
      outFile.close();
#endif
    }
    else
      solver.out() << "No solution!" << endl;


    env.end();

  } catch (IloException& exc) {
    cout << exc << endl;
  }

  return 0;
}

////////////////////////////////////////////////////////////////////////////
//
// RESULTS
//
////////////////////////////////////////////////////////////////////////////

/*
  A1 [4..6 -- 4 --> 8..10]
  A2 [2..4 -- 2 --> 4..6]
  A3 [8..24 -- 2 --> 10..26]
  A4 [0..2 -- 2 --> 2..4]
  A5 [17..28 -- 2 --> 19..30]
  A6 [19..31 -- 5 --> 24..36]
  P1 [15..26 -- 20 --> 35..46]
  P2 [2..13 -- 13 --> 15..26]
  S1 [10 -- 8 --> 18]
  S2 [6 -- 4 --> 10]
  S3 [36..46 -- 4 --> 40..50]
  S4 [18..26 -- 4 --> 22..30]
  S5 [22..30 -- 4 --> 26..34]
  S6 [26..36 -- 10 --> 36..46]
  B1 [18 -- 1 --> 19]
  B2 [10 -- 1 --> 11]
  B3 [40..50 -- 1 --> 41..51]
  B4 [22..33 -- 1 --> 23..34]
```

```
  B5 [26..34 -- 1 --> 27..35]
  B6 [36..49 -- 1 --> 37..50]
  AB1 [19 -- 1 --> 20]
  AB2 [11 -- 1 --> 12]
  AB3 [41..51 -- 1 --> 42..52]
  AB4 [23..43 -- 1 --> 24..44]
  AB5 [27..35 -- 1 --> 28..36]
  AB6 [37..59 -- 1 --> 38..60]
  M1 [20 -- 16 --> 36]
  M2 [12 -- 8 --> 20]
  M3 [52 -- 8 --> 60]
  M4 [44 -- 8 --> 52]
  M5 [36 -- 8 --> 44]
  M6 [60 -- 20 --> 80]
  T1 [36..44 -- 12 --> 48..56]
  T2 [92 -- 12 --> 104]
  T3 [64..68 -- 12 --> 76..80]
  T4 [52..56 -- 12 --> 64..68]
  T5 [80 -- 12 --> 92]
  V1 [48..79 -- 15 --> 63..94]
  V2 [92..94 -- 10 --> 102..104]
  L [30..42 -- 2 --> 32..44]
  UE [0 -- 10 --> 10]
  UA [78..94 -- 10 --> 88..104]
  PE [104 -- 0 --> 104]
  Number of fails               : 83
  Number of choice points       : 102
  Number of variables           : 129
  Number of constraints         : 170
  Reversible stack (bytes)      : 20124
  Solver heap (bytes)           : 56304
  Solver global heap (bytes)    : 112696
  And stack (bytes)             : 4044
  Or stack (bytes)              : 4044
  Search Stack (bytes)          : 4044
  Constraint queue (bytes)      : 11144
  Total memory used (bytes)     : 212400
  Elapsed time since creation   : 0.681
*/
```

The output shows each activity, identified by its name, followed by information about its place in the schedule. This information is enclosed in square brackets. It consists of three items: start time, duration, and end time of the activity. Each item can be represented either as a single value or as an interval. If the item is represented as an interval, it appears as two integers separated by two dots.

For example, activity `A1` can begin during the interval day 4 to day 6; it lasts four days, and terminates in the interval day 8 to day 10.

C H A P T E R

15

Scheduling with Unary Resources: the Job-Shop Problem

Some scheduling problems are very difficult to find an optimal solution for, even if the problem is of a small size. The aim of this chapter (and of Chapter 16 and Chapter 30) is to explore various problem-solving methods that address such difficult scheduling problems. The examples are based on well-known problems from operations research about scheduling operations in a job-shop. Because these problems have proved so intractable in the past, we offer different approaches to them here with the aim of showing how to achieve satisfactory results for other intractable problems in a realistic industrial setting.

Describing the Problem presents three job-shop scheduling problems known as MT06, MT10, and MT20. They were first proposed in 1963 in the book *Industrial Scheduling*. The second of these problems remained unsolved for nearly 25 years and is still considered hard. Very few scheduling specialists have been able to solve it efficiently or satisfactorily.

Defining the Problem, Designing a Model develops the code needed to define these problems.

A Minimizing Algorithm applies the algorithm given in Chapter 14 to these problems. We observe that to get an optimal solution for MT10 we have to use a very high number of optimizing iterations, and the final iterations require a great amount of computational time. Similarly, MT20 remains unsolved after a large amount of computational time.

Another algorithm is developed in Chapter 16 to facilitate faster convergence toward an optimal solution. The idea is to converge to the optimal makespan by applying a dichotomic

search. A specific search goal provides a good upper bound of the optimal makespan for this dichotomic search. That example also introduces a more efficient heuristic. The approach in Chapter 16 allows the three job-shop problems to be solved in a reasonable amount of computational time.

The job-shop problems are also discussed in Chapter 30. That chapter presents yet another kind of algorithm, one not guaranteed to generate an optimal solution, but one that tends to provide "good" solutions. Such an approach is often useful when solving complex and realistic problems.

The study of these different approaches strongly suggests that a combination of algorithms may, on the average, produce better results than any algorithm by itself.

One of the main advantages of constraint programming is precisely that such a combination is easy to implement. In the context of an industrial application, the right way to proceed is to implement a first version of the application, using a unique algorithm, typically guaranteed to stop after a given number of backtracks. Other algorithms can be designed, experimented with, and integrated in subsequent versions of the application, thereby achieving better and better results.

Describing the Problem

The job-shop scheduling problem consists of *n* jobs to be performed using *m* machines. Each job is described as a list of *m* activities of given processing times, to be executed in a given order. Each activity requires a specified machine and each machine is required by a unique activity of each job. The goal is to find a schedule with a minimal makespan, that is, a schedule for which the latest completion time for all the activities is minimal.

Three problem instances are considered:

- MT06 consists of 6 jobs and 6 machines.
- MT10 consists of 10 jobs and 10 machines.
- MT20 consists of 20 jobs and 5 machines.

The following arrays provide the data defining the problems. At the intersection of the i^{th} row and j^{th} column, each `ResourceNumbers**` array provides the number of the machine used to perform the j^{th} step of the i^{th} job. At the intersection of the i^{th} row and j^{th} column, each `Durations**` array indicates the processing time of the j^{th} step of the i^{th} job.

```
IloInt ResourceNumbers06 [] = {2, 0, 1, 3, 5, 4,
                               1, 2, 4, 5, 0, 3,
                               2, 3, 5, 0, 1, 4,
                               1, 0, 2, 3, 4, 5,
                               2, 1, 4, 5, 0, 3,
                               1, 3, 5, 0, 4, 2};

IloInt Durations06 [] = { 1,  3,  6,  7,  3,  6,
```

```
                          8,  5, 10, 10, 10,  4,
                          5,  4,  8,  9,  1,  7,
                          5,  5,  5,  3,  8,  9,
                          9,  3,  5,  4,  3,  1,
                          3,  3,  9, 10,  4,  1};

IloInt ResourceNumbers10 [] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
                               0, 2, 4, 9, 3, 1, 6, 5, 7, 8,
                               1, 0, 3, 2, 8, 5, 7, 6, 9, 4,
                               1, 2, 0, 4, 6, 8, 7, 3, 9, 5,
                               2, 0, 1, 5, 3, 4, 8, 7, 9, 6,
                               2, 1, 5, 3, 8, 9, 0, 6, 4, 7,
                               1, 0, 3, 2, 6, 5, 9, 8, 7, 4,
                               2, 0, 1, 5, 4, 6, 8, 9, 7, 3,
                               0, 1, 3, 5, 2, 9, 6, 7, 4, 8,
                               1, 0, 2, 6, 8, 9, 5, 3, 4, 7};

IloInt Durations10 [] = {29, 78,  9, 36, 49, 11, 62, 56, 44, 21,
                         43, 90, 75, 11, 69, 28, 46, 46, 72, 30,
                         91, 85, 39, 74, 90, 10, 12, 89, 45, 33,
                         81, 95, 71, 99,  9, 52, 85, 98, 22, 43,
                         14,  6, 22, 61, 26, 69, 21, 49, 72, 53,
                         84,  2, 52, 95, 48, 72, 47, 65,  6, 25,
                         46, 37, 61, 13, 32, 21, 32, 89, 30, 55,
                         31, 86, 46, 74, 32, 88, 19, 48, 36, 79,
                         76, 69, 76, 51, 85, 11, 40, 89, 26, 74,
                         85, 13, 61,  7, 64, 76, 47, 52, 90, 45};

IloInt ResourceNumbers20 [] = {0, 1, 2, 3, 4,
                               0, 1, 3, 2, 4,
                               1, 0, 2, 4, 3,
                               1, 0, 4, 2, 3,
                               2, 1, 0, 3, 4,
                               2, 1, 4, 0, 3,
                               1, 0, 2, 3, 4,
                               2, 1, 0, 3, 4,
                               0, 3, 2, 1, 4,
                               1, 2, 0, 3, 4,
                               1, 3, 0, 4, 2,
                               2, 0, 1, 3, 4,
                               0, 2, 1, 3, 4,
                               2, 0, 1, 3, 4,
                               0, 1, 4, 2, 3,
                               1, 0, 3, 4, 2,
                               0, 2, 1, 3, 4,
                               0, 1, 4, 2, 3,
                               1, 2, 0, 3, 4,
                               0, 1, 2, 3, 4};

IloInt Durations20 [] = {29,  9, 49, 62, 44,
                         43, 75, 69, 46, 72,
                         91, 39, 90, 12, 45,
                         81, 71,  9, 85, 22,
                         14, 22, 26, 21, 72,
                         84, 52, 48, 47,  6,
                         46, 61, 32, 32, 30,
                         31, 46, 32, 19, 36,
                         76, 76, 85, 40, 26,
```

```
                        85, 61, 64, 47, 90,
                        78, 36, 11, 56, 21,
                        90, 11, 28, 46, 30,
                        85, 74, 10, 89, 33,
                        95, 99, 52, 98, 43,
                         6, 61, 69, 49, 53,
                         2, 95, 72, 65, 25,
                        37, 13, 21, 89, 55,
                        86, 74, 88, 48, 79,
                        69, 51, 11, 89, 74,
                        13,  7, 76, 52, 45};
```

The `main` function accepts the number of the problem to be solved (6, 10, or 20) as its argument. Four parameters, `numberOfJobs`, `numberOfResources`, `resourceNumbers`, and `durations`, are set accordingly. Of course, in a typical industrial application, these data would have to be obtained from a database.

```
void
InitParameters(int argc,
               char** argv,
               IloInt& numberOfJobs,
               IloInt& numberOfResources,
               IloInt*& resourceNumbers,
               IloInt*& durations)
{
  if (argc > 1) {
    IloInt number = atol(argv[1]);
    if (number == 10) {
      numberOfJobs = 10;
      numberOfResources = 10;
      resourceNumbers = ResourceNumbers10;
      durations = Durations10;
    }
    else if (number == 20) {
      numberOfJobs = 20;
      numberOfResources = 5;
      resourceNumbers = ResourceNumbers20;
      durations = Durations20;
    }
  }
}

int main(int argc, char** argv)
{
  try {
    IloEnv env;

    IloInt numberOfJobs = 6;
    IloInt numberOfResources = 6;
    IloInt* resourceNumbers = ResourceNumbers06;
    IloInt* durations = Durations06;
    InitParameters(argc,
                   argv,
                   numberOfJobs,
                   numberOfResources,
                   resourceNumbers,
                   durations);
```

```
  /* ... */
}
```

These parameters are then passed to the function `DefineModel`; it returns an instance of the class `IloModel` that corresponds to the problem and sets the constrained `makespan` variable to be minimized.

```
    IloNumVar makespan;
    IloAnyArray jobs;
    IloModel model = DefineModel(env,
                                 numberOfJobs,
                                 numberOfResources,
                                 resourceNumbers,
                                 durations,
                                 makespan,
                                 jobs);
    model.add(IloMinimize(env, makespan));

  /* ... */
```

The function `DefineModel` is developed in the next section.

Defining the Problem, Designing a Model

As we define the problem, we'll need to represent the origin and horizon of the schedule as well as its resources and activities.

Schedule

No time origin nor horizon is specified in the problem. Since the problem statement refers to no absolute date, the *time origin* can arbitrarily be set to 0. The *time horizon* can be set to the sum of the durations of the activities, since performing all activities one after the other (in an order compatible with the order imposed by each job) results in a solution to the problem. Similarly, the constrained `makespan` variable can be created with 0 as its minimal value and the sum of the durations of the activities as its maximal value.

```
IloModel
DefineModel(const IloEnv& env,
            IloInt numberOfJobs,
            IloInt numberOfResources,
            IloInt* resourceNumbers,
            IloInt* durations,
            IloNumVar& makespan,
            IloAnyArray& jobs)
{
  IloModel model(env);

  /* CREATE THE MAKESPAN VARIABLE. */
```

```
  IloInt numberOfActivities = numberOfJobs * numberOfResources;
  IloInt horizon = 0;
  IloInt k;
  for (k = 0; k < numberOfActivities; k++)
    horizon += durations[k];

  makespan = IloNumVar(env, 0, horizon, ILOINT);

  /* ... */
  /* RETURN THE MODEL. */
  delete [] resources;
  return model;
}
```

Resources and Activities

The following code creates the resources, creates the activities, posts the precedence constraints, posts the resource constraints, and constrains `makespan` to be at least the end time of the last activity of each job.

An instance of the class `IloUnaryResource` is used to represent each machine.

The capacity enforcement level of the resource parameter is set to `IloMediumHigh`, stating that extra effort should be spent in enforcing the capacity constraints on each resource.

We'll give a name to each activity. The format of this name is `JiSjRr`, to express the idea that the activity corresponding to the `j`th step of the `i`th job uses resource `r`.

```
  /* CREATE THE RESOURCES. */
  IloSchedulerEnv schedEnv(env);
  schedEnv.getResourceParam().setCapacityEnforcement(IloMediumHigh);

  IloInt j;
  IloUnaryResource *resources =
    new IloUnaryResource[numberOfResources];
  char buffer[128];
  for (j = 0; j < numberOfResources; j++) {
    sprintf(buffer, "R%ld", j);
    resources[j] = IloUnaryResource(env, buffer);
  }
  /* CREATE THE ACTIVITIES. */
  k = 0;
  IloInt i;
  jobs = IloAnyArray(env, numberOfJobs);
  for (i = 0; i < numberOfJobs; i++) {
    IlcInt j = i*numberOfResources;
    Job* job = new (env) Job(env, i, numberOfResources,
                             &durations[j],
                             &resourceNumbers[j],
                             resources);
    job->addToModel(model, makespan);
    jobs[i] = job;
  }
```

A Minimizing Algorithm

As we indicated at the beginning of this chapter, this kind of problem is normally so difficult that we are going to tackle it in several different ways. Here, we'll give a minimizing algorithm. In Chapter 16, we develop an algorithm that uses a dichotomizing binary search for `makespan`. In Chapter 30, we explain how to use a "randomized" algorithm.

Solving an instance of the job-shop scheduling problem is equivalent to sequencing the activities optimally for each resource. The following algorithm, already highlighted in Chapter 14, can be used to order the activities.

1. Select a resource among the resources required by unranked activities.
2. Select the activity to execute first among the unranked activities that require the chosen resource. Post the corresponding precedence constraints. Keep the other activities as alternatives to be tried upon backtracking.
3. Iterate step 2 until all the activities that require the *chosen* resource have been put in order.
4. Iterate steps 1 to 3 until all the activities that require a *common* resource have been put in order.

As in Chapter 14, the predefined goal `IloRankForward` implements this algorithm. Arguments of this goal are a resource selector and a resource constraint selector. Again as in Chapter 14, we use the resource selector `IloSelResMinGlobalSlack` and the resource constraint selector `IloSelFirstRCMinStartMax`. The resource selector selects the resource on which not all activities are ordered and which has minimal slack according to the member function `IlcUnaryResource::getGlobalSlack`.

The resource constraint selector selects the resource constraint corresponding to the unranked activity with the minimal *earliest* start time and, in addition, the activity with the minimal *latest* start time when two or more activities have the same earliest start time.

Using the predefined goal `IloRankForward`, we can obtain an optimal solution to the problem by simply setting `makespan` as the objective variable to minimize.

```
IloSolver solver(model);
IloGoal goal = IloRankForward(env,
                              makespan,
                              IloSelResMinGlobalSlack,
                              IloSelFirstRCMinStartMax);

if (solver.solve(goal))
  PrintSolution(solver, jobs, makespan);
else {
  solver.out() << " Failure for Makespan "
               << solver.getMax(makespan) << endl;
}
```

```
    solver.printInformation();
    env.end();

  } catch (IloException& exc) {
    cout << exc << endl;
  }
```

Complete Program and Output

You can see the entire program `jobshopm.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN
IloInt ResourceNumbers06 [] = {2, 0, 1, 3, 5, 4,
                               1, 2, 4, 5, 0, 3,
                               2, 3, 5, 0, 1, 4,
                               1, 0, 2, 3, 4, 5,
                               2, 1, 4, 5, 0, 3,
                               1, 3, 5, 0, 4, 2};

IloInt Durations06 [] = { 1,  3,  6,  7,  3,  6,
                          8,  5, 10, 10, 10,  4,
                          5,  4,  8,  9,  1,  7,
                          5,  5,  5,  3,  8,  9,
                          9,  3,  5,  4,  3,  1,
                          3,  3,  9, 10,  4,  1};

IloInt ResourceNumbers10 [] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
                               0, 2, 4, 9, 3, 1, 6, 5, 7, 8,
                               1, 0, 3, 2, 8, 5, 7, 6, 9, 4,
                               1, 2, 0, 4, 6, 8, 7, 3, 9, 5,
                               2, 0, 1, 5, 3, 4, 8, 7, 9, 6,
                               2, 1, 5, 3, 8, 9, 0, 6, 4, 7,
                               1, 0, 3, 2, 6, 5, 9, 8, 7, 4,
                               2, 0, 1, 5, 4, 6, 8, 9, 7, 3,
                               0, 1, 3, 5, 2, 9, 6, 7, 4, 8,
                               1, 0, 2, 6, 8, 9, 5, 3, 4, 7};

IloInt Durations10 [] = {29, 78,  9, 36, 49, 11, 62, 56, 44, 21,
                         43, 90, 75, 11, 69, 28, 46, 46, 72, 30,
                         91, 85, 39, 74, 90, 10, 12, 89, 45, 33,
                         81, 95, 71, 99,  9, 52, 85, 98, 22, 43,
                         14,  6, 22, 61, 26, 69, 21, 49, 72, 53,
                         84,  2, 52, 95, 48, 72, 47, 65,  6, 25,
                         46, 37, 61, 13, 32, 21, 32, 89, 30, 55,
                         31, 86, 46, 74, 32, 88, 19, 48, 36, 79,
                         76, 69, 76, 51, 85, 11, 40, 89, 26, 74,
                         85, 13, 61,  7, 64, 76, 47, 52, 90, 45};

IloInt ResourceNumbers20 [] = {0, 1, 2, 3, 4,
                               0, 1, 3, 2, 4,
                               1, 0, 2, 4, 3,
```

```
                                1, 0, 4, 2, 3,
                                2, 1, 0, 3, 4,
                                2, 1, 4, 0, 3,
                                1, 0, 2, 3, 4,
                                2, 1, 0, 3, 4,
                                0, 3, 2, 1, 4,
                                1, 2, 0, 3, 4,
                                1, 3, 0, 4, 2,
                                2, 0, 1, 3, 4,
                                0, 2, 1, 3, 4,
                                2, 0, 1, 3, 4,
                                0, 1, 4, 2, 3,
                                1, 0, 3, 4, 2,
                                0, 2, 1, 3, 4,
                                0, 1, 4, 2, 3,
                                1, 2, 0, 3, 4,
                                0, 1, 2, 3, 4};

IloInt Durations20 [] = {29,  9, 49, 62, 44,
                         43, 75, 69, 46, 72,
                         91, 39, 90, 12, 45,
                         81, 71,  9, 85, 22,
                         14, 22, 26, 21, 72,
                         84, 52, 48, 47,  6,
                         46, 61, 32, 32, 30,
                         31, 46, 32, 19, 36,
                         76, 76, 85, 40, 26,
                         85, 61, 64, 47, 90,
                         78, 36, 11, 56, 21,
                         90, 11, 28, 46, 30,
                         85, 74, 10, 89, 33,
                         95, 99, 52, 98, 43,
                          6, 61, 69, 49, 53,
                          2, 95, 72, 65, 25,
                         37, 13, 21, 89, 55,
                         86, 74, 88, 48, 79,
                         69, 51, 11, 89, 74,
                         13,  7, 76, 52, 45};
////////////////////////////////////////////////////////////////////////////
//
// DEFINITION OF THE JOB CLASS
//
////////////////////////////////////////////////////////////////////////////

class Job {
private:
  char* _name;
  IloInt _index;
  IloArray<IloActivity> _inShops;
  IloArray<IloUnaryResource> _machines;
public:
  Job(IloEnv env,
      IloInt index,
      IloInt size,
      IloInt* durations,
      IloInt* resourceNumbers,
      IloUnaryResource* resources);
  const char* getName() const { return _name;}
```

```
  IloInt getIndex() const { return _index;}
  IloInt getSize() const { return _inShops.getSize();}
  IloActivity getActivity(IloInt i) const {
    IloAssert(i >= 0, "bad index range");
    IloAssert(i < _inShops.getSize(), "bad index range");
    return _inShops[i];
  }
  IloInt getShop(IloActivity act) const;
  void addToModel(IloModel model, IloIntVar makespan);
  void display(ILOSTD(ostream)& out) const {
    out << "Job<" << _name
        << " ," << _index
        << ", " << _inShops.getSize() << ">";
  }
};
Job::Job(IloEnv env,
         IloInt index,
         IloInt size,
         IloInt* durations,
         IloInt* resourceNumbers,
         IloUnaryResource* resources)
  :_index(index)
{
  char buffer[128];
  sprintf(buffer, "J%ld", index);
  IlcInt len = strlen(buffer);
  _name =  new (env) char[len+1];
  strcpy(_name, buffer);
  IloArray<IloActivity> inShops(env, size);
  IloArray<IloUnaryResource> machines(env, size);
  IlcInt i;
  for(i = 0; i < size; i++) {
    sprintf(buffer, "J%ldS%ldR%ld", index, i, resourceNumbers[i]);
    IloActivity act(env, durations[i], buffer);
    act.setObject(this);
    inShops[i] = act;
    machines[i] = resources[resourceNumbers[i]];
  }
  _inShops = inShops;
  _machines = machines;
}

IloInt Job::getShop(IloActivity act) const {
  IloArray<IloActivity> inShops = _inShops;
  IloInt size = inShops.getSize();
  IloInt i;
  for(i = 0; i < size; i++) {
    if (inShops[i].getImpl() == act.getImpl())
      return i;
  }
  return -1;
}

void Job::addToModel(IloModel model, IloIntVar makespan) {
  IloArray<IloActivity> inShops = _inShops;
  IloInt size = inShops.getSize();
  IloInt i;
  for(i = 1; i < size; i++)
```

```
    model.add(inShops[i].startsAfterEnd(inShops[i-1]));
  model.add(inShops[size - 1].endsBefore(makespan));
  IloArray<IloUnaryResource> machines = _machines;
  for(i = 0; i < size; i++)
    model.add(inShops[i].requires(machines[i]));
}


////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////

#if defined(ILO_SDXLOUTPUT)
#include "sdxljssp.h"
#endif

IloModel
DefineModel(const IloEnv& env,
            IloInt numberOfJobs,
            IloInt numberOfResources,
            IloInt* resourceNumbers,
            IloInt* durations,
            IloNumVar& makespan,
            IloAnyArray& jobs)
{
  IloModel model(env);

  /* CREATE THE MAKESPAN VARIABLE. */
  IloInt numberOfActivities = numberOfJobs * numberOfResources;
  IloInt horizon = 0;
  IloInt k;
  for (k = 0; k < numberOfActivities; k++)
    horizon += durations[k];

  makespan = IloNumVar(env, 0, horizon, ILOINT);
  /* CREATE THE RESOURCES. */
  IloSchedulerEnv schedEnv(env);
  schedEnv.getResourceParam().setCapacityEnforcement(IloMediumHigh);

  IloInt j;
  IloUnaryResource *resources =
    new IloUnaryResource[numberOfResources];
  char buffer[128];
  for (j = 0; j < numberOfResources; j++) {
    sprintf(buffer, "R%ld", j);
    resources[j] = IloUnaryResource(env, buffer);
  }
  /* CREATE THE ACTIVITIES. */
  k = 0;
  IloInt i;
  jobs = IloAnyArray(env, numberOfJobs);
  for (i = 0; i < numberOfJobs; i++) {
    IlcInt j = i*numberOfResources;
    Job* job = new (env) Job(env, i, numberOfResources,
                             &durations[j],
                             &resourceNumbers[j],
```

```
                              resources);
    job->addToModel(model, makespan);
    jobs[i] = job;
  }
  /* RETURN THE MODEL. */
  delete [] resources;
  return model;
}
////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////////////////

void
PrintSolution(IloSolver solver,
              IloAnyArray jobs,
              IloIntVar makespan)
{
  solver.out() << " Solution for Makespan "
               << solver.getValue(makespan) << endl;
  IlcScheduler scheduler(solver);
  IloEnv env = solver.getEnv();
  IloInt numberOfJobs = jobs.getSize();
  IloInt i;
  for(i = 0; i < numberOfJobs; i++) {
    Job* job = (Job*) jobs[i];
    job->display(solver.out());
    solver.out() << endl;
    IloInt size = job->getSize();
    IloInt j;
    for(j = 0; j < size; j++) {
      IloActivity act = job->getActivity(j);
      solver.out() << "\t" << scheduler.getActivity(act) << endl;
    }
  }

#if defined(ILO_SDXLOUTPUT)
  IloSDXLJobShopOutput output(env);
  ofstream outFile("jobshopm.xml");
  output.writeJobShop(scheduler, outFile, jobs, makespan);
  outFile.close();
#endif
}

////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////
void
InitParameters(int argc,
               char** argv,
               IloInt& numberOfJobs,
               IloInt& numberOfResources,
               IloInt*& resourceNumbers,
               IloInt*& durations)
{
```

```
  if (argc > 1) {
    IloInt number = atol(argv[1]);
    if (number == 10) {
      numberOfJobs = 10;
      numberOfResources = 10;
      resourceNumbers = ResourceNumbers10;
      durations = Durations10;
    }
    else if (number == 20) {
      numberOfJobs = 20;
      numberOfResources = 5;
      resourceNumbers = ResourceNumbers20;
      durations = Durations20;
    }
  }
}

int main(int argc, char** argv)
{
  try {
    IloEnv env;

    IloInt numberOfJobs = 6;
    IloInt numberOfResources = 6;
    IloInt* resourceNumbers = ResourceNumbers06;
    IloInt* durations = Durations06;
    InitParameters(argc,
                   argv,
                   numberOfJobs,
                   numberOfResources,
                   resourceNumbers,
                   durations);
    IloNumVar makespan;
    IloAnyArray jobs;
    IloModel model = DefineModel(env,
                                 numberOfJobs,
                                 numberOfResources,
                                 resourceNumbers,
                                 durations,
                                 makespan,
                                 jobs);
    model.add(IloMinimize(env, makespan));

    IloSolver solver(model);
    IloGoal goal = IloRankForward(env,
                                  makespan,
                                  IloSelResMinGlobalSlack,
                                  IloSelFirstRCMinStartMax);

    if (solver.solve(goal))
      PrintSolution(solver, jobs, makespan);
    else {
      solver.out() << " Failure for Makespan "
                   << solver.getMax(makespan) << endl;
    }

    solver.printInformation();
    env.end();
```

```
  } catch (IloException& exc) {
    cout << exc << endl;
  }
  return 0;
}

////////////////////////////////////////////////////////////////////////////
//
// RESULTS
//
////////////////////////////////////////////////////////////////////////////

/* jobshopm 6
   Solution for Makespan 55
   J0S0R2 [5..7 -- 1 --> 6..8]
   J0S1R0 [6..11 -- 3 --> 9..14]
   J0S2R1 [16 -- 6 --> 22]
   J0S3R3 [30..36 -- 7 --> 37..43]
   J0S4R5 [42..43 -- 3 --> 45..46]
   J0S5R4 [49 -- 6 --> 55]
   J1S0R1 [0 -- 8 --> 8]
   J1S1R2 [8 -- 5 --> 13]
   J1S2R4 [13..15 -- 10 --> 23..25]
   J1S3R5 [28..29 -- 10 --> 38..39]
   J1S4R0 [38..40 -- 10 --> 48..50]
   J1S5R3 [48..50 -- 4 --> 52..54]
   J2S0R2 [0..2 -- 5 --> 5..7]
   J2S1R3 [5..7 -- 4 --> 9..11]
   J2S2R5 [9..11 -- 8 --> 17..19]
   J2S3R0 [18..19 -- 9 --> 27..28]
   J2S4R1 [27..41 -- 1 --> 28..42]
   J2S5R4 [42 -- 7 --> 49]
   J3S0R1 [8 -- 5 --> 13]
   J3S1R0 [13..14 -- 5 --> 18..19]
   J3S2R2 [22 -- 5 --> 27]
   J3S3R3 [27 -- 3 --> 30]
   J3S4R4 [30 -- 8 --> 38]
   J3S5R5 [45..46 -- 9 --> 54..55]
   J4S0R2 [13 -- 9 --> 22]
   J4S1R1 [22 -- 3 --> 25]
   J4S2R4 [25 -- 5 --> 30]
   J4S3R5 [38..39 -- 4 --> 42..43]
   J4S4R0 [48..51 -- 3 --> 51..54]
   J4S5R3 [52..54 -- 1 --> 53..55]
   J5S0R1 [13 -- 3 --> 16]
   J5S1R3 [16 -- 3 --> 19]
   J5S2R5 [19 -- 9 --> 28]
   J5S3R0 [28 -- 10 --> 38]
   J5S4R4 [38 -- 4 --> 42]
   J5S5R2 [42..54 -- 1 --> 43..55]
*/

/* jobshopm 10
   Solution for Makespan 930
   J0S0R0 [119 -- 29 --> 148]
   J0S1R1 [445..448 -- 78 --> 523..526]
   J0S2R2 [523..526 -- 9 --> 532..535]
```

```
J0S3R3 [532..540 -- 36 --> 568..576]
J0S4R4 [568..595 -- 49 --> 617..644]
J0S5R5 [640..644 -- 11 --> 651..655]
J0S6R6 [651..655 -- 62 --> 713..717]
J0S7R7 [721 -- 56 --> 777]
J0S8R8 [792..856 -- 44 --> 836..900]
J0S9R9 [893..909 -- 21 --> 914..930]
J1S0R0 [76 -- 43 --> 119]
J1S1R2 [224..244 -- 90 --> 314..334]
J1S2R4 [355 -- 75 --> 430]
J1S3R9 [430..467 -- 11 --> 441..478]
J1S4R3 [568..576 -- 69 --> 637..645]
J1S5R1 [637..689 -- 28 --> 665..717]
J1S6R6 [713..717 -- 46 --> 759..763]
J1S7R5 [759..767 -- 46 --> 805..813]
J1S8R7 [813 -- 72 --> 885]
J1S9R8 [885..900 -- 30 --> 915..930]
J2S0R1 [308..311 -- 91 --> 399..402]
J2S1R0 [408..411 -- 85 --> 493..496]
J2S2R3 [493..496 -- 39 --> 532..535]
J2S3R2 [532..535 -- 74 --> 606..609]
J2S4R8 [609 -- 90 --> 699]
J2S5R5 [699 -- 10 --> 709]
J2S6R7 [709 -- 12 --> 721]
J2S7R6 [759..763 -- 89 --> 848..852]
J2S8R9 [848..852 -- 45 --> 893..897]
J2S9R4 [893..897 -- 33 --> 926..930]
J3S0R1 [0..6 -- 81 --> 81..87]
J3S1R2 [84..90 -- 95 --> 179..185]
J3S2R0 [185 -- 71 --> 256]
J3S3R4 [256 -- 99 --> 355]
J3S4R6 [355..370 -- 9 --> 364..379]
J3S5R8 [364..379 -- 52 --> 416..431]
J3S6R7 [416..431 -- 85 --> 501..516]
J3S7R3 [637..645 -- 98 --> 735..743]
J3S8R9 [766..830 -- 22 --> 788..852]
J3S9R5 [805..887 -- 43 --> 848..930]
J4S0R2 [210..230 -- 14 --> 224..244]
J4S1R0 [269..272 -- 6 --> 275..278]
J4S2R1 [286..289 -- 22 --> 308..311]
J4S3R5 [308..313 -- 61 --> 369..374]
J4S4R3 [370..404 -- 26 --> 396..430]
J4S5R4 [430 -- 69 --> 499]
J4S6R8 [499 -- 21 --> 520]
J4S7R7 [530..541 -- 49 --> 579..590]
J4S8R9 [593..657 -- 72 --> 665..729]
J4S9R6 [848..877 -- 53 --> 901..930]
J5S0R2 [0..3 -- 84 --> 84..87]
J5S1R1 [84..87 -- 2 --> 86..89]
J5S2R5 [86..90 -- 52 --> 138..142]
J5S3R3 [138..142 -- 95 --> 233..237]
J5S4R8 [233..244 -- 48 --> 281..292]
J5S5R9 [281..292 -- 72 --> 353..364]
J5S6R0 [361..364 -- 47 --> 408..411]
J5S7R6 [420..445 -- 65 --> 485..510]
J5S8R4 [499..510 -- 6 --> 505..516]
J5S9R7 [505..516 -- 25 --> 530..541]
J6S0R1 [86..89 -- 46 --> 132..135]
```

```
   J6S1R0 [148 -- 37 --> 185]
   J6S2R3 [233..237 -- 61 --> 294..298]
   J6S3R2 [375..395 -- 13 --> 388..408]
   J6S4R6 [388..408 -- 32 --> 420..440]
   J6S5R5 [421..440 -- 21 --> 442..461]
   J6S6R9 [442..478 -- 32 --> 474..510]
   J6S7R8 [520 -- 89 --> 609]
   J6S8R7 [668..679 -- 30 --> 698..709]
   J6S9R4 [698..740 -- 55 --> 753..795]
   J7S0R2 [179..199 -- 31 --> 210..230]
   J7S1R0 [275..278 -- 86 --> 361..364]
   J7S2R1 [399..402 -- 46 --> 445..448]
   J7S3R5 [445..461 -- 74 --> 519..535]
   J7S4R4 [519..535 -- 32 --> 551..567]
   J7S5R6 [557..567 -- 88 --> 645..655]
   J7S6R8 [699..710 -- 19 --> 718..729]
   J7S7R9 [718..729 -- 48 --> 766..777]
   J7S8R7 [777 -- 36 --> 813]
   J7S9R3 [813..851 -- 79 --> 892..930]
   J8S0R0 [0 -- 76 --> 76]
   J8S1R1 [217..220 -- 69 --> 286..289]
   J8S2R3 [294..298 -- 76 --> 370..374]
   J8S3R5 [370..374 -- 51 --> 421..425]
   J8S4R2 [421..425 -- 85 --> 506..510]
   J8S5R9 [506..510 -- 11 --> 517..521]
   J8S6R6 [517..527 -- 40 --> 557..567]
   J8S7R7 [579..590 -- 89 --> 668..679]
   J8S8R4 [668..714 -- 26 --> 694..740]
   J8S9R8 [718..782 -- 74 --> 792..856]
   J9S0R1 [132..135 -- 85 --> 217..220]
   J9S1R0 [256..259 -- 13 --> 269..272]
   J9S2R2 [314..334 -- 61 --> 375..395]
   J9S3R6 [375..401 -- 7 --> 382..408]
   J9S4R8 [416..435 -- 64 --> 480..499]
   J9S5R9 [517..521 -- 76 --> 593..597]
   J9S6R5 [593..597 -- 47 --> 640..644]
   J9S7R3 [735..743 -- 52 --> 787..795]
   J9S8R4 [787..795 -- 90 --> 877..885]
   J9S9R7 [885 -- 45 --> 930]
*/
```

Computational Remarks: Where to Go from Here

This program does not prove to be very efficient for solving the job-shop scheduling problem. Only MT06 is solved in a reasonable amount of time. Many optimizing iterations are necessary to solve MT10, with the final iterations requiring a great deal of computational time. MT20 remains unsolved after much computational time.

These observations suggest three ways of improving the algorithm.

1. Attempt to reduce the number of iterations.

2. Attempt to simplify the most difficult iterations.

3. Attempt to generate good solutions quickly, rather than seeking the optimal solution.

Chapter 16 discusses points 1 and 2 and proposes a better algorithm for the job-shop scheduling problem. Point 3 is covered in Chapter 30 and offers a third algorithm, which generates an optimal solution for MT20.

C H A P T E R

16

A Dichotomizing Binary-Search Algorithm: the Job-Shop Problem

In this chapter, we look again at the job-shop problems MT06, MT10, and MT20. In Chapter 15, *Scheduling with Unary Resources: the Job-Shop Problem*, we tried an algorithm that finds an optimal solution, using the makespan of the schedule as the cost function to optimize, but found that for some instances of the problem, the number of optimizing iterations was so great and the calculations so difficult as to make the search for an optimal solution impractical. In this chapter we look at ways to reduce the number of optimizing iterations and to make the remaining iterations easier to compute.

This time, rather than trying to systematically reduce the cost (the makespan) by at least one unit at each iteration, we'll use a binary search for the possible values of the constrained `makespan` variable. The binary search strictly dichotomizes the interval where the makespan might be found and applies appropriate constraints.

A first resolution of the problem, with a goal that finds a good first solution, allows us to start with a short interval for the possible values of the makespan. This tactic helps reduce the number of iterations needed to solve the problem.

Although the two ideas above (dichotomic search, good first solution) result in a small number of iterations, the remaining iterations are hard because we are close to the optimal value of the makespan. To simplify those "hard" iterations, we introduce a better heuristic, and then study the results of that improvement. Altogether, this new algorithm performs better than the previous one.

Setting Initial Interval Values for the Makespan Variable

The objective of the function `SetMakespanInitialBounds` is to find a smaller initial domain for the makespan variable.

The lower bound is easily computed by propagating the constraints of the problem, without executing any search goal. The makespan lower bound is then set with the Concert Technology function `IloNumVar::setLb`.

Finding a good value for the upper bound is more complex. We need to implement a goal, `SolveConflicts`, that generates a good first solution. This goal is executed and the makespan of the solution is taken as upper bound for the domain of the optimal makespan variable.

Here is the corresponding code.

```
void
SetMakespanInitialBounds(IloSolver& solver,
                         IloNumVar& makespan)
{
  IloGoal goal;
  /* SET MAKESPAN LOWER BOUND AND RESTART */
  goal = IloGoalTrue(solver.getEnv());
  solver.solve(goal);
  makespan.setLB(solver.getMin(makespan));

  /* SOLVE WITH GOAL SOLVECONFLICTS */
  goal = SolveConflicts(solver.getEnv());
  solver.solve(goal);

  /* SET MAKESPAN UPPER BOUND */
  makespan.setUB(solver.getMin(makespan));
  solver.out() << “Solution with makespan “ << makespan.getUB() << endl;
}
```

Now, let's look in detail at the goal `SolveConflicts`.

This goal supposes that each unary resource of the schedule is associated with a *precedence graph constraint*. (See the concept *Precedence Graph Constraint* in the *IBM ILOG Scheduler Reference Manual*.) The principle of this goal is to select, at each node of the search tree, a pair of resource constraints `(srct1, srct2)` on the same resource and to impose a precedence relation `srct1.setSuccessor(srct2)` between them. In case of a failure, the alternative choice `srct2.setSuccessor(srct1)` is tried. This is done with the predefined Scheduler goal `IlcTrySetSuccessor`.

Now, the critical point of the goal is the selection of the pair of resource constraints `(srct1,srct2)`. This selection is done by the function `SelectMostOpportunisticConflict`.

```
ILCGOAL0(SolveConflictsIlc) {
  IloSolver s = getSolver();
```

```
  IlcScheduler scheduler = IlcScheduler(s);

  IlcResourceConstraint srct1;
  IlcResourceConstraint srct2;
  if (SelectMostOpportunisticConflict(scheduler, srct1, srct2))
    return IlcAnd(IlcTrySetSuccessor(srct1, srct2), this);

  return 0;
}

ILOCPGOALWRAPPER0(SolveConflicts, solver) {
  return SolveConflictsIlc(solver);
}
```

If `{srct1,srct2}` is a pair of unranked resource constraints, we define a criterion `impact(srct1, srct2)` that approximates the impact on the schedule of the decision of ordering `srct1` before `srct2`. The impact of a decision on the schedule is a quantity that is proportional to the domain reduction of the start and end variables of activities due to the propagation of the decision.

Suppose that *act1* and *act2* are the activities of *srct1* and *srct2*, and that:

- *emin1* is the current minimal end time of *act1*,
- *emax1* is the current maximal end time of *act1*,
- *smin2* is the current minimal start time of *act2*, and
- *smax2* is the current maximal start time of *act2*.

When adding the precedence relation *srct1* < *srct2*:

- the reduction of the domain of the end time of *act1* will be at least *deltaEnd1 = emax1 - IlcMin(emax1, smax2).*
- the reduction of the domain of the start time of *act2* will be at least *deltaStart2 = IlcMax(emin1, smin2) - smin2.*

Thus, we estimate *impact(srct1,srct2) = deltaEnd1 + deltaStart2.* See Figure 16.1.

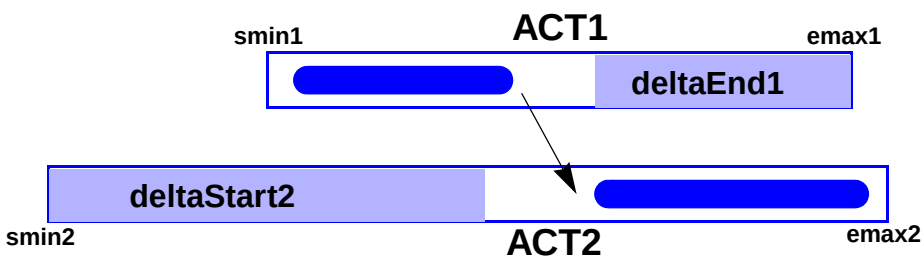


Figure 16.1 *Ordering srct1 before srct2*

If a choice exists between ordering `srct1` before `srct2` or `srct2` before `srct1`, we will always choose the order of minimal impact. Indeed, in such a way the domain reduction for the start and end variables of activities will be smaller. Therefore:

- it is hoped to stay as close as possible to the initial domain of these variables and obtain a small makespan, and
- as much room as possible is left for future decisions.

In all the solutions, all the pairs `{srct1,srct2}` will have to be ranked. As our goal is to minimize the impact of all the decisions taken until a solution is found, a good strategy is to select in priority those potential conflicts `{srct1, srct2}` for which the choice of the decision between `srct1.setSuccessor(srct2)` and `srct2.setSuccessor(srct1)` is the most evident; that is, the potential conflicts that maximize `IlcAbs(impact(srct1,srct2) - impact(srct2,srct1))`.

Suppose we have two pairs of unranked resource constraints `{srctA,srctB}` and `{srctC,srctD}` such that:

```
impact(srctA,srctB) = 10, impact(srctB,srctA) = 10; and
impact(srctC,srctD) = 20, impact(srctD,srctC) = 100
```

This means that we would first select the pair `{srctC,srctD}` and choose the decision `srctC.setSuccessor(srctD)`. After this decision has been propagated, it may be that it will be easier to choose between `srctA.setSuccessor(srctB)` and `srctB.setSuccessor(srctA)`.

Until this point, our criterion to select a potential conflict has been very local. A way to improve this is to prefer the potential conflicts that belong to a part of the schedule where there are still a lot of unresolved potential conflicts. This strategy has a double advantage.

1. The resolution of such conflicts may globally have less impact as they belong to a rather relaxed part of the schedule.
2. It ensures a more homogeneous repartition of the solved potential conflicts, so that the local criterion to measure the impact of a decision is more accurate.

The criterion used to estimate the potential conflict around a pair `(srct1, srct2)` is the minimum between the number of resource constraints that are unranked with respect to `srct1` (`nbUnranked(srct1)`), and the number of resource constraints that are unranked with respect to `srct2` (`nbUnranked(srct2)`).

Therefore the criterion:

```
IlcMin(nbUnranked(srct1), nbUnranked(srct2)) *
IlcAbs(impact(srct1,srct2) - impact(srct2,srct1))
```

is used as a measure of the opportunity of the potential conflict `{srct1,srct2}`.

The corresponding code follows. Note that a class `NbUnrankedExt` is attached as an object to the resource constraints in order to avoid the recomputation of the number of unranked resource constraints for each potential conflict `{srct1, srct2}`.

```
class NbUnrankedExt {
private:
  IlcInt _nbOfUnranked;

public:
  NbUnrankedExt() :_nbOfUnranked(0) {};
  ~NbUnrankedExt(){};
  void setValue(IlcInt nbOfUnranked) {
    _nbOfUnranked = nbOfUnranked;
  }
  IlcInt getValue() const {
    return _nbOfUnranked;
  }
};

IlcInt GetNumberOfUnranked(const IlcResourceConstraint& rct) {
  /* RETURN NUMBER OF UNRANKED W.R.T. RCT */
  IlcInt nb = 0;
  for (IlcResourceConstraintIterator ite(rct, IlcUnranked);
       ite.ok(); ++ite)
    nb++;
  return nb;
}

IlcInt GetOpportunity(const IlcScheduler& scheduler,
                      const IlcResourceConstraint& srct1,
                      const IlcResourceConstraint& srct2) {

  IlcActivity act1 = srct1.getActivity();
  IlcActivity act2 = srct2.getActivity();

  IlcInt smin1 = act1.getStartMin();
  IlcInt smax1 = act1.getStartMax();
  IlcInt emin1 = act1.getEndMin();
  IlcInt emax1 = act1.getEndMax();
  IlcInt smin2 = act2.getStartMin();
  IlcInt smax2 = act2.getStartMax();
  IlcInt emin2 = act2.getEndMin();
  IlcInt emax2 = act2.getEndMax();

  /* DOMAIN DELTA WHEN RCT1 RANKED BEFORE RCT2 */
  IlcInt deltaEnd12   = ((emax1 < smax2) ? 0L : emax1 - smax2);
  IlcInt deltaStart12 = ((smin2 > emin1) ? 0L : emin1 - smin2);
  IlcInt delta12      = deltaEnd12 + deltaStart12;

  /* DOMAIN DELTA WHEN RCT2 RANKED BEFORE RCT1 */
  IlcInt deltaEnd21   = ((emax2 < smax1) ? 0L : emax2 - smax1);
  IlcInt deltaStart21 = ((smin1 > emin2) ? 0L : emin2 - smin1);
  IlcInt delta21      = deltaEnd21 + deltaStart21;

  /* MINIMAL NUMBER OF UNRANKED RESOURCE CONSTRAINTS */
  IlcInt nbUrkd1   =
((NbUnrankedExt*)(scheduler.getExtractable(srct1).getObject()))->getValue();
  IlcInt nbUrkd2   =
```

```
((NbUnrankedExt*)(scheduler.getExtractable(srct2).getObject()))->getValue();
  IlcInt minNbUrkd = (nbUrkd1 <= nbUrkd2) ? nbUrkd1 : nbUrkd2;

  /* RETURN MEASURE OF OPPORTUNITY */
  return (minNbUrkd * (delta12 - delta21));
}

IlcBool
SelectMostOpportunisticConflict(IlcScheduler& schedule,
                                IlcResourceConstraint& selectedRct1,
                                IlcResourceConstraint& selectedRct2) {

  IlcBool existsConflict = IlcFalse;

  IlcInt oppMaxAbs = -1;
  IlcInt oppMax = 0;
  IlcInt opp;

  for (IlcUnaryResourceIterator ires(schedule); ires.ok(); ++ires) {
    IlcUnaryResource resource = (*ires);
    if (resource.hasPrecedenceGraphConstraint() &&
        !resource.isRanked()) {

      /* FOR EACH RESOURCE CONSTRAINT, COMPUTE AND STORE THE NUMBER OF
         RESOURCE CONSTRAINTS UNRANKED W.R.T. IT */
      for (IlcResourceConstraintIterator irct(resource);
           irct.ok(); ++irct) {
        IlcResourceConstraint rct = (*irct);
        if (!rct.isRanked())
          ((NbUnrankedExt*)schedule.getExtractable(rct).getObject())->
            setValue(GetNumberOfUnranked(rct));
     }

      /* SELECT MOST OPPORTUNISTIC PAIR OF RESOURCE CONSTRAINT */
      for (IlcResourceConstraintIterator isrct1(resource);
           isrct1.ok(); ++isrct1) {
        IlcResourceConstraint srct1 = (*isrct1);
        if (!srct1.isRanked()) {
          for (IlcResourceConstraintIterator isrct2(srct1, IlcUnranked);
               isrct2.ok(); ++isrct2) {
            IlcResourceConstraint srct2 = (*isrct2);
      opp = GetOpportunity(schedule, srct1, srct2);
      if (oppMaxAbs < IloAbs(opp)) {
              existsConflict = IlcTrue;
              oppMaxAbs      = IlcAbs(opp);
              oppMax         = opp;
              selectedRct1   = srct1;
              selectedRct2   = srct2;
            }
          }
        }
      }
    }
  }

  /* SELECT WHICH BRANCH WILL BE CHOSEN FIRST AMONG RCT1 << RCT2 AND
     RCT2 << RCT1 */
  if (existsConflict && (0 < oppMax)) {
```

```
    IlcResourceConstraint tmpRct = selectedRct1;
    selectedRct1 = selectedRct2;
    selectedRct2 = tmpRct;
  }

  return existsConflict;
}
```

Developing the Problem-Solving Algorithm

A simple way to reduce the number of optimizing iterations is to replace the minimization loop by a binary search that dichotomizes the interval where the possible values of the constrained `makespan` variable occur.

When the domain of the `makespan` variable is `[min max]`, we try to solve the problem with a new constraint stating that the `makespan` variable must be less than or equal to `(min+max)/2`. If this trial succeeds, `max` becomes the `makespan` of the solution that was found; if the trial fails, `min` becomes `1+((min+max)/2)`. The optimal solution is found when `min` and `max` are equal.

```
void
Dichotomize(IloModel& model,
            IloSolver& solver,
            const IloNumVar& makespan)
{
  /* GET MAKESPAN INITIAL BOUNDS */
  IloNum min = makespan.getLB();
  IloNum max = makespan.getUB();

  /* OPTIMIZE. */
  IloGoal goal = IloRankForward(solver.getEnv(), makespan,
                                IloSelResMinLocalSlack,
                                IloSelFirstRCMinStartMax);
  while (min < max) {
    IloNum value = IloFloor((min + max) / 2);
    IloConstraint ct = (makespan <= value);
    model.add(ct);
    if (solver.solve(goal)) {
      max = solver.getMin(makespan);
      solver.out() << "Solution with makespan " << max << endl;
    }
    else {
      solver.out() << "Failure with makespan " << value << endl;
      min = value + 1;
    }

    model.remove(ct);
  }

  /* RECOMPUTE THE OPTIMAL SOLUTION. THIS STEP COULD BE AVOIDED IF
     SOLUTIONS WERE STORED AS PART OF THE OPTIMIZATION PROCESS. */
  model.add(makespan == max);
  solver.solve(goal);
```

```
}
```

Using this dichotomizing binary search reduces the number of iterations needed to solve the problem, but it means harder iterations. The iterations are harder because we have to prove quasi-optimality of a solution prior to proving its optimality. To simplify those hard iterations, we use the predefined resource selector `IloSelResMinLocalSlack`. This resource selector uses the member function `getLocalSlack` to calculate the *slack* of a resource more carefully. It considers all time periods [*T1 T2*) for which *T1* is the earliest start time of an activity and *T2* is the latest end time of any other activity.

Let's consider an example to clarify this idea. Let's say there are three activities, A, B, and C. Activity A has a duration of two days and can take place anytime in the next twenty days; activity B lasts 5 days, cannot start before day 1, and must be finished by day 12; and activity C will last 4 days, can start on day 2, and must be finished by day 13.

Table 16.1 *Scheduling Activities*

Activity	Earliest Start Time	Latest End Time	Duration
A	0	20	2
B	1	12	5
C	2	13	4

Considering the problem globally, we can take the minimal earliest start time. That is, we take the soonest of the earliest start times (day 0 for activity A) and the last of the latest end times (day 20 for activity A), so there are twenty days available. The activities take only 11 days total (2+5+4), so there are 9 days slack globally.

However, if we refine our idea of slack by considering the earliest start time of any activity and the latest end time of any other activity, we get much tighter slack times. By considering the earliest start time of activity B and the latest end time of activity C, we get an overall span of 12 days. The total duration of the activities that must absolutely execute during this period is 9 days (5 days for B and 4 days for C), so we now have only 3 days of slack.

This more refined idea of slack is used in the program.

Complete Program and Output

You can see the entire program `jobshopd.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

#if defined(ILO_SDXLOUTPUT)
```

```
#include “sdxloutput.h”
#endif

IloInt ResourceNumbers06 [] = {2, 0, 1, 3, 5, 4,
                               1, 2, 4, 5, 0, 3,
                               2, 3, 5, 0, 1, 4,
                               1, 0, 2, 3, 4, 5,
                               2, 1, 4, 5, 0, 3,
                               1, 3, 5, 0, 4, 2};

IloInt Durations06 [] = { 1,  3,  6,  7,  3,  6,
                          8,  5, 10, 10, 10,  4,
                          5,  4,  8,  9,  1,  7,
                          5,  5,  5,  3,  8,  9,
                          9,  3,  5,  4,  3,  1,
                          3,  3,  9, 10,  4,  1};

IloInt ResourceNumbers10 [] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
                               0, 2, 4, 9, 3, 1, 6, 5, 7, 8,
                               1, 0, 3, 2, 8, 5, 7, 6, 9, 4,
                               1, 2, 0, 4, 6, 8, 7, 3, 9, 5,
                               2, 0, 1, 5, 3, 4, 8, 7, 9, 6,
                               2, 1, 5, 3, 8, 9, 0, 6, 4, 7,
                               1, 0, 3, 2, 6, 5, 9, 8, 7, 4,
                               2, 0, 1, 5, 4, 6, 8, 9, 7, 3,
                               0, 1, 3, 5, 2, 9, 6, 7, 4, 8,
                               1, 0, 2, 6, 8, 9, 5, 3, 4, 7};

IloInt Durations10 [] = {29, 78,  9, 36, 49, 11, 62, 56, 44, 21,
                         43, 90, 75, 11, 69, 28, 46, 46, 72, 30,
                         91, 85, 39, 74, 90, 10, 12, 89, 45, 33,
                         81, 95, 71, 99,  9, 52, 85, 98, 22, 43,
                         14,  6, 22, 61, 26, 69, 21, 49, 72, 53,
                         84,  2, 52, 95, 48, 72, 47, 65,  6, 25,
                         46, 37, 61, 13, 32, 21, 32, 89, 30, 55,
                         31, 86, 46, 74, 32, 88, 19, 48, 36, 79,
                         76, 69, 76, 51, 85, 11, 40, 89, 26, 74,
                         85, 13, 61,  7, 64, 76, 47, 52, 90, 45};

IloInt ResourceNumbers20 [] = {0, 1, 2, 3, 4,
                               0, 1, 3, 2, 4,
                               1, 0, 2, 4, 3,
                               1, 0, 4, 2, 3,
                               2, 1, 0, 3, 4,
                               2, 1, 4, 0, 3,
                               1, 0, 2, 3, 4,
                               2, 1, 0, 3, 4,
                               0, 3, 2, 1, 4,
                               1, 2, 0, 3, 4,
                               1, 3, 0, 4, 2,
                               2, 0, 1, 3, 4,
                               0, 2, 1, 3, 4,
                               2, 0, 1, 3, 4,
                               0, 1, 4, 2, 3,
                               1, 0, 3, 4, 2,
                               0, 2, 1, 3, 4,
                               0, 1, 4, 2, 3,
                               1, 2, 0, 3, 4,
```

```
                                0, 1, 2, 3, 4};

IloInt Durations20 [] = {29,  9, 49, 62, 44,
                         43, 75, 69, 46, 72,
                         91, 39, 90, 12, 45,
                         81, 71,  9, 85, 22,
                         14, 22, 26, 21, 72,
                         84, 52, 48, 47,  6,
                         46, 61, 32, 32, 30,
                         31, 46, 32, 19, 36,
                         76, 76, 85, 40, 26,
                         85, 61, 64, 47, 90,
                         78, 36, 11, 56, 21,
                         90, 11, 28, 46, 30,
                         85, 74, 10, 89, 33,
                         95, 99, 52, 98, 43,
                          6, 61, 69, 49, 53,
                          2, 95, 72, 65, 25,
                         37, 13, 21, 89, 55,
                         86, 74, 88, 48, 79,
                         69, 51, 11, 89, 74,
                         13,  7, 76, 52, 45};


////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////
class NbUnrankedExt {
private:
  IlcInt _nbOfUnranked;

public:
  NbUnrankedExt() :_nbOfUnranked(0) {};
  ~NbUnrankedExt(){};
  void setValue(IlcInt nbOfUnranked) {
    _nbOfUnranked = nbOfUnranked;
  }
  IlcInt getValue() const {
    return _nbOfUnranked;
  }
};
IloModel
DefineModel(IloEnv& env,
            IloInt numberOfJobs,
            IloInt numberOfResources,
            IloInt* resourceNumbers,
            IloInt* durations,
            IloNumVar& makespan)
{
  IloModel model(env);

  /* CREATE THE MAKESPAN VARIABLE. */
  IloInt numberOfActivities = numberOfJobs * numberOfResources;
  IloInt horizon = 0;
  IloInt k;
```

```
  for (k = 0; k < numberOfActivities; k++)
    horizon += durations[k];

  makespan = IloNumVar(env, 0, horizon, ILOINT);

  /* CREATE THE RESOURCES. */
  IloSchedulerEnv schedEnv(env);
  schedEnv.getResourceParam().setCapacityEnforcement(IloMediumHigh);
  schedEnv.getResourceParam().setPrecedenceEnforcement(IloMediumHigh);

  IloInt j;
  IloUnaryResource *resources =
    new (env) IloUnaryResource[numberOfResources];
  for (j = 0; j < numberOfResources; j++)
    resources[j] = IloUnaryResource(env);

  /* CREATE THE ACTIVITIES. */
  char buffer[128];
  k = 0;
  IloInt i;
  for (i = 0; i < numberOfJobs; i++) {
    IloActivity previousActivity;
    for (j = 0; j < numberOfResources; j++) {
      IloActivity activity(env, durations[k]);
      sprintf(buffer, "J%ldS%ldR%ld", i, j, resourceNumbers[k]);
      activity.setName(buffer);

      IloResourceConstraint rct =
        activity.requires(resources[resourceNumbers[k]]);
      NbUnrankedExt* nbOfUnranked =
        new (env) NbUnrankedExt();
      rct.setObject(nbOfUnranked);
      model.add(rct);

      if (j != 0)
        model.add(activity.startsAfterEnd(previousActivity));
      previousActivity = activity;
      k++;
    }
    model.add(previousActivity.endsBefore(makespan));
  }

  /* RETURN THE MODEL. */
  return model;
}


////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////////////////

void
PrintSolution(const IloSolver& solver)
{
  IlcScheduler scheduler(solver);
  IloEnv env = solver.getEnv();
```

```
  for(IloIterator<IloActivity> ite(env); ite.ok(); ++ite)
    solver.out() << scheduler.getActivity(*ite) << endl;
}

////////////////////////////////////////////////////////////////////////////
//
// PROBLEM SOLVING
//
////////////////////////////////////////////////////////////////////////////
IlcInt GetNumberOfUnranked(const IlcResourceConstraint& rct) {
  /* RETURN NUMBER OF UNRANKED W.R.T. RCT */
  IlcInt nb = 0;
  for (IlcResourceConstraintIterator ite(rct, IlcUnranked);
       ite.ok(); ++ite)
    nb++;
  return nb;
}

IlcInt GetOpportunity(const IlcScheduler& scheduler,
                      const IlcResourceConstraint& srct1,
                      const IlcResourceConstraint& srct2) {

  IlcActivity act1 = srct1.getActivity();
  IlcActivity act2 = srct2.getActivity();

  IlcInt smin1 = act1.getStartMin();
  IlcInt smax1 = act1.getStartMax();
  IlcInt emin1 = act1.getEndMin();
  IlcInt emax1 = act1.getEndMax();
  IlcInt smin2 = act2.getStartMin();
  IlcInt smax2 = act2.getStartMax();
  IlcInt emin2 = act2.getEndMin();
  IlcInt emax2 = act2.getEndMax();

  /* DOMAIN DELTA WHEN RCT1 RANKED BEFORE RCT2 */
  IlcInt deltaEnd12   = ((emax1 < smax2) ? 0L : emax1 - smax2);
  IlcInt deltaStart12 = ((smin2 > emin1) ? 0L : emin1 - smin2);
  IlcInt delta12      = deltaEnd12 + deltaStart12;

  /* DOMAIN DELTA WHEN RCT2 RANKED BEFORE RCT1 */
  IlcInt deltaEnd21   = ((emax2 < smax1) ? 0L : emax2 - smax1);
  IlcInt deltaStart21 = ((smin1 > emin2) ? 0L : emin2 - smin1);
  IlcInt delta21      = deltaEnd21 + deltaStart21;

  /* MINIMAL NUMBER OF UNRANKED RESOURCE CONSTRAINTS */
  IlcInt nbUrkd1   =
((NbUnrankedExt*)(scheduler.getExtractable(srct1).getObject()))->getValue();
  IlcInt nbUrkd2   =
((NbUnrankedExt*)(scheduler.getExtractable(srct2).getObject()))->getValue();
  IlcInt minNbUrkd = (nbUrkd1 <= nbUrkd2) ? nbUrkd1 : nbUrkd2;

  /* RETURN MEASURE OF OPPORTUNITY */
  return (minNbUrkd * (delta12 - delta21));
}

IlcBool
SelectMostOpportunisticConflict(IlcScheduler& schedule,
                                IlcResourceConstraint& selectedRct1,
```

```
                                IlcResourceConstraint& selectedRct2) {

  IlcBool existsConflict = IlcFalse;

  IlcInt oppMaxAbs = -1;
  IlcInt oppMax = 0;
  IlcInt opp;

  for (IlcUnaryResourceIterator ires(schedule); ires.ok(); ++ires) {
    IlcUnaryResource resource = (*ires);
    if (resource.hasPrecedenceGraphConstraint() &&
        !resource.isRanked()) {

      /* FOR EACH RESOURCE CONSTRAINT, COMPUTE AND STORE THE NUMBER OF
         RESOURCE CONSTRAINTS UNRANKED W.R.T. IT */
      for (IlcResourceConstraintIterator irct(resource);
           irct.ok(); ++irct) {
        IlcResourceConstraint rct = (*irct);
        if (!rct.isRanked())
          ((NbUnrankedExt*)schedule.getExtractable(rct).getObject())->
            setValue(GetNumberOfUnranked(rct));
     }

      /* SELECT MOST OPPORTUNISTIC PAIR OF RESOURCE CONSTRAINT */
      for (IlcResourceConstraintIterator isrct1(resource);
           isrct1.ok(); ++isrct1) {
        IlcResourceConstraint srct1 = (*isrct1);
        if (!srct1.isRanked()) {
          for (IlcResourceConstraintIterator isrct2(srct1, IlcUnranked);
               isrct2.ok(); ++isrct2) {
            IlcResourceConstraint srct2 = (*isrct2);
      opp = GetOpportunity(schedule, srct1, srct2);
      if (oppMaxAbs < IloAbs(opp)) {
            existsConflict = IlcTrue;
            oppMaxAbs      = IlcAbs(opp);
            oppMax         = opp;
            selectedRct1   = srct1;
            selectedRct2   = srct2;
          }
        }
      }
    }
  }

  /* SELECT WHICH BRANCH WILL BE CHOSEN FIRST AMONG RCT1 << RCT2 AND
     RCT2 << RCT1 */
  if (existsConflict && (0 < oppMax)) {
    IlcResourceConstraint tmpRct = selectedRct1;
    selectedRct1 = selectedRct2;
    selectedRct2 = tmpRct;
  }

  return existsConflict;
}

ILCGOAL0(SolveConflictsIlc) {
  IloSolver s = getSolver();
```

```
  IlcScheduler scheduler = IlcScheduler(s);

  IlcResourceConstraint srct1;
  IlcResourceConstraint srct2;
  if (SelectMostOpportunisticConflict(scheduler, srct1, srct2))
    return IlcAnd(IlcTrySetSuccessor(srct1, srct2), this);

  return 0;
}

ILOCPGOALWRAPPER0(SolveConflicts, solver) {
  return SolveConflictsIlc(solver);
}
void
SetMakespanInitialBounds(IloSolver& solver,
                         IloNumVar& makespan)
{
  IloGoal goal;
  /* SET MAKESPAN LOWER BOUND AND RESTART */
  goal = IloGoalTrue(solver.getEnv());
  solver.solve(goal);
  makespan.setLB(solver.getMin(makespan));

  /* SOLVE WITH GOAL SOLVECONFLICTS */
  goal = SolveConflicts(solver.getEnv());
  solver.solve(goal);

  /* SET MAKESPAN UPPER BOUND */
  makespan.setUB(solver.getMin(makespan));
  solver.out() << "Solution with makespan " << makespan.getUB() << endl;
}

void
Dichotomize(IloModel& model,
            IloSolver& solver,
            const IloNumVar& makespan)
{
  /* GET MAKESPAN INITIAL BOUNDS */
  IloNum min = makespan.getLB();
  IloNum max = makespan.getUB();

  /* OPTIMIZE. */
  IloGoal goal = IloRankForward(solver.getEnv(), makespan,
                                IloSelResMinLocalSlack,
                                IloSelFirstRCMinStartMax);
  while (min < max) {
    IloNum value = IloFloor((min + max) / 2);
    IloConstraint ct = (makespan <= value);
    model.add(ct);
    if (solver.solve(goal)) {
      max = solver.getMin(makespan);
      solver.out() << "Solution with makespan " << max << endl;
    }
    else {
      solver.out() << "Failure with makespan " << value << endl;
      min = value + 1;
    }
```

```
    model.remove(ct);
  }

  /* RECOMPUTE THE OPTIMAL SOLUTION. THIS STEP COULD BE AVOIDED IF
     SOLUTIONS WERE STORED AS PART OF THE OPTIMIZATION PROCESS. */
  model.add(makespan == max);
  solver.solve(goal);
}
////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////

void
InitParameters(int argc,
               char** argv,
               IloInt& numberOfJobs,
               IloInt& numberOfResources,
               IloInt*& resourceNumbers,
               IloInt*& durations)
{
  if (argc > 1) {
    IloInt number = atol(argv[1]);
    if (number == 10) {
      numberOfJobs = 10;
      numberOfResources = 10;
      resourceNumbers = ResourceNumbers10;
      durations = Durations10;
    }
    else if (number == 20) {
      numberOfJobs = 20;
      numberOfResources = 5;
      resourceNumbers = ResourceNumbers20;
      durations = Durations20;
    }
  }
}

int main(int argc, char** argv)
{
  try {
    IloEnv env;

    IloInt numberOfJobs = 6;
    IloInt numberOfResources = 6;
    IloInt* resourceNumbers = ResourceNumbers06;
    IloInt* durations = Durations06;
    InitParameters(argc,
                   argv,
                   numberOfJobs,
                   numberOfResources,
                   resourceNumbers,
                   durations);
    IloNumVar makespan;
    IloModel model = DefineModel(env,
                                 numberOfJobs,
                                 numberOfResources,
```

```
                                  resourceNumbers,
                                  durations,
                                  makespan);
    IloSolver solver(model);
    SetMakespanInitialBounds(solver, makespan);
    Dichotomize(model, solver, makespan);
    PrintSolution(solver);
#if defined(ILO_SDXLOUTPUT)
      IloSDXLOutput output(env);
      ofstream outFile("jobshopd.xml");
      output.write(IlcScheduler(solver), outFile);
      outFile.close();
#endif
    solver.printInformation();
    env.end();
  }
  catch (IloException& exc) {
    cout << exc << endl;
  }

  return 0;
}

////////////////////////////////////////////////////////////////////////////
//
// RESULTS
//
////////////////////////////////////////////////////////////////////////////

/* jobshopd 6
   Solution with makespan 57
   Failure with makespan 52
   Solution with makespan 55
   Failure with makespan 54
   J0S0R2 [5..7 -- 1 --> 6..8]
   J0S1R0 [6..11 -- 3 --> 9..14]
   J0S2R1 [16 -- 6 --> 22]
   J0S3R3 [30..36 -- 7 --> 37..43]
   J0S4R5 [42..43 -- 3 --> 45..46]
   J0S5R4 [49 -- 6 --> 55]
   J1S0R1 [0 -- 8 --> 8]
   J1S1R2 [8 -- 5 --> 13]
   J1S2R4 [13..15 -- 10 --> 23..25]
   J1S3R5 [28..29 -- 10 --> 38..39]
   J1S4R0 [38..40 -- 10 --> 48..50]
   J1S5R3 [48..50 -- 4 --> 52..54]
   J2S0R2 [0..2 -- 5 --> 5..7]
   J2S1R3 [5..7 -- 4 --> 9..11]
   J2S2R5 [9..11 -- 8 --> 17..19]
   J2S3R0 [18..19 -- 9 --> 27..28]
   J2S4R1 [27..41 -- 1 --> 28..42]
   J2S5R4 [42 -- 7 --> 49]
   J3S0R1 [8 -- 5 --> 13]
   J3S1R0 [13..14 -- 5 --> 18..19]
   J3S2R2 [22 -- 5 --> 27]
   J3S3R3 [27 -- 3 --> 30]
   J3S4R4 [30 -- 8 --> 38]
   J3S5R5 [45..46 -- 9 --> 54..55]
```

```
   J4S0R2 [13 -- 9 --> 22]
   J4S1R1 [22 -- 3 --> 25]
   J4S2R4 [25 -- 5 --> 30]
   J4S3R5 [38..39 -- 4 --> 42..43]
   J4S4R0 [48..51 -- 3 --> 51..54]
   J4S5R3 [52..54 -- 1 --> 53..55]
   J5S0R1 [13 -- 3 --> 16]
   J5S1R3 [16 -- 3 --> 19]
   J5S2R5 [19 -- 9 --> 28]
   J5S3R0 [28 -- 10 --> 38]
   J5S4R4 [38 -- 4 --> 42]
   J5S5R2 [42..54 -- 1 --> 43..55]
*/

/* jobshopd 10
   Solution with makespan 1146
   Failure with makespan 900
   Solution with makespan 1023
   Solution with makespan 961
   Solution with makespan 930
   Failure with makespan 915
   Failure with makespan 923
   Failure with makespan 927
   Failure with makespan 929
   J0S0R0 [119 -- 29 --> 148]
   J0S1R1 [445..448 -- 78 --> 523..526]
   J0S2R2 [523..526 -- 9 --> 532..535]
   J0S3R3 [532..540 -- 36 --> 568..576]
   J0S4R4 [568..595 -- 49 --> 617..644]
   J0S5R5 [641..644 -- 11 --> 652..655]
   J0S6R6 [652..655 -- 62 --> 714..717]
   J0S7R7 [721 -- 56 --> 777]
   J0S8R8 [777..782 -- 44 --> 821..826]
   J0S9R9 [821..831 -- 21 --> 842..852]
   J1S0R0 [76 -- 43 --> 119]
   J1S1R2 [224..235 -- 90 --> 314..325]
   J1S2R4 [355 -- 75 --> 430]
   J1S3R9 [474..499 -- 11 --> 485..510]
   J1S4R3 [568..576 -- 69 --> 637..645]
   J1S5R1 [637..689 -- 28 --> 665..717]
   J1S6R6 [714..717 -- 46 --> 760..763]
   J1S7R5 [760..767 -- 46 --> 806..813]
   J1S8R7 [813 -- 72 --> 885]
   J1S9R8 [895..900 -- 30 --> 925..930]
   J2S0R1 [308..311 -- 91 --> 399..402]
   J2S1R0 [408..411 -- 85 --> 493..496]
   J2S2R3 [493..496 -- 39 --> 532..535]
   J2S3R2 [532..535 -- 74 --> 606..609]
   J2S4R8 [609 -- 90 --> 699]
   J2S5R5 [699 -- 10 --> 709]
   J2S6R7 [709 -- 12 --> 721]
   J2S7R6 [760..763 -- 89 --> 849..852]
   J2S8R9 [849..852 -- 45 --> 894..897]
   J2S9R4 [894..897 -- 33 --> 927..930]
   J3S0R1 [0..6 -- 81 --> 81..87]
   J3S1R2 [84..90 -- 95 --> 179..185]
   J3S2R0 [185 -- 71 --> 256]
   J3S3R4 [256 -- 99 --> 355]
```

```
J3S4R6 [359..370 -- 9 --> 368..379]
J3S5R8 [368..379 -- 52 --> 420..431]
J3S6R7 [420..431 -- 85 --> 505..516]
J3S7R3 [637..645 -- 98 --> 735..743]
J3S8R9 [766..809 -- 22 --> 788..831]
J3S9R5 [806..887 -- 43 --> 849..930]
J4S0R2 [210..221 -- 14 --> 224..235]
J4S1R0 [256..259 -- 6 --> 262..265]
J4S2R1 [286..289 -- 22 --> 308..311]
J4S3R5 [308..313 -- 61 --> 369..374]
J4S4R3 [370..404 -- 26 --> 396..430]
J4S5R4 [430 -- 69 --> 499]
J4S6R8 [499 -- 21 --> 520]
J4S7R7 [530..541 -- 49 --> 579..590]
J4S8R9 [594..657 -- 72 --> 666..729]
J4S9R6 [849..877 -- 53 --> 902..930]
J5S0R2 [0..3 -- 84 --> 84..87]
J5S1R1 [84..87 -- 2 --> 86..89]
J5S2R5 [86..90 -- 52 --> 138..142]
J5S3R3 [138..142 -- 95 --> 233..237]
J5S4R8 [233..244 -- 48 --> 281..292]
J5S5R9 [281..292 -- 72 --> 353..364]
J5S6R0 [361..364 -- 47 --> 408..411]
J5S7R6 [429..445 -- 65 --> 494..510]
J5S8R4 [499..510 -- 6 --> 505..516]
J5S9R7 [505..516 -- 25 --> 530..541]
J6S0R1 [86..89 -- 46 --> 132..135]
J6S1R0 [148 -- 37 --> 185]
J6S2R3 [233..237 -- 61 --> 294..298]
J6S3R2 [314..325 -- 13 --> 327..338]
J6S4R6 [327..338 -- 32 --> 359..370]
J6S5R5 [421..440 -- 21 --> 442..461]
J6S6R9 [442..467 -- 32 --> 474..499]
J6S7R8 [520 -- 89 --> 609]
J6S8R7 [668..679 -- 30 --> 698..709]
J6S9R4 [698..740 -- 55 --> 753..795]
J7S0R2 [179..190 -- 31 --> 210..221]
J7S1R0 [262..265 -- 86 --> 348..351]
J7S2R1 [399..402 -- 46 --> 445..448]
J7S3R5 [445..461 -- 74 --> 519..535]
J7S4R4 [519..535 -- 32 --> 551..567]
J7S5R6 [558..567 -- 88 --> 646..655]
J7S6R8 [699..710 -- 19 --> 718..729]
J7S7R9 [718..729 -- 48 --> 766..777]
J7S8R7 [777 -- 36 --> 813]
J7S9R3 [813..851 -- 79 --> 892..930]
J8S0R0 [0 -- 76 --> 76]
J8S1R1 [217..220 -- 69 --> 286..289]
J8S2R3 [294..298 -- 76 --> 370..374]
J8S3R5 [370..374 -- 51 --> 421..425]
J8S4R2 [422..425 -- 85 --> 507..510]
J8S5R9 [507..510 -- 11 --> 518..521]
J8S6R6 [518..527 -- 40 --> 558..567]
J8S7R7 [579..590 -- 89 --> 668..679]
J8S8R4 [668..714 -- 26 --> 694..740]
J8S9R8 [821..826 -- 74 --> 895..900]
J9S0R1 [132..135 -- 85 --> 217..220]
J9S1R0 [348..351 -- 13 --> 361..364]
```

```
   J9S2R2 [361..364 -- 61 --> 422..425]
   J9S3R6 [422..428 -- 7 --> 429..435]
   J9S4R8 [429..435 -- 64 --> 493..499]
   J9S5R9 [518..521 -- 76 --> 594..597]
   J9S6R5 [594..597 -- 47 --> 641..644]
   J9S7R3 [735..743 -- 52 --> 787..795]
   J9S8R4 [787..795 -- 90 --> 877..885]
   J9S9R7 [885 -- 45 --> 930]
*/

/* jobshopd 20
   Solution with makespan 1405
   Failure with makespan 896
   Failure with makespan 1151
   Solution with makespan 1275
   Solution with makespan 1212
   Solution with makespan 1178
   Solution with makespan 1165
   Failure with makespan 1158
   Failure with makespan 1162
   Failure with makespan 1164
   J0S0R0 [188 -- 29 --> 217]
   J0S1R1 [552..575 -- 9 --> 561..584]
   J0S2R2 [581..598 -- 49 --> 630..647]
   J0S3R3 [749 -- 62 --> 811]
   J0S4R4 [954 -- 44 --> 998]
   J1S0R0 [145 -- 43 --> 188]
   J1S1R1 [209..215 -- 75 --> 284..290]
   J1S2R3 [338 -- 69 --> 407]
   J1S3R2 [439..440 -- 46 --> 485..486]
   J1S4R4 [485..486 -- 72 --> 557..558]
   J2S0R1 [830..832 -- 91 --> 921..923]
   J2S1R0 [921..923 -- 39 --> 960..962]
   J2S2R2 [960..962 -- 90 --> 1050..1052]
   J2S3R4 [1061 -- 12 --> 1073]
   J2S4R3 [1092 -- 45 --> 1137]
   J3S0R1 [601 -- 81 --> 682]
   J3S1R0 [692..726 -- 71 --> 763..797]
   J3S2R4 [900 -- 9 --> 909]
   J3S3R2 [1050..1052 -- 85 --> 1135..1137]
   J3S4R3 [1137 -- 22 --> 1159]
   J4S0R2 [0..1 -- 14 --> 14..15]
   J4S1R1 [14..15 -- 22 --> 36..37]
   J4S2R0 [37 -- 26 --> 63]
   J4S3R3 [63 -- 21 --> 84]
   J4S4R4 [84..168 -- 72 --> 156..240]
   J5S0R2 [204..205 -- 84 --> 288..289]
   J5S1R1 [369..375 -- 52 --> 421..427]
   J5S2R4 [421..438 -- 48 --> 469..486]
   J5S3R0 [763..797 -- 47 --> 810..844]
   J5S4R3 [1159 -- 6 --> 1165]
   J6S0R1 [756..758 -- 46 --> 802..804]
   J6S1R0 [810..844 -- 61 --> 871..905]
   J6S2R2 [892..905 -- 32 --> 924..937]
   J6S3R3 [988 -- 32 --> 1020]
   J6S4R4 [1109 -- 30 --> 1139]
   J7S0R2 [14..15 -- 31 --> 45..46]
   J7S1R1 [921..942 -- 46 --> 967..988]
```

```
J7S2R0 [967..988 -- 32 --> 999..1020]
J7S3R3 [1020 -- 19 --> 1039]
J7S4R4 [1073 -- 36 --> 1109]
J8S0R0 [63 -- 76 --> 139]
J8S1R3 [173 -- 76 --> 249]
J8S2R2 [706..723 -- 85 --> 791..808]
J8S3R1 [967..1099 -- 40 --> 1007..1139]
J8S4R4 [1139 -- 26 --> 1165]
J9S0R1 [284..290 -- 85 --> 369..375]
J9S1R2 [378..379 -- 61 --> 439..440]
J9S2R0 [507 -- 64 --> 571]
J9S3R3 [577 -- 47 --> 624]
J9S4R4 [665..666 -- 90 --> 755..756]
J10S0R1 [474..497 -- 78 --> 552..575]
J10S1R3 [624 -- 36 --> 660]
J10S2R0 [681..715 -- 11 --> 692..726]
J10S3R4 [844 -- 56 --> 900]
J10S4R2 [1135..1144 -- 21 --> 1156..1165]
J11S0R2 [288..289 -- 90 --> 378..379]
J11S1R0 [670..704 -- 11 --> 681..715]
J11S2R1 [802..804 -- 28 --> 830..832]
J11S3R3 [863 -- 46 --> 909]
J11S4R4 [1031 -- 30 --> 1061]
J12S0R0 [422 -- 85 --> 507]
J12S1R2 [507..517 -- 74 --> 581..591]
J12S2R1 [591 -- 10 --> 601]
J12S3R3 [660 -- 89 --> 749]
J12S4R4 [998 -- 33 --> 1031]
J13S0R2 [58..59 -- 95 --> 153..154]
J13S1R0 [323 -- 99 --> 422]
J13S2R1 [422..427 -- 52 --> 474..479]
J13S3R3 [479 -- 98 --> 577]
J13S4R4 [622..623 -- 43 --> 665..666]
J14S0R0 [139 -- 6 --> 145]
J14S1R1 [148..154 -- 61 --> 209..215]
J14S2R4 [228..295 -- 69 --> 297..364]
J14S3R2 [791..808 -- 49 --> 840..857]
J14S4R3 [1039 -- 53 --> 1092]
J15S0R1 [0..13 -- 2 --> 2..15]
J15S1R0 [228 -- 95 --> 323]
J15S2R3 [407 -- 72 --> 479]
J15S3R4 [557..558 -- 65 --> 622..623]
J15S4R2 [924..937 -- 25 --> 949..962]
J16S0R0 [0 -- 37 --> 37]
J16S1R2 [45..46 -- 13 --> 58..59]
J16S2R1 [58..63 -- 21 --> 79..84]
J16S3R3 [84 -- 89 --> 173]
J16S4R4 [173..240 -- 55 --> 228..295]
J17S0R0 [584..596 -- 86 --> 670..682]
J17S1R1 [682 -- 74 --> 756]
J17S2R4 [756 -- 88 --> 844]
J17S3R2 [844..857 -- 48 --> 892..905]
J17S4R3 [909 -- 79 --> 988]
J18S0R1 [79..85 -- 69 --> 148..154]
J18S1R2 [153..154 -- 51 --> 204..205]
J18S2R0 [217 -- 11 --> 228]
J18S3R3 [249 -- 89 --> 338]
J18S4R4 [338..364 -- 74 --> 412..438]
```

```
   J19S0R0 [571 -- 13 --> 584]
   J19S1R1 [584 -- 7 --> 591]
   J19S2R2 [630..647 -- 76 --> 706..723]
   J19S3R3 [811 -- 52 --> 863]
   J19S4R4 [909 -- 45 --> 954]
*/
```

Computational Remarks

This program works better than the one in Chapter 15. Here, we solve MT06 in 4 iterations and 4 backtracks, instead of 1 iteration and 78 backtracks. We solve MT10 in 8 iterations and 31011 backtracks. MT20 is solved in 11 iterations and 40 backtracks.

This example shows that combining a goal that finds a good first solution with an efficient dichotomic search may prove to be a good strategy for solving an optimization problem.

Starting from a good solution and using a dichotomic search both help lower the number of iterations. Implementing an efficient heuristic helps to reduce the complexity of each iteration.

We have experimented in this example with three improvements of the first basic version of the job-shop problem.

1. Starting with a good upper bound for the optimal makespan.
2. Using a dichotomic search.
3. Using a better heuristic to guide the search.

It can easily be shown by modifying parts of the program that it is the conjunction of these three improvements that leads to a satisfactory resolution of the job-shop problems MT06, MT10 and MT20.

- Not using the goal that finds a good upper bound for `makespan` leads to an approach that solves MT06 and MT10, but that leaves MT20 unsolved after a huge amount of computational time.
- Not using the dichotomic search leads to a version that only solves MT06 in reasonable computational time.
- Using the global slack instead of the local slack leads to a program that efficiently solves MT06, solves MT20 in more than 18000 backtracks, and does not solve MT10 in a reasonable computational time.

This example illustrates the fact that a combination of techniques is often necessary to solve hard optimization problems. Yet even with this improved algorithm, we notice that solving some of the job-shop problems still consumes an inordinate amount of time. Another way of handling such problems is presented in Chapter 30, *A Randomizing Algorithm: the Job-Shop Problem.*

C H A P T E R

17

Scheduling with Discrete Resources: the Ship-loading Problem

The bridge construction problem in Chapter 14 and the job-shop scheduling problem in Chapter 15 consist of activities subject only to temporal constraints and to unary resource constraints. In that context, to generate a schedule, we simply had to *order* the activities requiring the same resource.

This chapter presents an example in which activities require a discrete resource of capacity strictly greater than 1. In this context, two activities which require the same resource may overlap in time, but the total capacity required at time `t` must not exceed the capacity available at time `t`. The problem is solved by assigning precise start and end times to each activity in the schedule. The propagation of resource constraints is exploited to prune the search tree and to guarantee that the available capacity is never exceeded.

The problem in this chapter consists of scheduling the loading of a ship. Describing the Problem presents the simplest version of the problem. In this version, the duration of the activities and the required amounts of resource capacity are fixed once and for all. The capacity of the resource is also fixed. The goal is to minimize the makespan.

Describing the Problem

In this example, the duration of activities and their resource requirements are fixed. Two or more activities may require the same discrete resource, so they may overlap in time when we schedule them, as long as their combined requirements for the discrete resource do not exceed its capacity.

The problem consists of scheduling the loading of a ship. To our knowledge, this problem was first published in a French operations research book. It consists of 34 activities subject to precedence constraints, as indicated in the following table.

Table 17.1 *Activities and Successors*

Act.	Successors	Act.	Successors	Act.	Successors	Act.	Successors
1	2, 4	11	13	21	22	31	28
2	3	12	13	22	23	32	33
3	5, 7	13	15, 16	23	24	33	34
4	5	14	15	24	25	34	
5	6	15	18	25	26, 30, 31, 32		
6	8	16	17	26	27		
7	8	17	18	27	28		
8	9	18	19, 20, 21	28	29		
9	10, 14	19	23	29			
10	11, 12	20	23	30	28		

In this example, we'll consider a unique resource, the capacity of which is set to 8. The duration of each activity and the capacity required by each activity are set to the values in the following table.

Table 17.2 *Activity, Duration, and Capacity*

Act.	Dur.	Cap.	Act.	Dur.	Cap.	Act.	Dur.	Cap.	Act.	Dur.	Cap.
1	3	4	11	3	4	21	1	4	31	2	3
2	4	4	12	2	5	22	2	4	32	1	3
3	4	3	13	1	4	23	4	7	33	2	3
4	6	4	14	5	3	24	5	8	34	2	3

Table 17.2 *Activity, Duration, and Capacity*

Act.	Dur.	Cap.	Act.	Dur.	Cap.	Act.	Dur.	Cap.	Act.	Dur.	Cap.
5	5	5	15	2	3	25	2	8			
6	2	5	16	3	3	26	1	3			
7	3	4	17	2	6	27	1	3			
8	4	3	18	2	7	28	2	6			
9	3	4	19	1	4	29	1	8			
10	2	8	20	1	4	30	3	3			

As in the preceding examples, our goal is to minimize the makespan of the overall schedule, that is, to minimize the total amount of time that elapses between the beginning of the first activity and the end of the last activity.

Defining the Problem, Designing a Model

The first step consists of defining the problem.

- An instance of the class `IloDiscreteResource` represents the resource, and the capacity of the resource is passed as an argument to the constructor.
- Instances of the class `IloActivity` represent the activities, and again, the duration of each activity is passed as an argument to the constructor.
- The member function `IloActivity::startsAfterEnd` declares the precedence constraints.
- The `requires` member function of the class `IloActivity` is used to declare the resource requirement constraints.

The underlying constraint propagation mechanism in Scheduler systematically updates the earliest and latest start and end times for activities when decisions are made. For each activity *A*, it is guaranteed that the earliest and latest start and end times of *A* are consistent with the temporal constraints and with the resource requirements of "scheduled" activities (that is, activities for which the start and end times have been fixed).[1] This fact, in turn, guarantees that any attempt to fix the start and end times of all activities will succeed only if the chosen start and end times satisfy all the constraints of the problem.

1. This does not mean that the earliest and latest start and end times of all activities (scheduled and unscheduled) are globally consistent. It is necessary to explore the search space to determine a globally consistent solution.

Setting the Enforcement Level

The following code allows you to specify the enforcement level from the command line. The enforcement level allows specifying the effort with which the constraints are enforced on a given resource.

```
IloEnforcementLevel level = IloBasic;
if (argc > 1) {
  if (!strcmp(argv[1], “IloLow”))
    level = IloLow;
  else if (!strcmp(argv[1], “IloMediumLow”))
    level = IloMediumLow;
  else if (!strcmp(argv[1], “IloBasic”))
    level = IloBasic;
  else if (!strcmp(argv[1], “IloMediumHigh”))
    level = IloMediumHigh;
  else if (!strcmp(argv[1], “IloHigh”))
    level = IloHigh;
  else if (!strcmp(argv[1], “IloExtended”))
    level = IloExtended;
}
```

Solving the Problem: Minimizing Makespan

Once all the constraints are added, we can use the non-deterministic programming primitives of Solver to generate a solution with minimal makespan. The first thing to do is to define a constrained `makespan` variable and specify that its value is greater than the end time of each activity. For each activity `activities[i]`, the statement `model.add(activities[i].endsBefore(makespan));` constrains the `makespan` variable to be greater than or equal to the end time of `activities[i]`.

The second thing to do is to write a function that (a) generates a solution to the problem and (b) sets the constrained `makespan` variable to the makespan of the solution we get. We’ll adopt the following algorithm for that purpose.

1. Consider all the activities to schedule as “selectable.”
2. If all the activities have fixed start and end times, set the constrained `makespan` variable to the makespan of the solution obtained this way and exit. Otherwise, consider as non-selectable those activities which have fixed start and end times.
3. If the set of selectable activities is not empty, select an activity from the set, create a choice point for the selected activity (to allow backtracking), and schedule the selected activity from its earliest start time to its earliest end time. Then go to step 2.
4. If the set of selectable activities is empty, backtrack to the most recent choice point. If there is no such choice point, report that there is no problem solution and exit.

5. Upon backtracking, consider the activity that was scheduled at the choice point under consideration as “non-selectable” as long as its earliest start time has not changed. Then go to step 2.

The correctness of this algorithm is easy to demonstrate: The activity *A* chosen at step 3 must either start at its earliest start time or must be postponed to start later, but starting *A* later makes sense only if other activities prevent *A* from starting at its earliest start time. In that case, the scheduling of these other activities must eventually result in the update of the earliest start and end times of *A*. Hence, a backtrack from step 4 signifies that all the activities which do not have fixed start and end times have been postponed. Such a situation would be absurd, as in any solution the activity which will start the earliest among those which have been postponed could be set to start at its earliest start time. Consequently, it is correct to backtrack from step 4 up to the most recent choice point.

The search algorithm we will use is implemented by the predefined goal `IloSetTimesForward`. At each step, this goal selects a selectable activity and sets a choice point: either the chosen activity is scheduled to start at its earliest start time, or it is postponed.

To select the activities, the predefined activity selector `IloSelFirstActMinEndMin` is used. This selector considers the earliest start and end times among the activities that can still be selected. Then among the selectable activities having the minimal earliest start time, it chooses one having the minimal earliest end time.

`IloMinimize` sets the objective of minimizing `makespan` in the model.

```
    IloModel model(env);
    IloNumVar makespan;
    DefineProblem(model,
                  NumberOfActivities,
                  Durations,
                  Demands,
                  Capacity,
                  Precedences,
                  level,
                  makespan);

    model.add(IloMinimize(env, makespan));

    // Algorithm
    IloSolver solver(model);
    IloGoal goal = IloSetTimesForward(env, makespan, IloSelFirstActMinEndMin);

    if (solver.solve(goal)) {
      PrintSolution(IlcScheduler(solver));
      solver.printInformation();

#if defined(ILO_SDXLOUTPUT)
      IloSDXLOutput output(env);
      ofstream outFile("ship.xml");
      output.write(IlcScheduler(solver), outFile, solver.getIntVar(makespan));
      outFile.close();
#endif
    }
    else
      solver.out() << "No solution!" << endl;

    env.end();
```

Complete Program and Output

You can see the entire program `ship.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

#if defined(ILO_SDXLOUTPUT)
#include "sdxloutput.h"
#endif

IloInt Capacity = 8;
IloInt NumberOfActivities = 34;
IloInt Durations [] = {3, 4, 4, 6, 5, 2, 3, 4, 3, 2,
                       3, 2, 1, 5, 2, 3, 2, 2, 1, 1,
                       1, 2, 4, 5, 2, 1, 1, 2, 1, 3,
                       2, 1, 2, 2 };
IloInt Demands [] = {4, 4, 3, 4, 5, 5, 4, 3, 4, 8,
```

```
                      4, 5, 4, 3, 3, 3, 6, 7, 4, 4,
                      4, 4, 7, 8, 8, 3, 3, 6, 8, 3,
                      3, 3, 3, 3 };
IloInt Precedences [] = {1, 2, 1, 4,
                         2, 3,
                         3, 5, 3, 7,
                         4, 5,
                         5, 6,
                         6, 8,
                         7, 8,
                         8, 9,
                         9, 10, 9, 14,
                         10, 11, 10, 12,
                         11, 13,
                         12, 13,
                         13, 15, 13, 16,
                         14, 15,
                         15, 18,
                         16, 17,
                         17, 18,
                         18, 19, 18, 20, 18, 21,
                         19, 23,
                         20, 23,
                         21, 22,
                         22, 23,
                         23, 24,
                         24, 25,
                         25, 26, 25, 30, 25, 31, 25, 32,
                         26, 27,
                         27, 28,
                         28, 29,
                         30, 28,
                         31, 28,
                         32, 33,
                         33, 34,
                         0, 0};

////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////

void DefineProblem(IloModel model,
                   IloInt numberOfActivities,
                   IloInt* durations,
                   IloInt* demands,
                   IloInt capacity,
                   IloInt* precedences,
                   IloEnforcementLevel level,
                   IloNumVar& makespan)
{
  IloEnv env = model.getEnv();

  /* COMPUTE THE HORIZON. */
  IloInt horizon = 0;
  IloInt i;
  for (i = 0; i < numberOfActivities; i++)
```

```
    horizon += durations[i];
  IloSchedulerEnv schedEnv (env);
  schedEnv.setHorizon(horizon);
  /* CREATE THE MAKESPAN VARIABLE. */
  makespan = IloIntVar(env, 0, horizon);

  /* CREATE THE RESOURCE. */
  IloDiscreteResource resource(env, capacity);
  resource.setCapacityEnforcement(level);

  /* CREATE THE ACTIVITIES. */
  IloArray<IloActivity> activities(env, numberOfActivities);
  for (i = 0; i < numberOfActivities; i++) {
    char name[128];
    sprintf(name, "Activity %ld ", i+1);
    activities[i] = IloActivity(env, durations[i], name);
    model.add(activities[i].requires(resource, demands[i]));
    model.add(activities[i].endsBefore(makespan));
  }
  /* POST THE PRECEDENCE CONSTRAINTS. */
  IloInt precedenceIndex;
  for (precedenceIndex = 0; ; precedenceIndex = precedenceIndex + 2) {
    IloInt predNumber = precedences[precedenceIndex] - 1;
    if (predNumber == -1)
      break;
    IloInt succNumber = precedences[precedenceIndex + 1] - 1;
    model.add(activities[succNumber].startsAfterEnd(activities[predNumber]));
  }
  /* CLEAR THE ARRAY OF ACTIVITIES. */
  activities.clear();
}

////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////////////////

void
PrintSolution(IlcScheduler scheduler)
{
  for(IlcActivityIterator iterator(scheduler);
      iterator.ok();
      ++iterator)
    scheduler.getSolver().out() << *iterator << endl;
}

////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////

int main(int argc, char** argv) {
  try {
    IloEnv env;
    IloEnforcementLevel level = IloBasic;
    if (argc > 1) {
```

```
      if (!strcmp(argv[1], “IloLow”))
        level = IloLow;
      else if (!strcmp(argv[1], “IloMediumLow”))
        level = IloMediumLow;
      else if (!strcmp(argv[1], “IloBasic”))
        level = IloBasic;
      else if (!strcmp(argv[1], “IloMediumHigh”))
        level = IloMediumHigh;
      else if (!strcmp(argv[1], “IloHigh”))
        level = IloHigh;
      else if (!strcmp(argv[1], “IloExtended”))
        level = IloExtended;
    }
    IloModel model(env);
    IloNumVar makespan;
    DefineProblem(model,
                  NumberOfActivities,
                  Durations,
                  Demands,
                  Capacity,
                  Precedences,
                  level,
                  makespan);

    model.add(IloMinimize(env, makespan));

    // Algorithm
    IloSolver solver(model);
    IloGoal goal = IloSetTimesForward(env, makespan, IloSelFirstActMinEndMin);

    if (solver.solve(goal)) {
      PrintSolution(IlcScheduler(solver));
      solver.printInformation();

#if defined(ILO_SDXLOUTPUT)
      IloSDXLOutput output(env);
      ofstream outFile(“ship.xml”);
      output.write(IlcScheduler(solver), outFile, solver.getIntVar(makespan));
      outFile.close();
#endif
    }
    else
      solver.out() << “No solution!” << endl;

    env.end();


  } catch (IloException& exc) {
    cout << exc << endl;
  }

  return 0;
}

////////////////////////////////////////////////////////////////////////////
//
// RESULTS
//
```

```
////////////////////////////////////////////////////////////////////////////

/* ship IloBasic
   Activity 34  [61 -- 2 --> 63]
   Activity 33  [58 -- 2 --> 60]
   Activity 32  [57 -- 1 --> 58]
   Activity 31  [59 -- 2 --> 61]
   Activity 30  [60 -- 3 --> 63]
   Activity 29  [65 -- 1 --> 66]
   Activity 28  [63 -- 2 --> 65]
   Activity 27  [58 -- 1 --> 59]
   Activity 26  [57 -- 1 --> 58]
   Activity 25  [55 -- 2 --> 57]
   Activity 24  [50 -- 5 --> 55]
   Activity 23  [46 -- 4 --> 50]
   Activity 22  [44 -- 2 --> 46]
   Activity 21  [43 -- 1 --> 44]
   Activity 20  [43 -- 1 --> 44]
   Activity 19  [44 -- 1 --> 45]
   Activity 18  [41 -- 2 --> 43]
   Activity 17  [39 -- 2 --> 41]
   Activity 16  [36 -- 3 --> 39]
   Activity 15  [36 -- 2 --> 38]
   Activity 14  [30 -- 5 --> 35]
   Activity 13  [35 -- 1 --> 36]
   Activity 12  [30 -- 2 --> 32]
   Activity 11  [32 -- 3 --> 35]
   Activity 10  [28 -- 2 --> 30]
   Activity 9   [25 -- 3 --> 28]
   Activity 8   [21 -- 4 --> 25]
   Activity 7   [11 -- 3 --> 14]
   Activity 6   [19 -- 2 --> 21]
   Activity 5   [14 -- 5 --> 19]
   Activity 4   [3 -- 6 --> 9]
   Activity 3   [7 -- 4 --> 11]
   Activity 2   [3 -- 4 --> 7]
   Activity 1   [0 -- 3 --> 3]
*/
```

The results show the name of each activity followed by information about its place in the schedule. This information is enclosed in square brackets. It consists of three items: start time, duration, and end time of the activity.

For example, the results show that Activity 1 should begin on the first day (day 0), last three days, and terminate by day 3. The following diagram displays the solution, which is split in two parts (day 0 to day 50 and day 50 to day 66) simply for ease of viewing. The numbered blocks on that diagram represent the scheduled activities. Block 1, for example, begins on day 0 and ends by day 3. Block 2 begins on day 3 and ends by day 7.

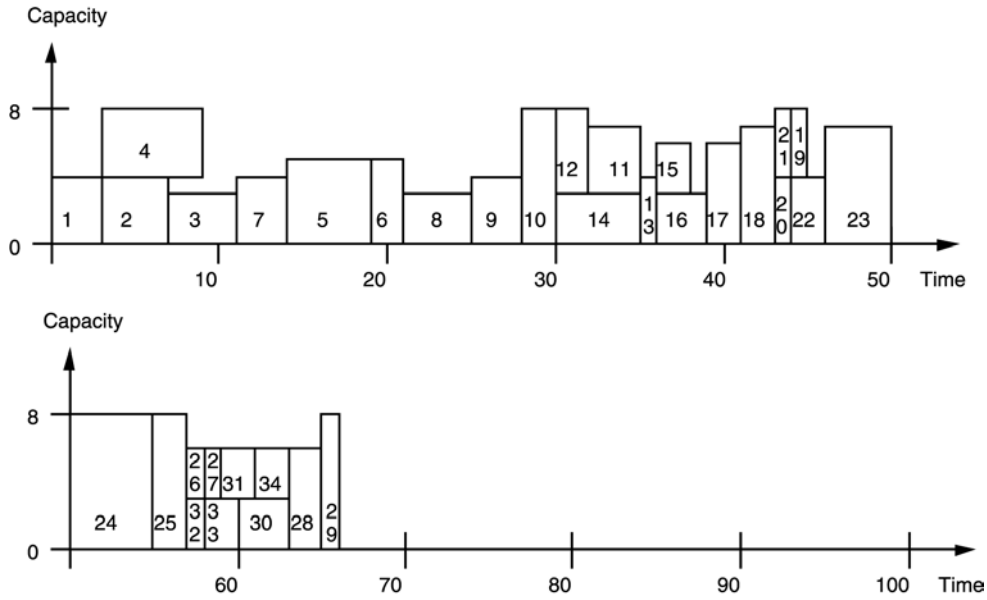


Figure 17.1 *Ship-Loading Problem Solution*

In Chapter 18, we show how the ship-loading problem can be decomposed into several independent subproblems. A new program that exploits that feature is developed there.

Computational Effects of Enforcement Level

The following table lists the computational effects of the various enforcement levels available in the program. (The CPU times are based on implementation of the program on a Pentium III ™ processor, and listed in seconds.)

Table 17.3 *Effects of Enforcement Level*

Level	Fails	Choice Points	Time
`IloLow`	513	544	0.09
`IloMediumLow`	513	544	0.09
`IloBasic`	513	544	0.09
`IloMediumHigh`	356	386	0.15
`IloHigh`	121	151	0.17

The three less-intensive enforcement levels have exactly the same effects on the number of fails, choice points, and CPU time. As the enforcement level increases to `IloMediumHigh`

and then `IloHigh`, the CPU time increases, as the amount of effort spent by the scheduler after each choice point at enforcing the constraints increases. Note, however, that the number of choice points and the number of fails substantially decrease at higher enforcement levels. While this decrease does not result in lower CPU time in this particular problem, it is sometimes the case that higher levels of enforcement do lead to significantly less CPU time. You should experiment with different levels in solving your own scheduling problems.

From the perspective of the extracted Scheduler classes:

- For discrete resources, the enforcement levels `IloLow`, `IloMediumLow`, and `IloHigh` correspond to the timetable constraint.
- For discrete resources, the enforcement level `IloMediumHigh` corresponds to using the timetable constraint and disjunctive constraint. By propagating the disjunctive constraints, the number of failures decreases. Doing more propagation, however, takes more time, and in this case, the problem is so easy that the investment in doing more propagation is not justified. If a more difficult problem were to be attacked, we might find that it is worthwhile to do more propagation; it would also be worthwhile in terms of CPU time.
- For discrete resources, the enforcement level `IloHigh` corresponds to using the timetable constraint, disjunctive constraint, and edgefinder algorithms. This decreases the number of failures further, and brings a smaller increase in CPU time.

For more information on how enforcement levels are interpreted, see *Resource Enforcement as Global Constraint Declaration* in the *IBM ILOG Scheduler Reference Manual*. The three algorithms mentioned here are also discussed there.

C H A P T E R

18

Scheduling with Discrete Resources: the Ship-loading Problem, Second Version

The ship-loading problem has a very interesting characteristic that we can actually see if we represent it as a graph where activities are vertices and the arcs between them are precedence constraints. Activities that are tightly constrained by each other will appear as a cluster of vertices connected by their constraints; other activities, more or less independent of that cluster, will appear separately (perhaps in other clusters) with few or no connecting arcs between such clusters. These clusters in the graph represent *subproblems* within our principal problem, subproblems that may prove more tractable than our initial problem.

For example, activity number 8 in the ship-loading problem is such that any other activity is either constrained to execute before activity number 8 or constrained to execute after activity number 8. As a result, any schedule with a minimal makespan must be such that activity number 8 executes as soon as possible. We could say that the activity number 8 is a *critical* activity and a *cut vertex* in the graph of activities.

On the basis of such a graph, we can decompose the problem into a series of smaller independent subproblems: each activity is either a cut vertex or belongs to a section separated by two cut vertices or the beginning and the end of the schedule. Consequently, it is possible to decompose the problem like this:

- First, select the next unscheduled *critical* activity. That first one is activity number 8.
- Next, consider only the selected activity and its predecessors, so as to minimize the start and end times of the selected activity.

- Then consider the other activities, without revising the schedule built for the selected activity and its predecessors.

Decomposing a Problem: General Observations

Exploiting such opportunities for decomposing problems can lead to very significant gains in terms of computational time. Furthermore, if an *incomplete* method is used, such a decomposition can produce a great improvement as well in terms of the quality of the generated solution.

Decomposing a scheduling problem is, in general, highly recommended. Several cases can occur.

- Case 1: It is possible to identify a perfect decomposition of the problem, that is, a decomposition that guarantees that the best problem solutions can be found by working on each subproblem separately or sequentially. Such is the case here for the ship-loading problem, so we can break the problem down into subproblems without hesitation.
- Case 2: It is possible to identify decompositions that seem very appropriate for the problem under consideration, although there is no guarantee that the best solutions will not be missed. An example of that situation is when there exists a very well identified bottleneck resource. In such a case, scheduling the bottleneck resource first is very likely to lead to globally optimal or almost optimal solutions. Another example occurs when we build a long-term plan and use it for short-term scheduling.
- Case 3: No appropriate decomposition is found. In such a case, from a purely theoretical point of view, it is better not to decompose the problem. However, it may be necessary to decompose the problem anyway for practical reasons, especially if the problem is very big or otherwise intractable.

In light of these generalities, this chapter provides an example of decomposition that uses the precedence graph facilities of Scheduler.

In this example, the definition of the problem is quite similar to the one described in Chapter 17, *Scheduling with Discrete Resources: the Ship-loading Problem*. It is described in Defining the Problem, Designing a Model. A first step of the search allows decomposition of the problem as illustrated in Decomposing the Problem. This decomposition is then exploited to solve each sub-problem independently, as shown in Solving the Problem: Exploiting the Decomposition.

Defining the Problem, Designing a Model

We define a class, `Activity`, with two member functions, `setCritical` and `isCritical`, for marking critical activities and two member functions, `setSubProblem` and `getSubProblem`, for marking to which subproblem the activity belongs.

```
class Activity {
private:
  IloActivity _act;
  IloBool     _critical;
  IloInt      _subProblem;

public:
  Activity(IloEnv env,
           IloInt number,
           IloInt processingTime);
  ~Activity();
  IloActivity getActivity() const {
    return _act;
  }
  void setCritical(IloBool critical) {
    _critical = critical;
  }
  IloBool isCritical() const {
    return _critical;
  }
  void setSubProblem(IloInt subProblem) {
    _subProblem = subProblem;
  }
  IloInt getSubProblem() const {
    return _subProblem;
  }
};
```

The rest of the problem definition is quite similar to the one described in the previous chapter except that each activity of the schedule is associated with an object `Activity` (by calling the constructor of the class `Activity`).

```
void
DefineProblem(IloModel          model,
              IloInt            numberOfActivities,
              IloInt*           processingTimes,
              IloInt*           demands,
              IloInt            capacity,
              IloInt*           precedences,
              IloActivityArray& activities,
              IloNumVar&        makespan)
{

    /* ... */

  /* CREATE THE ACTIVITIES. */
  activities = IloActivityArray(env,numberOfActivities);
  for (i = 0; i < numberOfActivities; i++) {
    Activity* act = new (env)
```

```
    Activity(env, i + 1, processingTimes[i]);
  activities[i] = act->getActivity();
  model.add(activities[i].requires(resource, demands[i]));
}
```

Decomposing the Problem

As we consider how to represent this problem, we need to tackle two issues. First, can we identify the critical activities that play a pivotal role in the problem, so that we can decompose the problem by breaking it down around those activities? The answer to that question is "yes" and there is even a formal mathematical technique for identifying those activities and a functionality of Scheduler (precedence graphs) that implements it.

Using Transitive Closure to Decompose a Problem

As you can see from our general remarks about decomposing problems, one way of decomposing a given problem is to find an activity (or a set of activities) that plays the same role that activity number 8 played in the previous chapter—the one that other activities necessarily precede or follow. Such an activity is known as a critical activity, and there is, in fact, a mathematical technique for identifying critical activities in a schedule. The technique is known as computing the *transitive closure* of the *precedence relation* imposed on the activities.

To put the same idea more formally, given a set of activities to schedule, we can identify an activity A such that any other activity B is either constrained to execute before A or constrained to execute after A by computing the transitive closure of the precedence relation.

This corresponds to the notion of "ranked activity" on Schedule Precedence Graphs.

Assigning Activities to Subproblems

The function `DecomposeProblem` first creates an extraction of the model, and then propagates on this extraction with an empty goal to solve. The purpose of this is to assign activities to the various subproblems. During this propagation, a precedence graph is used to determine which activities are the critical activities. The critical activities are then marked as belonging to the appropriate submodels.

The precedence enforcement is set to `IloMediumHigh`. This enforcement level aims at maintaining a precedence graph whose nodes are the schedule activities and whose directed edges represent precedence relations between activities. The `startsAfterEnd` constraints are automatically inserted as new edges when a precedence graph exists. The transitive closure of the graph is maintained by the constraint during the search.

The decomposition function scans the precedence graph and initializes the criticality status and the sections of the instances of class `Activity`. More precisely, the critical activities are the ones that are ranked on the precedence graph and the section of an activity corresponds to the number of ranked activities that precedes it.

As the precedence graph will not be used later in the program, once the problem has been decomposed, we set the enforcement level back to the default of `IloBasic`.

```
void
DecomposeProblem(IloModel            model,
                 IloActivityArray    activities,
                 IloNumVar           makespan,
                 IloModelArray&      subModels)
{
  // 1. CREATE DECOMPOSITION ALGORITHM AND SET SUBPROBLEMS
  IloEnv env = model.getEnv();
  IloSchedulerEnv schedEnv(env);

  schedEnv.setPrecedenceEnforcement(IloMediumHigh);

  IloSolver solver(model);
  solver.solve(IloGoalTrue(env));

  IloInt numberOfActivities = activities.getSize();
  IloInt nbSubProblems = 0;
  IloInt i;
  IlcScheduler schedr(solver);
  for (i = 0; i < numberOfActivities; i++) {
    Activity* act = ((Activity*) activities[i].getObject());
    if (schedr.getActivity(activities[i]).isRanked())
      act->setCritical(IloTrue);
    IloInt subProblem = 0;
    for (IloInt j = 0; j < numberOfActivities; j++)
      if ((i != j) &&
          schedr.getActivity(activities[j]).isRanked() &&
          schedr.getActivity(activities[j]).isSucceededBy
                          (schedr.getActivity(activities[i])))
        subProblem++;
    act->setSubProblem(subProblem);
    if (subProblem >= nbSubProblems)
      nbSubProblems = subProblem + 1;
  }

  schedEnv.setPrecedenceEnforcement(IloBasic);

 /* ... */
 }
```

Assigning Resource Constraints to Subproblems

Once the activities have all been marked, we create as many submodels as there are subproblems, and post the constraints into each submodel.

```
  // 2. CREATE SUB-MODELS
  subModels = IloModelArray(env,nbSubProblems);
  for (i = 0; i < nbSubProblems; i++) {
```

```
  subModels[i] = IloModel(env);
}

// 2.1 DECOMPOSE RESOURCE CONSTRAINTS
for(IloIterator<IloResourceConstraint> iterct(env);
    iterct.ok();
    ++iterct) {
  IloResourceConstraint rct(*iterct);
  Activity* act = ((Activity*) rct.getActivity().getObject());
  subModels[act->getSubProblem()].add(rct);
}

// 2.2 DECOMPOSE PRECEDENCE CONSTRAINTS
for(IloIterator<IloPrecedenceConstraint> itepct(env);
    itepct.ok();
    ++itepct) {
  IloPrecedenceConstraint pct(*itepct);
  Activity* precAct = ((Activity*) pct.getPrecedingActivity().getObject());
  Activity* follAct = ((Activity*) pct.getFollowingActivity().getObject());
  if ((precAct->getSubProblem() == follAct->getSubProblem()) ||
      ((precAct->getSubProblem() == follAct->getSubProblem() - 1) &&
       (precAct->isCritical())))
    subModels[follAct->getSubProblem()].add(pct);
}

// 2.3 SET SUB-MODELS MAKESPAN VARIABLES
for (i = 0; i < numberOfActivities; i++) {
  Activity* act = ((Activity*) activities[i].getObject());
  if (act->isCritical()) {
    IloNumExpr subModelMakespan = activities[i].getEndExpr();
    subModels[act->getSubProblem()].add(IloMinimize(env, subModelMakespan));
  }
  if (act->getSubProblem() == nbSubProblems - 1) {
    subModels[act->getSubProblem()].add(activities[i].endsBefore(makespan));
  }
}
subModels[nbSubProblems-1].add(IloMinimize(env, makespan));
```

Solving the Problem: Exploiting the Decomposition

The function `SolveSubProblem` assigns the start and end times of the activities within each subproblem, and stores them in the global solution given as argument.

As critical activities play a pivotal role between subproblems, each critical activity belongs to two submodels (the one it is the last activity of and the one it is the first activity of). When solving a subproblem *i*, the start and end times of the critical activity that starts subproblem *i* are already known as this critical activity was scheduled in subproblem *i-1*. Those start and end times are extracted from the solution.

```
void
SolveSubProblem(IloModel subModel,
                IloInt   s,
                IloActivityArray& activities,
```

```
                const IloSchedulerSolution& solution,
                IloBool lastSubProblem,
                IloNumVar makespan
                ) {
  IloEnv env = subModel.getEnv();
  IloSolver solver(env);
  solver.extract(subModel);
  IlcScheduler scheduler(solver);

  IloInt i = 0;
  IloInt numberOfActivities = activities.getSize();

  IloGoal goal;
  // Instantiate the  start- and end-time of the critical activities
  // in this sub-model based on the saved solution.
  for (i = 0; i < numberOfActivities; i++) {
    Activity* act = ((Activity*) activities[i].getObject());
    if ((act->getSubProblem() == s-1) && act->isCritical())  {
      IlcActivity ilcAct = scheduler.getActivity(activities[i]);
      ilcAct.setStartTime((IlcInt)solution.getStartMin(activities[i]));
      ilcAct.setEndTime((IlcInt)solution.getEndMax(activities[i]));
    }
    if ((act->getSubProblem() == s) && act->isCritical())  {
      assert(goal.getImpl() == 0);
      IloNumVar endVar(env, 0, IloInfinity, ILOINT);
      subModel.add(activities[i].endsAt(endVar));
      goal = IloSetTimesForward(env, endVar,
                                IloSelFirstActMinEndMin);
    }
  }

  if (!lastSubProblem) {
    solver.solve(goal);
  } else {
    assert(goal.getImpl() == 0);
    solver.solve(IloSetTimesForward(env, makespan, IloSelFirstActMinEndMin));
  }

  // Store the sub-model results in the global solution
  for (i = 0; i < numberOfActivities; i++) {
    Activity* act = ((Activity*) activities[i].getObject());
    if (act->getSubProblem() == s)
      solution.store(activities[i], scheduler);
  }
}
```

The final code creates an empty solution which is updated with the submodel solutions computed from `SolveSubProblem`.

```
int main() {
  try {
    IloEnv env;
    IloModel model(env);
    IloNumVar makespan;

    IloActivityArray activities(env);

    DefineProblem(model,
```

```
                    NumberOfActivities,
                    ProcessingTimes,
                    Demands,
                    Capacity,
                    Precedences,
                    activities,
                    makespan);

    /** Decompose the problem in a set of independent subproblems. */
    IloModelArray  subModels(env);
    DecomposeProblem(model, activities, makespan, subModels);

    /** Solve each sub-problem independently. */
    IloSchedulerSolution solution(env);
    IloInt nbSubProblems = subModels.getSize();
    for (IloInt i=0; i < nbSubProblems; i++) {
      IloModel subModel = subModels[i];
      SolveSubProblem(subModel, i, activities, solution,
                      i==nbSubProblems-1, makespan);
    }

    PrintSolution(activities, solution);
    solution.end();
    env.end();

  } catch (IloException& exc) {
    cout << exc << endl;
  }

  return 0;
}
```

Complete Program and Output

You can see the entire program `shipdec.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

IloInt Capacity = 8;
IloInt NumberOfActivities = 34;
IloInt ProcessingTimes [] = {3, 4, 4, 6, 5, 2, 3, 4, 3, 2,
                        3, 2, 1, 5, 2, 3, 2, 2, 1, 1,
                        1, 2, 4, 5, 2, 1, 1, 2, 1, 3,
                        2, 1, 2, 2};
IloInt Demands [] = {4, 4, 3, 4, 5, 5, 4, 3, 4, 8,
                     4, 5, 4, 3, 3, 3, 6, 7, 4, 4,
                     4, 4, 7, 8, 8, 3, 3, 6, 8, 3,
                     3, 3, 3, 3};
IloInt Precedences [] = {1, 2, 1, 4,
                         2, 3,
                         3, 5, 3, 7,
```

```
                                4, 5,
                                5, 6,
                                6, 8,
                                7, 8,
                                8, 9,
                                9, 10, 9, 14,
                                10, 11, 10, 12,
                                11, 13,
                                12, 13,
                                13, 15, 13, 16,
                                14, 15,
                                15, 18,
                                16, 17,
                                17, 18,
                                18, 19, 18, 20, 18, 21,
                                19, 23,
                                20, 23,
                                21, 22,
                                22, 23,
                                23, 24,
                                24, 25,
                                25, 26, 25, 30, 25, 31, 25, 32,
                                26, 27,
                                27, 28,
                                28, 29,
                                30, 28,
                                31, 28,
                                32, 33,
                                33, 34,
                                0, 0};

typedef IloArray<IloModel>    IloModelArray;
typedef IloArray<IloActivity> IloActivityArray;

////////////////////////////////////////////////////////////////////////////
//
// DEFINITION OF THE ACTIVITY CLASS
//
////////////////////////////////////////////////////////////////////////////
class Activity {
private:
  IloActivity _act;
  IloBool     _critical;
  IloInt      _subProblem;

public:
  Activity(IloEnv env,
           IloInt number,
           IloInt processingTime);
  ~Activity();
  IloActivity getActivity() const {
    return _act;
  }
  void setCritical(IloBool critical) {
    _critical = critical;
  }
  IloBool isCritical() const {
    return _critical;
```

```
  }
  void setSubProblem(IloInt subProblem) {
    _subProblem = subProblem;
  }
  IloInt getSubProblem() const {
    return _subProblem;
  }
};

Activity::Activity(IloEnv env,
                   IloInt number,
                   IloInt processingTime)
  :_act        (env, processingTime),
   _critical   (IloFalse),
   _subProblem (0)
{
  _act.setObject(this);
  char buffer[128];
  sprintf(buffer, "Activity %ld ", number);
  _act.setName(buffer);
}

Activity::~Activity() {}

////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////
void
DefineProblem(IloModel          model,
              IloInt            numberOfActivities,
              IloInt*           processingTimes,
              IloInt*           demands,
              IloInt            capacity,
              IloInt*           precedences,
              IloActivityArray& activities,
              IloNumVar&        makespan)
{
  IloEnv env = model.getEnv();

  /* CREATE THE MAKESPAN VARIABLE. */
  IloInt horizon = 0;
  IloInt i = 0;
  for (i = 0; i < numberOfActivities; i++)
    horizon += processingTimes[i];
  makespan = IloNumVar(env, 0, horizon, IloNumVar::Int);

  /* CREATE THE RESOURCE. */
  IloDiscreteResource resource(env, capacity);
  resource.setPrecedenceEnforcement(IloExtended);

  /* CREATE THE ACTIVITIES. */
  activities = IloActivityArray(env,numberOfActivities);
  for (i = 0; i < numberOfActivities; i++) {
    Activity* act = new (env)
      Activity(env, i + 1, processingTimes[i]);
    activities[i] = act->getActivity();
```

```
    model.add(activities[i].requires(resource, demands[i]));
  }

  /* POST THE PRECEDENCE CONSTRAINTS. */
  IloInt precedenceIndex;
  for (precedenceIndex = 0; ; precedenceIndex = precedenceIndex + 2) {
    IloInt predNumber = precedences[precedenceIndex] - 1;
    if (predNumber == -1)
      break;
    IloInt succNumber = precedences[precedenceIndex + 1] - 1;
    model.add(activities[succNumber].startsAfterEnd(activities[predNumber]));
  }
}

////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DECOMPOSITION
//
////////////////////////////////////////////////////////////////////////////
void
DecomposeProblem(IloModel            model,
                 IloActivityArray    activities,
                 IloNumVar           makespan,
                 IloModelArray&      subModels)
{
  // 1. CREATE DECOMPOSITION ALGORITHM AND SET SUBPROBLEMS
  IloEnv env = model.getEnv();
  IloSchedulerEnv schedEnv(env);

  schedEnv.setPrecedenceEnforcement(IloMediumHigh);

  IloSolver solver(model);
  solver.solve(IloGoalTrue(env));

  IloInt numberOfActivities = activities.getSize();
  IloInt nbSubProblems = 0;
  IloInt i;
  IlcScheduler schedr(solver);
  for (i = 0; i < numberOfActivities; i++) {
    Activity* act = ((Activity*) activities[i].getObject());
    if (schedr.getActivity(activities[i]).isRanked())
      act->setCritical(IloTrue);
    IloInt subProblem = 0;
    for (IloInt j = 0; j < numberOfActivities; j++)
      if ((i != j) &&
          schedr.getActivity(activities[j]).isRanked() &&
          schedr.getActivity(activities[j]).isSucceededBy
                          (schedr.getActivity(activities[i])))
        subProblem++;
    act->setSubProblem(subProblem);
    if (subProblem >= nbSubProblems)
      nbSubProblems = subProblem + 1;
  }

  schedEnv.setPrecedenceEnforcement(IloBasic);
  // 2. CREATE SUB-MODELS
  subModels = IloModelArray(env,nbSubProblems);
  for (i = 0; i < nbSubProblems; i++) {
```

```
    subModels[i] = IloModel(env);
  }

  // 2.1 DECOMPOSE RESOURCE CONSTRAINTS
  for(IloIterator<IloResourceConstraint> iterct(env);
      iterct.ok();
      ++iterct) {
    IloResourceConstraint rct(*iterct);
    Activity* act = ((Activity*) rct.getActivity().getObject());
    subModels[act->getSubProblem()].add(rct);
  }

  // 2.2 DECOMPOSE PRECEDENCE CONSTRAINTS
  for(IloIterator<IloPrecedenceConstraint> itepct(env);
      itepct.ok();
      ++itepct) {
    IloPrecedenceConstraint pct(*itepct);
    Activity* precAct = ((Activity*) pct.getPrecedingActivity().getObject());
    Activity* follAct = ((Activity*) pct.getFollowingActivity().getObject());
    if ((precAct->getSubProblem() == follAct->getSubProblem()) ||
        ((precAct->getSubProblem() == follAct->getSubProblem() - 1) &&
         (precAct->isCritical())))
      subModels[follAct->getSubProblem()].add(pct);
  }

  // 2.3 SET SUB-MODELS MAKESPAN VARIABLES
  for (i = 0; i < numberOfActivities; i++) {
    Activity* act = ((Activity*) activities[i].getObject());
    if (act->isCritical()) {
      IloNumExpr subModelMakespan = activities[i].getEndExpr();
      subModels[act->getSubProblem()].add(IloMinimize(env, subModelMakespan));
    }
    if (act->getSubProblem() == nbSubProblems - 1) {
      subModels[act->getSubProblem()].add(activities[i].endsBefore(makespan));
    }
  }
  subModels[nbSubProblems-1].add(IloMinimize(env, makespan));
}

////////////////////////////////////////////////////////////////////////////
//
// SOLVE SUB-MODELS
//
////////////////////////////////////////////////////////////////////////////
void
SolveSubProblem(IloModel subModel,
                IloInt   s,
                IloActivityArray& activities,
                const IloSchedulerSolution& solution,
                IloBool lastSubProblem,
                IloNumVar makespan
                ) {
  IloEnv env = subModel.getEnv();
  IloSolver solver(env);
  solver.extract(subModel);
  IlcScheduler scheduler(solver);

  IloInt i = 0;
```

```
  IloInt numberOfActivities = activities.getSize();

  IloGoal goal;
  // Instantiate the  start- and end-time of the critical activities
  // in this sub-model based on the saved solution.
  for (i = 0; i < numberOfActivities; i++) {
    Activity* act = ((Activity*) activities[i].getObject());
    if ((act->getSubProblem() == s-1) && act->isCritical())  {
      IlcActivity ilcAct = scheduler.getActivity(activities[i]);
      ilcAct.setStartTime((IlcInt)solution.getStartMin(activities[i]));
      ilcAct.setEndTime((IlcInt)solution.getEndMax(activities[i]));
    }
    if ((act->getSubProblem() == s) && act->isCritical())  {
      assert(goal.getImpl() == 0);
      IloNumVar endVar(env, 0, IloInfinity, ILOINT);
      subModel.add(activities[i].endsAt(endVar));
      goal = IloSetTimesForward(env, endVar,
                                IloSelFirstActMinEndMin);
    }
  }

  if (!lastSubProblem) {
    solver.solve(goal);
  } else {
    assert(goal.getImpl() == 0);
    solver.solve(IloSetTimesForward(env, makespan, IloSelFirstActMinEndMin));
  }

  // Store the sub-model results in the global solution
  for (i = 0; i < numberOfActivities; i++) {
    Activity* act = ((Activity*) activities[i].getObject());
    if (act->getSubProblem() == s)
      solution.store(activities[i], scheduler);
  }
}

////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////////////////

void
PrintSolution(IloActivityArray& activities,
              const IloSchedulerSolution& solution)
{
  IloInt numberOfActivities = activities.getSize();
  for (IloInt i = 0; i < numberOfActivities; i++) {
    IloActivity act = activities[i];
    solution.getEnv().out() << act.getName() << "  ["
                            << solution.getStartMin(act)
                            << " -- "
                            << solution.getProcessingTimeMin(act)
                            << " --> "
                            << solution.getEndMin(act)
                            << "]" << endl;
  }
}
```

```
////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////
int main() {
  try {
    IloEnv env;
    IloModel model(env);
    IloNumVar makespan;

    IloActivityArray activities(env);

    DefineProblem(model,
                  NumberOfActivities,
                  ProcessingTimes,
                  Demands,
                  Capacity,
                  Precedences,
                  activities,
                  makespan);

    /** Decompose the problem in a set of independent subproblems. */
    IloModelArray  subModels(env);
    DecomposeProblem(model, activities, makespan, subModels);

    /** Solve each sub-problem independently. */
    IloSchedulerSolution solution(env);
    IloInt nbSubProblems = subModels.getSize();
    for (IloInt i=0; i < nbSubProblems; i++) {
      IloModel subModel = subModels[i];
      SolveSubProblem(subModel, i, activities, solution,
                      i==nbSubProblems-1, makespan);
    }

    PrintSolution(activities, solution);
    solution.end();
    env.end();

  } catch (IloException& exc) {
    cout << exc << endl;
  }

  return 0;
}

////////////////////////////////////////////////////////////////////////////
//
// RESULTS
//
////////////////////////////////////////////////////////////////////////////

/* shipdec
   Activity 1    [0 -- 3 --> 3]
   Activity 2    [3 -- 4 --> 7]
   Activity 3    [7 -- 4 --> 11]
   Activity 4    [3 -- 6 --> 9]
```

```
   Activity 5    [14 -- 5 --> 19]
   Activity 6    [19 -- 2 --> 21]
   Activity 7    [11 -- 3 --> 14]
   Activity 8    [21 -- 4 --> 25]
   Activity 9    [25 -- 3 --> 28]
   Activity 10   [28 -- 2 --> 30]
   Activity 11   [32 -- 3 --> 35]
   Activity 12   [30 -- 2 --> 32]
   Activity 13   [35 -- 1 --> 36]
   Activity 14   [30 -- 5 --> 35]
   Activity 15   [36 -- 2 --> 38]
   Activity 16   [36 -- 3 --> 39]
   Activity 17   [39 -- 2 --> 41]
   Activity 18   [41 -- 2 --> 43]
   Activity 19   [44 -- 1 --> 45]
   Activity 20   [43 -- 1 --> 44]
   Activity 21   [43 -- 1 --> 44]
   Activity 22   [44 -- 2 --> 46]
   Activity 23   [46 -- 4 --> 50]
   Activity 24   [50 -- 5 --> 55]
   Activity 25   [55 -- 2 --> 57]
   Activity 26   [57 -- 1 --> 58]
   Activity 27   [58 -- 1 --> 59]
   Activity 28   [63 -- 2 --> 65]
   Activity 29   [65 -- 1 --> 66]
   Activity 30   [60 -- 3 --> 63]
   Activity 31   [59 -- 2 --> 61]
   Activity 32   [57 -- 1 --> 58]
   Activity 33   [58 -- 2 --> 60]
   Activity 34   [61 -- 2 --> 63]
*/
```

The decomposition dramatically decreases the number of failures to 38 and the CPU time to 0.2 seconds.

C H A P T E R

19

Moving Activities from Discrete to Unary Resources

A discrete resource of capacity *n* is often used in Scheduler to represent a set of *n* equivalent resources of capacity 1. For example, a team of *n* people with the same capabilities is represented by a discrete resource of capacity *n*.

One question you might ask is how to transform a schedule which satisfies the capacity constraint (that is, a schedule which guarantees that at any point in time, the number of resource units required by activities does not exceed *n*) into a schedule which specifies precisely which *unary resource* will contribute to the execution of which activity. For example, how can we transform the schedule of a group of *n* people, considered globally as a discrete resource of capacity *n*, into a schedule of *n* unary resources, each of which represents a specific person?

In this chapter, we see that a very simple algorithm is available for changing activities from discrete resources to unary resources.

Describing the Problem

Let’s reconsider the problem presented in Chapter 13, *More Advanced Problem Modeling with Concert Technology,* of assigning the activities of the house construction to three equivalent workers. Let's assume that we used a discrete resource of capacity three to

calculate a schedule. The start times that are found and the processing times of the activities are given in the following table. We already know that at any time `t` there are at most three activities executing. In this case, the algorithm we are going to develop is guaranteed to find a solution to the assignment problem under consideration.

Table 19.1 *Activity, Start Time, Processing Time*

Activity	Start Time	Processing Time
masonry	0	7
carpentry	7	3
plumbing	7	8
ceiling	7	3
roofing	10	1
painting	10	2
windows	11	1
facade	15	2
garden	15	1
moving	17	1

It is assumed that the data is provided in the form of four arrays.

- `ActivityNames`, containing the names of the ten activities.
- `WorkerNames`, containing the names of the three workers.
- `StartTimes`, containing the start times of the ten activities.
- `Processing Times`, containing the durations of the ten activities.

```
IloInt NumberOfActivities = 10;
IloInt NumberOfWorkers = 3;
const char* ActivityNames [] = {
  “masonry   “,
  “carpentry “,
  “plumbing  “,
  “ceiling   “,
  “roofing   “,
  “painting  “,
  “windows   “,
  “facade    “,
  “garden    “,
  “moving    “};
const char* WorkerNames [] = {
  “joe”,
  “jack”,
```

```
  “jim”};
IloInt StartTimes [] = { 0, 7, 7, 7, 10, 10, 11, 15, 15, 17 };
IloInt ProcessingTimes [] = { 7, 3, 8, 3,  1,  2,  1,  2,  1,  1 };
```

Solving the Problem

The following code defines a selector which selects the earliest activity to which no resource is assigned. This selector needs a visitor to visit all the activities of the environment, a predicate to filter the ones that are still not allocated, and an evaluator to select the activity with the smallest minimal start time.

```
// selector for non-assigned activity

ILOVISITOR0(ForAllActivitiesInEnv,
                                                                  IloActivity,
                                                                 IloEnv, env) {
  for(IloIterator<IloActivity> ite(env); ite.ok(); ++ite) {
    visit(*ite);
  }
}

ILOPREDICATE0(IsAllocatedPredicate,
                                                          IloActivity, act) {
  if (act.getObject() == 0)
    return IloFalse;
  else
    return IloTrue;
}

ILOEVALUATOR0(StartMinEvaluator,
                                                          IloActivity, act) {
  return act.getStartMin();
}

  IloPredicate<IloActivity> notAllocated = !IsAllocatedPredicate(env);
  IloSelector<IloActivity,IloEnv> selector =
    IloBestSelector<IloActivity,IloEnv>
    (ForAllActivitiesInEnv(env),
     notAllocated ,
     StartMinEvaluator(env).makeLessThanComparator());
```

Solving the problem consequently consists of iterating the following loop.

1. Select the next activity in chronological order.
2. Find a resource for this activity. For each resource, try to assign the resource to the activity. If the assignment succeeds, proceed to the next activity. If the assignment fails, proceed to the next resource for the activity under consideration.

The following function implements the algorithm. It returns `IloTrue` if the problem has a solution. Otherwise, it returns `IloFalse`. The function can return `IloFalse` only if there exists a time point `t` such that the number of activities executing at `t` exceeds the number of

unary resources that are available. In other words, the function returns `IloTrue` if the global resource capacity constraint is satisfied by the solution; it returns `IloFalse` if the global resource capacity constraint is not satisfied by the solution.

```
IloBool
SolveProblem(IloSolver solver, IloModel model) {
  IloEnv env = model.getEnv();
  IlcScheduler scheduler = IlcScheduler(solver);
  IloActivity act;

  IloPredicate<IloActivity> notAllocated = !IsAllocatedPredicate(env);
  IloSelector<IloActivity,IloEnv> selector =
    IloBestSelector<IloActivity,IloEnv>
    (ForAllActivitiesInEnv(env),
     notAllocated ,
     StartMinEvaluator(env).makeLessThanComparator());

  while ( selector.select(act, env) ) {
    for (IloIterator<IloUnaryResource> iterator(model.getEnv());
         notAllocated(act);
         ++iterator) {
      if (iterator.ok()) {
        IloUnaryResource res = *iterator;
        IloConstraint ct = act.requires(res);
        model.add(ct);

        if(solver.solve())
        {
          // store the assigned resource for the given activity
          act.setObject( (void*)(*iterator).getImpl() );
        }
        else {
          model.remove(ct);
        }
      }
      else
        return IloFalse;
    }
  }
  return IloTrue;
}
```

Complete Program and Output

You can see the entire program `assign.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

#if defined(ILO_SDXLOUTPUT)
#include “sdxloutput.h”
```

```
#endif

IloInt NumberOfActivities = 10;
IloInt NumberOfWorkers = 3;
const char* ActivityNames [] = {
  “masonry   “,
  “carpentry “,
  “plumbing  “,
  “ceiling   “,
  “roofing   “,
  “painting  “,
  “windows   “,
  “facade    “,
  “garden    “,
  “moving    “};
const char* WorkerNames [] = {
  “joe”,
  “jack”,
  “jim”};
IloInt StartTimes [] = { 0, 7, 7, 7, 10, 10, 11, 15, 15, 17 };
IloInt ProcessingTimes [] = { 7, 3, 8, 3,  1,  2,  1,  2,  1,  1 };

////////////////////////////////////////////////////////////////////////////
//
// MODEL DEFINITION
//
////////////////////////////////////////////////////////////////////////////

IloModel DefineModel(IloEnv env,
              IloInt numberOfActivities,
              IloInt numberOfWorkers,
              const char** activityNames,
              const char** workerNames,
              IloInt* startTimes,
              IloInt* processingTimes)
{
  IloModel model(env);

  /* CREATE THE WORKERS. */
  IloInt i;
  for (i = 0; i < numberOfWorkers; i++) {
    IloUnaryResource resource(env, workerNames[i]);
  }

  /* CREATE THE ACTIVITIES. */
  for (i = 0; i < numberOfActivities; i++) {
    IloActivity activity(env, processingTimes[i], activityNames[i]);
    activity.setStartTime(startTimes[i]);
  }

  return model;
}

////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////////////////
```

```
void
PrintSolution(IloSolver solver)
{
  IlcScheduler scheduler(solver);
  for (IloIterator<IloActivity> iterator(solver.getEnv());
       iterator.ok();
       ++iterator) {
    IloActivity act(*iterator);
    IloUnaryResource res =
      IloUnaryResource( (IloUnaryResourceI*)act.getObject() );

    solver.out() << scheduler.getActivity( act )
      << “\tassigned to “ << res.getName()
      << endl;
  }
}

////////////////////////////////////////////////////////////////////////////
//
// PROBLEM SOLVING
//
////////////////////////////////////////////////////////////////////////////
// selector for non-assigned activity

ILOVISITOR0(ForAllActivitiesInEnv,
                                                          IloActivity,
                                                        IloEnv, env) {
  for(IloIterator<IloActivity> ite(env); ite.ok(); ++ite) {
    visit(*ite);
  }
}

ILOPREDICATE0(IsAllocatedPredicate,
                                                    IloActivity, act) {
  if (act.getObject() == 0)
    return IloFalse;
  else
    return IloTrue;
}

ILOEVALUATOR0(StartMinEvaluator,
                                                    IloActivity, act) {
  return act.getStartMin();
}


IloBool
SolveProblem(IloSolver solver, IloModel model) {
  IloEnv env = model.getEnv();
  IlcScheduler scheduler = IlcScheduler(solver);
  IloActivity act;

  IloPredicate<IloActivity> notAllocated = !IsAllocatedPredicate(env);
  IloSelector<IloActivity,IloEnv> selector =
    IloBestSelector<IloActivity,IloEnv>
    (ForAllActivitiesInEnv(env),
     notAllocated ,
```

```
     StartMinEvaluator(env).makeLessThanComparator());

  while ( selector.select(act, env) ) {
    for (IloIterator<IloUnaryResource> iterator(model.getEnv());
         notAllocated(act);
         ++iterator) {
      if (iterator.ok()) {
        IloUnaryResource res = *iterator;
        IloConstraint ct = act.requires(res);
        model.add(ct);

        if(solver.solve())
        {
          // store the assigned resource for the given activity
          act.setObject( (void*)(*iterator).getImpl() );
        }
        else {
          model.remove(ct);
        }
      }
      else
        return IloFalse;
    }
  }
  return IloTrue;
}

////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////

int main()
{
  try {

    IloEnv env;
    IloModel model;

    // Model
    model = DefineModel(env,
      NumberOfActivities,
      NumberOfWorkers,
      ActivityNames,
      WorkerNames,
      StartTimes,
      ProcessingTimes);

    // Algorithm
    IloSolver solver(model);

    if (SolveProblem(solver, model)) {
      PrintSolution(solver);
#if defined(ILO_SDXLOUTPUT)
      IloSDXLOutput output(env);
      ofstream outFile("assign.xml");
      output.write(IlcScheduler(solver), outFile);
```

```
      outFile.close();
#endif
    }
    else
      solver.out() << “No Solution!” << endl;
    env.end();

  } catch (IloException& exc) {
    cout << exc << endl;
  }

  return 0;
}

////////////////////////////////////////////////////////////////////////////
//
// RESULTS
//
////////////////////////////////////////////////////////////////////////////

/* assign
   masonry    [0 -- 7 --> 7]        assigned to joe
   carpentry  [7 -- 3 --> 10]        assigned to joe
   plumbing   [7 -- 8 --> 15]        assigned to jack
   ceiling    [7 -- 3 --> 10]        assigned to jim
   roofing    [10 -- 1 --> 11]        assigned to joe
   painting   [10 -- 2 --> 12]        assigned to jim
   windows    [11 -- 1 --> 12]        assigned to joe
   facade     [15 -- 2 --> 17]        assigned to joe
   garden     [15 -- 1 --> 16]        assigned to jack
   moving     [17 -- 1 --> 18]        assigned to joe
*/
```

C H A P T E R

20

Working with Transition Times: the Job Shop Problem

There are many scheduling problems that include the definition of *transition times* between certain activities. Formally, we say that the transition time between two activities, A_1 and A_2, is the amount of time that must elapse between the end of A_1, and the beginning of A_2, when A_1 precedes A_2.

Transition times are highly problem-dependent, so in Scheduler you can define your own transition times by providing an instance of the class `IloTransitionTime`. This chapter shows you how to define the transition time function you want.

In this version of the job-shop problem, the activities have a transition type, and transition times are used on the resource when switching from one type of activity to another. The objective of the problem is to minimize the makespan. We use a dichotomizing binary-search strategy; this search algorithm is described in detail in Chapter 16, *A Dichotomizing Binary-Search Algorithm: the Job-Shop Problem*.

Defining Transition Types and Times

In this variant of the job-shop problem, each activity of each job has a given type, and each machine needs a transition time when it executes an activity that has a different type from the previously executed activity.

Arbitrarily, we decide the first activity of a given job has the type 0, the second one the type 1, and so on. To specify a type on an activity, the member function `IloActivity::setTransitionType` is used as follows.

```
   /* CREATE THE ACTIVITIES. */
  char buffer[128];
  k = 0;
  for (i = 0; i < numberOfJobs; i++) {
    IloActivity previousActivity;
    for (j = 0; j < numberOfResources; j++) {
      IloActivity activity(env, durations[k]);
      sprintf(buffer, "J%ldS%ldR%ld", i, j, resourceNumbers[k]);
      activity.setName(buffer);
      activity.setTransitionType(j);

      IloResourceConstraint rct =
        activity.requires(resources[resourceNumbers[k]]);
      NbUnrankedExt* nbOfUnranked =
        new (env) NbUnrankedExt();
      rct.setObject(nbOfUnranked);
      model.add(rct);

      if (j != 0)
        model.add(activity.startsAfterEnd(previousActivity));
      previousActivity = activity;
      k++;
    }
    model.add(previousActivity.endsBefore(makespan));
  }
```

The transition times between each type are stored in an array. For instance, for MT06, the array `TransTime06` is declared like this:

```
IloInt TransTimes06 [] = { 0,  2,  7,  3,  1,  3,
                           3,  0,  4,  7,  8,  8,
                           4,  2,  0,  5,  3,  6,
                           3,  6,  4,  0,  4,  7,
                           2,  1,  5,  5,  0,  1,
                           6,  4,  6,  9,  4,  0};
```

The elements of the array are used to initialize the instance of `IloTransitionParam`.

```
  /* CREATE THE TRANSITION TIMES */
  IloTransitionParam ttParam(env, numberOfResources, IloFalse);
  IloInt i, j;
  for (i = 0; i < numberOfResources; i++)
    for (j = 0; j < numberOfResources; j++)
      ttParam.setValue(i, j, transTimes[j + (numberOfResources * i)]);
```

Now we have to specify the transition times related to a given machine at its creation. An instance of an `IloTransitionTime` is created and it associates the resource with

`ttParam`, an instance of `IloTranstionParam`. The transition times on the resource will be computed using `ttParam` and the transition types of the activities

```
IloUnaryResource *resources =
  new (env) IloUnaryResource[numberOfResources];
for (j = 0; j < numberOfResources; j++) {
  IloUnaryResource res = IloUnaryResource(env);
  IloTransitionTime( res, ttParam );
  resources[j] = res;
}
```

The capacity enforcement and the precedence enforcement levels are set to `IloMediumHigh` so that more effort will be expended in enforcing the capacity constraints and transition times. This is not strictly necessary as transition times and capacity constraints will be taken into account at the default enforcement level, `IloBasic`. The problem is then solved using the dichotomizing search strategy discussed in Chapter 16; see that chapter for information on how that search algorithm can reduce the number of optimizing iterations and make the remaining iterations easier to compute.

Complete Program

You can see the entire program `jobshopt.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>
#include <ilsolver/iimls.h>

ILOSTLBEGIN

IloInt ResourceNumbers06 [] = {2, 0, 1, 3, 5, 4,
                               1, 2, 4, 5, 0, 3,
                               2, 3, 5, 0, 1, 4,
                               1, 0, 2, 3, 4, 5,
                               2, 1, 4, 5, 0, 3,
                               1, 3, 5, 0, 4, 2};

IloInt Durations06 [] = { 1,  3,  6,  7,  3,  6,
                          8,  5, 10, 10, 10,  4,
                          5,  4,  8,  9,  1,  7,
                          5,  5,  5,  3,  8,  9,
                          9,  3,  5,  4,  3,  1,
                          3,  3,  9, 10,  4,  1};
IloInt TransTimes06 [] = { 0,  2,  7,  3,  1,  3,
                           3,  0,  4,  7,  8,  8,
                           4,  2,  0,  5,  3,  6,
                           3,  6,  4,  0,  4,  7,
                           2,  1,  5,  5,  0,  1,
                           6,  4,  6,  9,  4,  0};

IloInt ResourceNumbers10 [] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
                               0, 2, 4, 9, 3, 1, 6, 5, 7, 8,
```

```
                                1, 0, 3, 2, 8, 5, 7, 6, 9, 4,
                                1, 2, 0, 4, 6, 8, 7, 3, 9, 5,
                                2, 0, 1, 5, 3, 4, 8, 7, 9, 6,
                                2, 1, 5, 3, 8, 9, 0, 6, 4, 7,
                                1, 0, 3, 2, 6, 5, 9, 8, 7, 4,
                                2, 0, 1, 5, 4, 6, 8, 9, 7, 3,
                                0, 1, 3, 5, 2, 9, 6, 7, 4, 8,
                                1, 0, 2, 6, 8, 9, 5, 3, 4, 7};

IloInt Durations10 [] = {29, 78,  9, 36, 49, 11, 62, 56, 44, 21,
                         43, 90, 75, 11, 69, 28, 46, 46, 72, 30,
                         91, 85, 39, 74, 90, 10, 12, 89, 45, 33,
                         81, 95, 71, 99,  9, 52, 85, 98, 22, 43,
                         14,  6, 22, 61, 26, 69, 21, 49, 72, 53,
                         84,  2, 52, 95, 48, 72, 47, 65,  6, 25,
                         46, 37, 61, 13, 32, 21, 32, 89, 30, 55,
                         31, 86, 46, 74, 32, 88, 19, 48, 36, 79,
                         76, 69, 76, 51, 85, 11, 40, 89, 26, 74,
                         85, 13, 61,  7, 64, 76, 47, 52, 90, 45};

IloInt TransTimes10 [] = {0, 9, 8, 1, 4, 3, 9, 2, 3, 7,
                          6, 0, 7, 5, 6, 1, 2, 1, 2, 3,
                          2, 3, 0, 3, 4, 3, 1, 9, 2, 2,
                          2, 7, 3, 0, 8, 7, 6, 4, 2, 1,
                          8, 3, 9, 8, 0, 5, 4, 9, 2, 3,
                          3, 2, 1, 3, 2, 0, 3, 2, 3, 9,
                          9, 8, 7, 4, 3, 8, 0, 6, 3, 4,
                          8, 7, 2, 1, 5, 3, 1, 0, 8, 2,
                          3, 9, 8, 5, 7, 1, 3, 2, 0, 9,
                          4, 8, 1, 2, 6, 4, 7, 2, 3, 0};

IloInt ResourceNumbers20 [] = {0, 1, 2, 3, 4,
                               0, 1, 3, 2, 4,
                               1, 0, 2, 4, 3,
                               1, 0, 4, 2, 3,
                               2, 1, 0, 3, 4,
                               2, 1, 4, 0, 3,
                               1, 0, 2, 3, 4,
                               2, 1, 0, 3, 4,
                               0, 3, 2, 1, 4,
                               1, 2, 0, 3, 4,
                               1, 3, 0, 4, 2,
                               2, 0, 1, 3, 4,
                               0, 2, 1, 3, 4,
                               2, 0, 1, 3, 4,
                               0, 1, 4, 2, 3,
                               1, 0, 3, 4, 2,
                               0, 2, 1, 3, 4,
                               0, 1, 4, 2, 3,
                               1, 2, 0, 3, 4,
                               0, 1, 2, 3, 4};

IloInt Durations20 [] = {29,  9, 49, 62, 44,
                         43, 75, 69, 46, 72,
                         91, 39, 90, 12, 45,
                         81, 71,  9, 85, 22,
                         14, 22, 26, 21, 72,
                         84, 52, 48, 47,  6,
```

```
                          46, 61, 32, 32, 30,
                          31, 46, 32, 19, 36,
                          76, 76, 85, 40, 26,
                          85, 61, 64, 47, 90,
                          78, 36, 11, 56, 21,
                          90, 11, 28, 46, 30,
                          85, 74, 10, 89, 33,
                          95, 99, 52, 98, 43,
                           6, 61, 69, 49, 53,
                           2, 95, 72, 65, 25,
                          37, 13, 21, 89, 55,
                          86, 74, 88, 48, 79,
                          69, 51, 11, 89, 74,
                          13,  7, 76, 52, 45};

IloInt TransTimes20 [] = {0, 3, 2, 9, 7,
                          3, 0, 2, 9, 5,
                          6, 4, 0, 5, 4,
                          1, 2, 3, 0, 7,
                          1, 9, 6, 4, 0};


////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////

class NbUnrankedExt {
private:
  IlcInt _nbOfUnranked;

public:
  NbUnrankedExt() :_nbOfUnranked(0) {};
  ~NbUnrankedExt(){};
  void setValue(IlcInt nbOfUnranked) {
    _nbOfUnranked = nbOfUnranked;
  }
  IlcInt getValue() const {
    return _nbOfUnranked;
  }
};


IloModel
DefineModel(IloEnv& env,
            IloInt numberOfJobs,
            IloInt numberOfResources,
            IloInt* resourceNumbers,
            IloInt* durations,
            IloInt* transTimes,
            IloNumVar& makespan)
{
  IloModel model(env);

  /* CREATE THE MAKESPAN VARIABLE. */
  IloInt numberOfActivities = numberOfJobs * numberOfResources;
  IloInt horizon = 0;
```

```
IloInt k;

IloInt maxTransTime = 0;
for (k = 0; k < numberOfResources*numberOfResources; k++)
  if (maxTransTime < transTimes[k])
    maxTransTime = transTimes[k];

for (k = 0; k < numberOfActivities; k++)
  horizon += durations[k] + maxTransTime;

makespan = IloNumVar(env, 0, horizon, ILOINT);

/* CREATE THE TRANSITION TIMES */
IloTransitionParam ttParam(env, numberOfResources, IloFalse);
IloInt i, j;
for (i = 0; i < numberOfResources; i++)
  for (j = 0; j < numberOfResources; j++)
    ttParam.setValue(i, j, transTimes[j + (numberOfResources * i)]);

/* CREATE THE RESOURCES. */
IloSchedulerEnv schedEnv(env);
schedEnv.getResourceParam().setCapacityEnforcement(IloMediumHigh);
schedEnv.getResourceParam().setPrecedenceEnforcement(IloMediumHigh);

IloUnaryResource *resources =
  new (env) IloUnaryResource[numberOfResources];
for (j = 0; j < numberOfResources; j++) {
  IloUnaryResource res = IloUnaryResource(env);
  IloTransitionTime( res, ttParam );
  resources[j] = res;
}

/* CREATE THE ACTIVITIES. */
char buffer[128];
k = 0;
for (i = 0; i < numberOfJobs; i++) {
  IloActivity previousActivity;
  for (j = 0; j < numberOfResources; j++) {
    IloActivity activity(env, durations[k]);
    sprintf(buffer, "J%ldS%ldR%ld", i, j, resourceNumbers[k]);
    activity.setName(buffer);
    activity.setTransitionType(j);

    IloResourceConstraint rct =
      activity.requires(resources[resourceNumbers[k]]);
    NbUnrankedExt* nbOfUnranked =
      new (env) NbUnrankedExt();
    rct.setObject(nbOfUnranked);
    model.add(rct);

    if (j != 0)
      model.add(activity.startsAfterEnd(previousActivity));
    previousActivity = activity;
    k++;
  }
  model.add(previousActivity.endsBefore(makespan));
}
```

```
  /* RETURN THE MODEL. */
  return model;
}

////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////////////////

void
PrintSolution(IloSolver& solver)
{
  IlcScheduler scheduler(solver);
  IloEnv env = solver.getEnv();
  for(IloIterator<IloActivity> ite(env); ite.ok(); ++ite)
    solver.out() << scheduler.getActivity(*ite) << endl;
}

////////////////////////////////////////////////////////////////////////////
//
// PROBLEM SOLVING
//
////////////////////////////////////////////////////////////////////////////

IlcInt GetNumberOfUnranked(const IlcResourceConstraint& rct) {

  /* RETURN NUMBER OF UNRANKED W.R.T. RCT */
  IlcInt nb = 0;
  for (IlcResourceConstraintIterator ite(rct, IlcUnranked);
       ite.ok(); ++ite)
    nb++;
  return nb;
}

IlcInt GetOpportunity(
                      const IlcScheduler& scheduler,
                      const IlcResourceConstraint& srct1,
                                  const IlcResourceConstraint& srct2) {

  IlcActivity act1 = srct1.getActivity();
  IlcActivity act2 = srct2.getActivity();

  IlcInt smin1 = act1.getStartMin();
  IlcInt smax1 = act1.getStartMax();
  IlcInt emin1 = act1.getEndMin();
  IlcInt emax1 = act1.getEndMax();
  IlcInt smin2 = act2.getStartMin();
  IlcInt smax2 = act2.getStartMax();
  IlcInt emin2 = act2.getEndMin();
  IlcInt emax2 = act2.getEndMax();

  /* DOMAIN DELTA WHEN RCT1 RANKED BEFORE RCT2 */
  IlcInt deltaEnd12   = ((emax1 < smax2) ? 0L : emax1 - smax2);
  IlcInt deltaStart12 = ((smin2 > emin1) ? 0L : emin1 - smin2);
  IlcInt delta12      = deltaEnd12 + deltaStart12;

  /* DOMAIN DELTA WHEN RCT2 RANKED BEFORE RCT1 */
```

```
  IlcInt deltaEnd21   = ((emax2 < smax1) ? 0L : emax2 - smax1);
  IlcInt deltaStart21 = ((smin1 > emin2) ? 0L : emin2 - smin1);
  IlcInt delta21      = deltaEnd21 + deltaStart21;

  /* MINIMAL NUMBER OF UNRANKED RESOURCE CONSTRAINTS */
  IlcInt nbUrkd1   =
((NbUnrankedExt*)(scheduler.getExtractable(srct1).getObject()))->getValue();
  IlcInt nbUrkd2   =
((NbUnrankedExt*)(scheduler.getExtractable(srct2).getObject()))->getValue();
  IlcInt minNbUrkd = (nbUrkd1 <= nbUrkd2) ? nbUrkd1 : nbUrkd2;

  /* RETURN MEASURE OF OPPORTUNITY */
  return (minNbUrkd * (delta12 - delta21));
}

IlcBool
SelectMostOpportunisticConflict(IlcScheduler& schedule,
                                IlcResourceConstraint& selectedRct1,
                                IlcResourceConstraint& selectedRct2) {

  IlcBool existsConflict = IlcFalse;

  IlcInt oppMaxAbs = -1;
  IlcInt oppMax = 0;
  IlcInt opp;

  for (IlcUnaryResourceIterator ires(schedule); ires.ok(); ++ires) {
    IlcUnaryResource resource = (*ires);
    if (resource.hasPrecedenceGraphConstraint() &&
        !resource.isRanked()) {

      /* FOR EACH RESOURCE CONSTRAINT, COMPUTE AND STORE THE NUMBER OF
         RESOURCE CONSTRAINTS UNRANKED W.R.T. IT */
      for (IlcResourceConstraintIterator irct(resource);
           irct.ok(); ++irct) {
        IlcResourceConstraint rct = (*irct);
        if (!rct.isRanked())
          ((NbUnrankedExt*)schedule.getExtractable(rct).getObject())->
        setValue(GetNumberOfUnranked(rct));
      }

      /* SELECT MOST OPPORTUNISTIC PAIR OF RESOURCE CONSTRAINT */
      for (IlcResourceConstraintIterator isrct1(resource);
           isrct1.ok(); ++isrct1) {
        IlcResourceConstraint srct1 = (*isrct1);
        if (!srct1.isRanked()) {
          for (IlcResourceConstraintIterator isrct2(srct1, IlcUnranked);
               isrct2.ok(); ++isrct2) {
            IlcResourceConstraint srct2 = (*isrct2);
      opp = GetOpportunity(schedule, srct1, srct2);
            if (oppMaxAbs < IloAbs(opp)) {
              existsConflict = IlcTrue;
              oppMaxAbs      = IlcAbs(opp);
              oppMax         = opp;
              selectedRct1   = srct1;
              selectedRct2   = srct2;
            }
          }
```

```
        }
      }
    }
  }

  /* SELECT WHICH BRANCH WILL BE CHOSEN FIRST AMONG RCT1 << RCT2 AND
     RCT2 << RCT1 */
  if (existsConflict && (0 < oppMax)) {
    IlcResourceConstraint tmpRct = selectedRct1;
    selectedRct1 = selectedRct2;
    selectedRct2 = tmpRct;
  }

  return existsConflict;
}

ILCGOAL0(SolveConflictsIlc) {
  IloSolver s = getSolver();
  IlcScheduler scheduler = IlcScheduler(s);

  IlcResourceConstraint srct1;
  IlcResourceConstraint srct2;
  if (SelectMostOpportunisticConflict(scheduler, srct1, srct2))
    return IlcAnd(IlcTrySetSuccessor(srct1, srct2), this);

  return 0;
}

ILOCPGOALWRAPPER0(SolveConflicts, solver) {
  return SolveConflictsIlc(solver);
}

void
SetMakespanInitialBounds(IloSolver& solver,
                         IloNumVar& makespan)
{
  /* SET MAKESPAN LOWER BOUND AND RESTART */
  solver.solve(IloGoalTrue(solver.getEnv()));
  makespan.setLB(solver.getMin(makespan));

  /* SOLVE WITH GOAL SOLVECONFLICTS */
  IloGoal solveConflicts = SolveConflicts(solver.getEnv());
  solver.solve(solveConflicts);

  /* SET MAKESPAN UPPER BOUND */
  makespan.setUB(solver.getMin(makespan));
  solver.out() << "Solution with makespan " << makespan.getUB() << endl;
}

void
Dichotomize(IloModel& model,
            IloSolver& solver,
            const IloNumVar& makespan)
{
  /* GET MAKESPAN INITIAL BOUNDS */
  IloNum min = makespan.getLB();
  IloNum max = makespan.getUB();
```

```
  /* OPTIMIZE. */
  IloGoal goal = IloRankForward(solver.getEnv(),
                                makespan,
                                IloSelResMinLocalSlack,
                                IloSelFirstRCMinStartMax);

  while(min < max) {
    IloNum value = IloFloor((min + max) / 2);
    IloConstraint ct = (makespan <= value);
    model.add(ct);

    if (solver.solve(goal)) {
      max = solver.getMin(makespan);
      solver.out() << "Solution with makespan " << max << endl;
    }
    else {
      solver.out() << "Failure with makespan " << value << endl;
      min = value + 1;
    }

    model.remove(ct);
  }

  /* RECOMPUTE THE OPTIMAL SOLUTION. THIS STEP COULD BE AVOIDED IF
     SOLUTIONS WERE STORED AS PART OF THE OPTIMIZATION PROCESS. */
  model.add(makespan == max);
  solver.solve(goal);
}

////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////

void
InitParameters(int argc,
               char** argv,
               IloInt& numberOfJobs,
               IloInt& numberOfResources,
               IloInt*& resourceNumbers,
               IloInt*& durations,
               IloInt*& transTimes)
{
  if (argc > 1) {
    IloInt number = atol(argv[1]);
    if (number == 10) {
      numberOfJobs = 10;
      numberOfResources = 10;
      resourceNumbers = ResourceNumbers10;
      durations = Durations10;
      transTimes = TransTimes10;
    }
    else if (number == 20) {
      numberOfJobs = 20;
      numberOfResources = 5;
      resourceNumbers = ResourceNumbers20;
      durations = Durations20;
```

```
      transTimes = TransTimes20;
    }
  }
}

int main(int argc, char** argv)
{
  try {
    IloEnv env;

    IloInt numberOfJobs = 6;
    IloInt numberOfResources = 6;
    IloInt* resourceNumbers = ResourceNumbers06;
    IloInt* durations = Durations06;
    IloInt* transTimes = TransTimes06;
    InitParameters(argc,
                   argv,
                   numberOfJobs,
                   numberOfResources,
                   resourceNumbers,
                   durations,
                   transTimes);
    IloNumVar makespan;
    IloModel model = DefineModel(env,
                                 numberOfJobs,
                                 numberOfResources,
                                 resourceNumbers,
                                 durations,
                                 transTimes,
                                 makespan);
    IloSolver solver(model);
    SetMakespanInitialBounds(solver, makespan);
    Dichotomize(model, solver, makespan);
    PrintSolution(solver);
    solver.printInformation();
    env.end();
  }
  catch (IloException& exc) {
    cout << exc << endl;
  }

  return 0;
}

////////////////////////////////////////////////////////////////////////////
//
// RESULTS
//
////////////////////////////////////////////////////////////////////////////

/* jobshopt 6
Solution with makespan 71
Failure with makespan 59
Solution with makespan 65
Failure with makespan 62
Failure with makespan 64
J0S0R2 [14..16 -- 1 --> 15..17]
J0S1R0 [15..17 -- 3 --> 18..20]
```

```
J0S2R1 [25..39 -- 6 --> 31..45]
J0S3R3 [34..45 -- 7 --> 41..52]
J0S4R5 [50..52 -- 3 --> 53..55]
J0S5R4 [59 -- 6 --> 65]
J1S0R1 [8 -- 8 --> 16]
J1S1R2 [17..21 -- 5 --> 22..26]
J1S2R4 [26 -- 10 --> 36]
J1S3R5 [36..38 -- 10 --> 46..48]
J1S4R0 [48..50 -- 10 --> 58..60]
J1S5R3 [58..60 -- 4 --> 62..64]
J2S0R2 [9..11 -- 5 --> 14..16]
J2S1R3 [14..17 -- 4 --> 18..21]
J2S2R5 [18..21 -- 8 --> 26..29]
J2S3R0 [35..37 -- 9 --> 44..46]
J2S4R1 [44..51 -- 1 --> 45..52]
J2S5R4 [52 -- 7 --> 59]
J3S0R1 [3 -- 5 --> 8]
J3S1R0 [8..12 -- 5 --> 13..17]
J3S2R2 [26..31 -- 5 --> 31..36]
J3S3R3 [31..36 -- 3 --> 34..39]
J3S4R4 [39 -- 8 --> 47]
J3S5R5 [54..56 -- 9 --> 63..65]
J4S0R2 [0..2 -- 9 --> 9..11]
J4S1R1 [18 -- 3 --> 21]
J4S2R4 [21 -- 5 --> 26]
J4S3R5 [31..34 -- 4 --> 35..38]
J4S4R0 [58..61 -- 3 --> 61..64]
J4S5R3 [62..64 -- 1 --> 63..65]
J5S0R1 [0 -- 3 --> 3]
J5S1R3 [3..9 -- 3 --> 6..12]
J5S2R5 [6..12 -- 9 --> 15..21]
J5S3R0 [25..27 -- 10 --> 35..37]
J5S4R4 [47 -- 4 --> 51]
J5S5R2 [51..64 -- 1 --> 52..65]
*/
```

C H A P T E R

21

Using Transition Times and Costs

Closely related to transition times are *transition costs*. In Scheduler, a transition cost is defined on a unary resource between two consecutive activities as the cost to switch the resource from processing the first activity to processing the second. Such costs could include adjusting or purging a machine, and may require manpower, material, or energy, for example.

In addition, Scheduler allows you to define a *setup cost* for the activity that starts the usage of the resource and a *teardown cost* for the activity that ends the usage of the resource.

Transition costs are highly problem-dependent, so in Scheduler you can define your own transition costs with an instance of the class `IloTransitionCost`. This chapter shows you how to define the transition cost function you want. You can add as many transition costs as are needed on a unary resource.

Scheduler associates an integer transition type with each activity. The accessors are the functions `IloActivity::getTransitionType` and `IloActivity::setTransitionType`. Transition types are used to index integer tables (transition tables), which are instances of the class `IloTransitionParam`. With the transition tables you can define instances of the classes `IloTransitionCost` and `IloTransitionTime`.

Problem Description

This problem is an extension of the job-shop problem where *n* jobs have to be performed using *m* machines. Each job consists of *m* successive activities such that each machine is required exactly once by the job. A maintenance operation must take place before each task and between each set of tasks. All maintenance operations have a duration and an energy that depend upon two successive operations. The maintenance duration is the transition time between two operations. The maintenance energy is the transition cost between two operations. The objective is to minimize the total transition cost.

Machines are unary resources.

Defining Job Activities

The activities of the jobs are of known duration and require a predefined machine. The jobs also define the temporal constraints between the activities. The temporal constraints are defined by `IloActivity::startsAfterEnd`.

```
const IloInt Horizon = 10000;
const IloInt NumberOfJobs = 6;
const IloInt NumberOfResources = 6;
const IloInt NumberOfWorkers = 5;
const IloInt NumberOfTypes = 4;

IloInt ResourceNumbers [] = {2, 0, 1, 3, 5, 4,
                             1, 2, 4, 5, 0, 3,
                             2, 3, 5, 0, 1, 4,
                             1, 0, 2, 3, 4, 5,
                             2, 1, 4, 5, 0, 3,
                             1, 3, 5, 0, 4, 2};

IloInt ProcessingTimes [] = { 10,  30,  60,  70,  30,  60,
                             80,  50, 100, 100, 100,  40,
                             50,  40,  80,  90,  10,  70,
                             50,  50,  50,  30,  80,  90,
                             90,  30,  50,  40,  30,  10,
                             30,  30,  90, 100,  40,  10};
```

Each of these activities has a transition type.

```
IloInt Types [] = { 0, 1, 2, 2, 0, 3,
                    1, 3, 2, 3, 3, 0,
                    2, 3, 1, 1, 0, 1,
                    2, 1, 0, 0, 1, 1,
                    3, 3, 2, 0, 1, 1,
                    3, 1, 2, 0, 3, 3};
```

Creating Transition Tables

In this model, a transition table defines the duration of the maintenance operations. This table is used to build both the duration of the maintenance operations as a transition cost and the transition time between two activities on a machine. An array of integers is given for the duration of the setup operation on each machine.

```
IloInt TableDurations [] = { 0, 16, 14, 19,
                             12, 0, 8, 15,
                             15, 10, 0, 14,
                             16, 15, 17, 0};

IloInt SetupDurations [] = { 7, 0, 6, 7};
```

A transition cost table and a setup cost array are also defined for maintenance or setup operation. We call this the transition capacity.

```
IloInt TableCapacities [] = { 0, 2, 3, 1,
                              2, 0, 4, 2,
                              3, 3, 0, 2,
                              1, 2, 3, 0};

IloInt SetupCapacities [] = { 2, 1, 1, 1};
```

In order to select the successor of an activity on a machine, we define a transition cost, defined as the energy required by the maintenance operation, that is the product between the transition time and the transition capacity.

```
/* CREATE THE TRANSITION FUNCTIONS. */
enrTransParam = IloTransitionParam(env, numberOfTypes, IloFalse);
IloTransitionParam durTransParam(env, numberOfTypes, IloFalse);
for(i = 0; i < numberOfTypes; ++i) {
  enrTransParam.setSetup(i, setupDurations[i] * setupCapacities[i]);
  IloInt index = i * numberOfTypes;
  for(j = 0 ; j < numberOfTypes; ++j) {
    durTransParam.setValue(i, j, tableDurations[index]);
    enrTransParam.setValue(i, j,
                    (tableDurations[index] * tableCapacities[index]));
    ++index;
  }
}
```

Setting Temporal Constraints

An activity in a job starts after the end of the preceding activity in the same job. Note that the maintenance operations are not explicitly modeled as an `IloActivity`. Rather, the transition time and transition cost are used to ensure that there is enough time and energy to execute the activity without explicitly representing the activity in the model.

```
/* CREATE THE ACTIVITIES AND THE TEMPORAL CONSTRAINTS. */
```

```
  k = 0;
  for (i = 0; i < numberOfJobs; i++) {
    IloActivity previousTask;
    for (j = 0; j < numberOfResources; j++) {
      IloInt number = resourceNumbers[k];
      sprintf(buffer, "J%ldS%ldR%ldT%ld", i, j, number, types[k]);
      IloActivity task = MakeTask(model,
                                  resources[number],
                                  types[k],
                                  processingTimes[k],
                                  setupDurations[types[k]],
                                  buffer);
      if (j != OL)
        model.add(task.startsAfterEnd(previousTask));
      previousTask = task;
      k++;
    }
    model.add(previousTask.endsBefore(makespan));
  }
}
```

Defining Machines

A machine is created as a unary resource, with a transition cost and transition time.

```
/* CREATION OF A MACHINE WITH ITS TRANSITION TIME AND COSTS. */
IloUnaryResource
MakeMachine(IloModel model,
            IloTransitionParam durTransParam,
            IloTransitionParam enrTransParam,
            IloNumVar cost,
            const char* name) {
  IloUnaryResource machine(model.getEnv(), name);
  IloTransitionTime durTrans(machine, durTransParam);
  IloTransitionCost enrTrans(machine, enrTransParam);
  model.add(enrTrans.getCostSumVar() <= cost);
  return machine;
}
```

Creating the Activities

The activities each require a unary resource and are of a specified `type`. The transition times and costs are based on the `type` of each activity.

```
/* CREATION OF A TASK */
IloActivity
MakeTask(IloModel model,
         IloUnaryResource machine,
         IloInt type,
         IloInt procTime,
         IloInt setupTime,
         const char* name) {
  IloActivity task(model.getEnv(), procTime, type, name);
  task.setStartMin(setupTime);
```

```
  model.add(task.requires(machine));
  return task;
}
```

Solving the Problem

To search for a solution, we must sequence each machine in the problem. The goal `IloSequenceForward` is provided by Scheduler for this purpose. To solve the problem, we instantiate the makespan (the total energy cost).

```
IloSolver solver(model);
IloGoal goal = IloSequenceForward(env, cost, enrTransParam) &&
  IloInstantiate(env, makespan) &&
  IloSetTimesForward(env);

if (solver.solve(goal)) {
  env.out() << " Solution with " << endl
            << "\tenergy : " << solver.getIntVar(cost) << endl
            << "\tmakespan : " << solver.getIntVar(makespan) << endl;
  PrintSolution(solver);
}
else
  solver.out() << "No Solution" << endl;
```

Complete Program

You can see the entire program `tcost.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

const IloInt Horizon = 10000;
const IloInt NumberOfJobs = 6;
const IloInt NumberOfResources = 6;
const IloInt NumberOfWorkers = 5;
const IloInt NumberOfTypes = 4;

IloInt ResourceNumbers [] = {2, 0, 1, 3, 5, 4,
                             1, 2, 4, 5, 0, 3,
                             2, 3, 5, 0, 1, 4,
                             1, 0, 2, 3, 4, 5,
                             2, 1, 4, 5, 0, 3,
                             1, 3, 5, 0, 4, 2};

IloInt ProcessingTimes [] = { 10,  30,  60,  70,  30,  60,
                       80,  50, 100, 100, 100,  40,
                       50,  40,  80,  90,  10,  70,
```

```
                        50,  50,  50,  30,  80,  90,
                        90,  30,  50,  40,  30,  10,
                        30,  30,  90, 100,  40,  10};
IloInt Types [] = { 0, 1, 2, 2, 0, 3,
                    1, 3, 2, 3, 3, 0,
                    2, 3, 1, 1, 0, 1,
                    2, 1, 0, 0, 1, 1,
                    3, 3, 2, 0, 1, 1,
                    3, 1, 2, 0, 3, 3};
IloInt TableDurations [] = { 0, 16, 14, 19,
                             12, 0, 8, 15,
                             15, 10, 0, 14,
                             16, 15, 17, 0};

IloInt SetupDurations [] = { 7, 0, 6, 7};
IloInt TableCapacities [] = { 0, 2, 3, 1,
                              2, 0, 4, 2,
                              3, 3, 0, 2,
                              1, 2, 3, 0};

IloInt SetupCapacities [] = { 2, 1, 1, 1};
////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////
/* CREATION OF A MACHINE WITH ITS TRANSITION TIME AND COSTS. */
IloUnaryResource
MakeMachine(IloModel model,
            IloTransitionParam durTransParam,
            IloTransitionParam enrTransParam,
            IloNumVar cost,
            const char* name) {
  IloUnaryResource machine(model.getEnv(), name);
  IloTransitionTime durTrans(machine, durTransParam);
  IloTransitionCost enrTrans(machine, enrTransParam);
  model.add(enrTrans.getCostSumVar() <= cost);
  return machine;
}
/* CREATION OF A TASK */
IloActivity
MakeTask(IloModel model,
         IloUnaryResource machine,
         IloInt type,
         IloInt procTime,
         IloInt setupTime,
         const char* name) {
  IloActivity task(model.getEnv(), procTime, type, name);
  task.setStartMin(setupTime);
  model.add(task.requires(machine));
  return task;
}
void
DefineProblem(IloModel model,
              IloInt horizon,
              IloInt numberOfJobs,
              IloInt numberOfResources,
              IloInt numberOfWorkers,
```

```
            IloInt numberOfTypes,
            IloInt* resourceNumbers,
            IloInt* processingTimes,
            IloInt* types,
            IloInt* tableDurations,
            IloInt* tableCapacities,
            IloInt* setupDurations,
            IloInt* setupCapacities,
            IloNumVar& makespan,
            IloNumVar& cost,
            IloTransitionParam& enrTransParam) {
/* CREATE THE MAKESPAN VARIABLE. */
IloEnv env = model.getEnv();
makespan = IloNumVar(env, 0, horizon, IloNumVar::Int);
/* CREATE THE COST VARIABLE. */
IloInt maxEnergy = horizon * numberOfWorkers;
cost = IloNumVar(env, 0, maxEnergy, IloNumVar::Int);
char buffer[128];
IloInt i, j, k;
/* CREATE THE TRANSITION FUNCTIONS. */
enrTransParam = IloTransitionParam(env, numberOfTypes, IloFalse);
IloTransitionParam durTransParam(env, numberOfTypes, IloFalse);
for(i = 0; i < numberOfTypes; ++i) {
  enrTransParam.setSetup(i, setupDurations[i] * setupCapacities[i]);
  IloInt index = i * numberOfTypes;
  for(j = 0 ; j < numberOfTypes; ++j) {
    durTransParam.setValue(i, j, tableDurations[index]);
    enrTransParam.setValue(i, j,
                    (tableDurations[index] * tableCapacities[index]));
    ++index;
  }
}
/* CREATE THE RESOURCES. */
IloUnaryResource* resources =
  new (env) IloUnaryResource[numberOfResources];
for (j = 0; j < numberOfResources; j++) {
  sprintf(buffer, "Machine%ld", j);
  resources[j] = MakeMachine(model, durTransParam, enrTransParam,
                             cost, buffer);
}
/* CREATE THE ACTIVITIES AND THE TEMPORAL CONSTRAINTS. */
k = 0;
for (i = 0; i < numberOfJobs; i++) {
  IloActivity previousTask;
  for (j = 0; j < numberOfResources; j++) {
    IloInt number = resourceNumbers[k];
    sprintf(buffer, "J%ldS%ldR%ldT%ld", i, j, number, types[k]);
    IloActivity task = MakeTask(model,
                                resources[number],
                                types[k],
                                processingTimes[k],
                                setupDurations[types[k]],
                                buffer);
    if (j != OL)
      model.add(task.startsAfterEnd(previousTask));
    previousTask = task;
    k++;
  }
```

```
    model.add(previousTask.endsBefore(makespan));
  }
}

////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////////////////

void PrintSolution(IloSolver& solver) {
  IlcScheduler schedule(solver);
  for(IlcUnaryResourceIterator iter(schedule); iter.ok(); ++iter) {
    IlcUnaryResource machine = *iter;
    solver.out() << endl << "Machine : " << machine << endl;
    IloInt index = machine.getSetupVar().getValue();
    IloInt sink = machine.getTeardownIndex();
    while(index != sink) {
      IlcResourceConstraint task = machine.getSequenceRC(index);
      solver.out() << "\t" << task.getActivity() << endl;
      index = task.getNextVar().getValue();
    }
  }
}

int main() {
  try {
    IloEnv env;
    IloModel model(env);
    IloNumVar makespan;
    IloNumVar cost;
    IloTransitionParam enrTransParam;
    DefineProblem(model,
                  Horizon,
                  NumberOfJobs,
                  NumberOfResources,
                  NumberOfWorkers,
                  NumberOfTypes,
                  ResourceNumbers,
                  ProcessingTimes,
                  Types,
                  TableDurations,
                  TableCapacities,
                  SetupDurations,
                  SetupCapacities,
                  makespan,
                  cost,
                  enrTransParam);

    IloSolver solver(model);
    IloGoal goal = IloSequenceForward(env, cost, enrTransParam) &&
      IloInstantiate(env, makespan) &&
      IloSetTimesForward(env);

    if (solver.solve(goal)) {
      env.out() << " Solution with " << endl
                << "\tenergy : " << solver.getIntVar(cost) << endl
                << "\tmakespan : " << solver.getIntVar(makespan) << endl;
```

```
      PrintSolution(solver);
    }
    else
      solver.out() << "No Solution" << endl;
    env.end();
  } catch (IloException& exc) {
    cout << exc << endl;
  }
  return 0;
}

////////////////////////////////////////////////////////////////////////////
//
// RESULTS
//
////////////////////////////////////////////////////////////////////////////

/* tcost
  Solution with
        energy : [168]
        makespan : [971]

  Machine : Machine4 [1]
        J4S2R4T2 [450 -- 50 --> 500]
        J1S2R4T2 [500 -- 100 --> 600]
        J3S4R4T1 [610 -- 80 --> 690]
        J2S5R4T1 [756 -- 70 --> 826]
        J5S4R4T3 [841 -- 40 --> 881]
        J0S5R4T3 [881 -- 60 --> 941]

  Machine : Machine5 [1]
        J5S2R5T2 [450 -- 90 --> 540]
        J2S2R5T1 [550 -- 80 --> 630]
        J0S4R5T0 [642 -- 30 --> 672]
        J4S3R5T0 [672 -- 40 --> 712]
        J1S3R5T3 [731 -- 100 --> 831]
        J3S5R5T1 [846 -- 90 --> 936]

  Machine : Machine3 [1]
        J2S1R3T3 [56 -- 40 --> 96]
        J0S3R3T2 [326 -- 70 --> 396]
        J5S1R3T1 [420 -- 30 --> 450]
        J3S3R3T0 [476 -- 30 --> 506]
        J4S5R3T1 [776 -- 10 --> 786]
        J1S5R3T0 [931 -- 40 --> 971]

  Machine : Machine1 [1]
        J1S0R1T1 [0 -- 80 --> 80]
        J0S2R1T2 [266 -- 60 --> 326]
        J3S0R1T2 [326 -- 50 --> 376]
        J5S0R1T3 [390 -- 30 --> 420]
        J4S1R1T3 [420 -- 30 --> 450]
        J2S4R1T0 [746 -- 10 --> 756]

  Machine : Machine0 [1]
        J0S1R0T1 [236 -- 30 --> 266]
        J3S1R0T1 [376 -- 50 --> 426]
        J5S3R0T0 [540 -- 100 --> 640]
```

```
        J2S3R0T1 [656 -- 90 --> 746]
        J4S4R0T1 [746 -- 30 --> 776]
        J1S4R0T3 [831 -- 100 --> 931]

  Machine : Machine2 [1]
        J2S0R2T2 [6 -- 50 --> 56]
        J4S0R2T3 [70 -- 90 --> 160]
        J1S1R2T3 [160 -- 50 --> 210]
        J0S0R2T0 [226 -- 10 --> 236]
        J3S2R2T0 [426 -- 50 --> 476]
        J5S5R2T3 [881 -- 10 --> 891]
*/
```

C H A P T E R

22

Using Strong Propagation on Reservoirs: the Balance Constraint

This example illustrates how to create a *balance constraint*, what kind of propagation it performs, and how the balance constraint communicates with other constraints.

Describing the Problem

Suppose we have a scheduling problem that consists in scheduling n activities with different durations on a single unary resource.

In a solution, all the activities are ordered on the unary resource and, therefore, we can define an integer that describes the position of an activity in this solution (see Figure 22.1).

Suppose also that we have a set of additional constraints that state some minimal delays between the activity scheduled at position i and the one scheduled at position j (regardless of which activities will be scheduled at these positions). More precisely, we have a set of m additional constraints $C(i, j, d)$ where i, j in $[0, n]$, $i < j$, $d > 0$. Let's suppose that in a solution, the activities are ranked at positions $0, \ldots, n-1$. Constraint $C(i, j, d)$ states that between the start of the activity ranked at position i and the start of the activity ranked at position j, there should be at least a delay d.

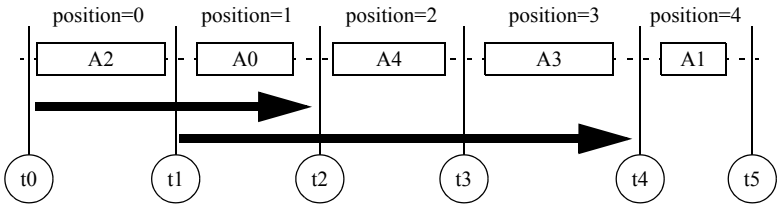


Figure 22.1 *Positions of activities on a unary resource*

The problem consists of finding an order between the activities on the unary resource that satisfies all constraints, *C*(*i*, *j*, *d*), and minimizes the makespan.

Defining the Problem, Designing a Model

Suppose we introduce a set of time variables *t0, …, tn* such that *ti* is constrained to execute between the end of the activity ranked at position *i-1* and the start of the activity ranked at position *i* as described in Figure 22.1.

The constraint *C*(*i*, *j*, *d*) can be expressed by (*tj* - *ti* >= *d*). Of course, the problem is that one doesn't know the actual activity that is ranked at position *i* until the activities have been ranked. Therefore, we have to implicitly constrain the time points *ti* to be between the activity ranked at position *i-1* and the one ranked at position *i*. This can easily be done with an additional resource that is a reservoir of capacity 2 and initial level 0. Each real activity *ak* on the unary resource will consume 1 unit of this reservoir at its start time and consume 1 unit of reservoir at its end time. We introduce a chain of *n* fake activities, *pi*, that produce 2 units of reservoir at their start time *ti*. The duration of these fake activities is chosen to be 1 but it could be any duration. The chain of fake activities is defined by taking *ti* < *tj*. This model is shown in Figure 22.2.

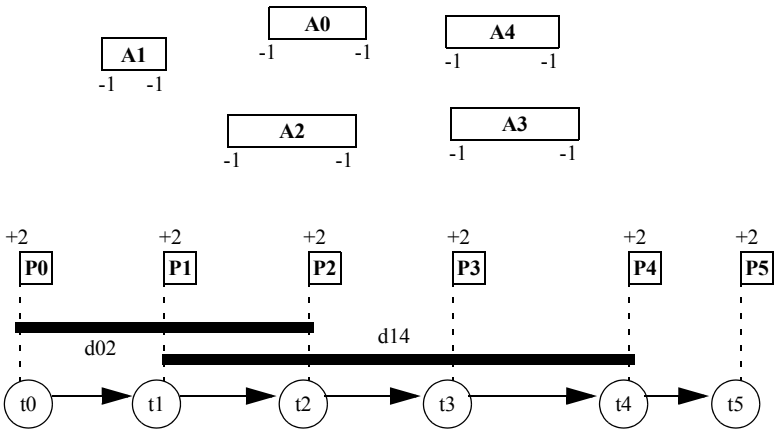


Figure 22.2 *Scheduling Model*

We see that, due to the minimal and maximal level on the reservoir, between *ti* and *ti+1* there must be at least two resource consumptions. Because of the unary resource, we also see that these two resource consumptions must correspond to the start of an activity *ak* followed by its end. Thus *ti*, the start time of *pi*, is really a time-point between the activity ranked at position *i-1* and the one ranked at position *i*. We can post the constraints (*tj - ti >= d*) directly as temporal constraints between the fake activities *pi* on the reservoir.

The model is implemented as follows. As usual, a function creates and returns the model:

```
IloModel
DefineModel(const IloEnv& env,
            IloInt numberOfActivities,
            IloInt numberOfConstraints,
            IloInt* durations,
            IloInt* constraints,
            IloNumVar& makespan)
{
  IloModel model(env);
```

First of all, an upper bound of the optimal makespan is computed and used to define the range of the makespan variable. This upper bound is computed as follows: suppose that there exists an optimal solution *Sopt*, that is, a total order between the activities *ak* (similar to the one described on Figure 22.1) that satisfies the temporal constraints between the time points *ti*. For simplicity, we assume *Sopt = (a0, a1, …, an-1)*. It's easy to show that the optimal makespan corresponding to this total order is the minimal value of *tn* that satisfies:

- for all *i*: *ti-1 <= start(ai) < end(ai) <= ti*
- for all constraints *C*(*i*, *j*, *d*): *tj - ti >= d*

In other words, given a feasible total order, the optimal makespan can be easily computed simply by propagating the PERT associated with the schedule and is equal to the minimal value of *tn* after propagation. If we use a more pessimistic version of this PERT where all activities have the same duration, that is, the one of the longest activity *ak*, we will obtain a worse makespan. This makespan, however, will be an upper bound of the optimal makespan obtained for any total order between activities. That's the idea of the function `getMakespanMax` below.

```
  IloNum makespanMax = getMakespanMax(env,
                                      numberOfActivities,
                                      numberOfConstraints,
                                      durations,
                                      constraints);
  makespan = IloNumVar(env, 0, makespanMax, ILOINT);

IloNum getMakespanMax(const IloEnv& env,
                      IloInt numberOfActivities,
                      IloInt numberOfConstraints,
                      IloInt* durations,
                      IloInt* constraints) {
  // THIS FUNCTION COMPUTES AN UPPER BOUND ON THE MAKESPAN BY ASSUMING
  // ALL THE ACTIVITIES HAVE A  DURATION EQUAL TO THE LONGEST ACTIVITY
  // OF THE   PROBLEM.
  IloModel model(env);

  IloNumVar makespan(env, 0, IloInfinity, ILOINT);
  IloInt i, j, k, t;

  // COMPUTE MAXIMAL DURATION OF ACTIVITIES
  IloInt dmax = 0;
  for (i=0; i < numberOfActivities; i++)
    if (durations[i] > dmax)
      dmax = durations[i];

  IloActivity lact;
  IloNumExprArray times(env, numberOfActivities+1);
  for (i=0; i < numberOfActivities; i++) {
    // USE ACTIVITIES WITH MAXIMAL DURATION
    IloActivity act(env, dmax);
    times[i] = act.getStartExpr();
    if (i > 0)
      model.add(act.startsAfterEnd(lact));
    lact = act;
  }
  model.add(lact.endsBefore(makespan));
  times[numberOfActivities] = makespan;

  // ADD MINIMAL DELAY CONSTRAINTS
  for (k=0; k < numberOfConstraints; k++) {
    i = constraints[3*k];
    j = constraints[(3*k) + 1];
    t = constraints[(3*k) + 2];
    model.add(times[j]-times[i] >= t);
  }

  // JUST PROPAGATE THE PERT
```

```
  IloSolver solver(model);
  IlcSearch s = solver.newSearch(IloGoalTrue(env));
  s.next();
  IloNum makespanMax = solver.getMin(makespan);
  solver.end();

  return makespanMax;
}
```

Now that the range of the makespan variable has been defined, we need to define the main ingredients of the model itself. As mentioned above, we need to define two resources: the unary resource that will impose a total order between the activities *ak* and the reservoir that will implicitly constrain the time points *ti* to be in between the activities *ak*.

In this scheduling problem, we see that our primary reasoning concerns the relative position of activities rather than the absolute dates at which activities execute. As we noticed before, once a total order between activities *ak* has been found, computing the corresponding optimal makespan is easy. Thus, the main problem is finding an optimal total order and we need to use propagation algorithms that perform strong pruning of the search by analyzing the *relative* positions of activities.

On the reservoir, we will post a balance constraint. This constraint is described in *Balance Constraint* in the *IBM ILOG Scheduler Reference Manual*. The balance constraint internally maintains a precedence graph between the time points (start, end) of the activities on the resource. It propagates by analyzing this graph in order to discover new time bounds for activities as well as new precedence relations between activity time points. In the Concert Technology model, the balance constraint is specified by the level `IloExtended` for the capacity enforcement as described in *Resource Enforcement as Global Constraint Declaration* in the *IBM ILOG Scheduler Reference Manual*.

```
  IloReservoir reserv(env, 2, 0);
  // THIS ENFORCEMENT LEVEL WILL POST THE BALANCE CONSTRAINT
  reserv.setCapacityEnforcement(IloExtended);
```

On the unary resource we will also post a balance constraint as well as a precedence graph constraint. The precedence graph constraint will collect all the new precedences detected by the disjunctive constraint and the edge-finder on the unary resource as well as the rank information on activities. The balance constraint on the unary resource will get this precedence information from the precedence graph and also exchange information with the balance constraint of the reservoir.

```
  IloUnaryResource ures(env);
  // THIS ENFORCEMENT LEVEL WILL POST THE BALANCE CONSTRAINT
  ures.setCapacityEnforcement(IloExtended);
  // THIS ENFORCEMENT LEVEL WILL POST THE PRECEDENCE GRAPH CONSTRAINT
  ures.setPrecedenceEnforcement(IloExtended);
```

This combination of global constraints ensures that whenever a total order between activities *ak* on the unary resource has been found, the balance constraint on the reservoir knows that

the set of reservoir consumption (at the start and end of activities *ak*) is totally ordered and this implies that the fake reservoir productions at time points *ti* are correctly interleaved between the activities *ak*. Thus, it ensures the soundness of the search, even though the search goal `IloRankForward` (see Solving the Problem) does not completely instantiate the start and end times of activities. Of course, during the search, the balance constraint on the reservoir may discover new precedence relations between activities. These precedence relations are automatically inserted in the unary resource precedence graph via the balance constraint of the unary resource and they enable reduction of the search tree by removing some possibly first resource constraint on the unary resource. The global communication between constraints is illustrated on Figure 22.3.

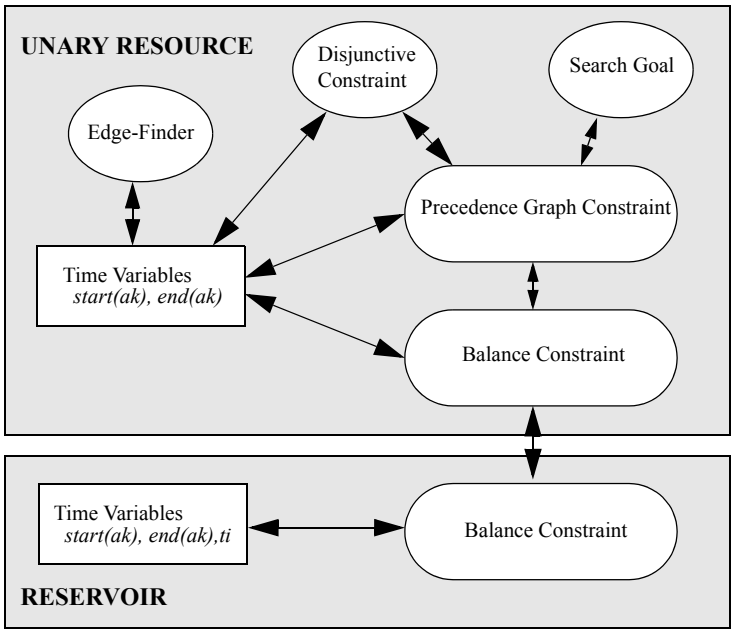


Figure 22.3 *Communication Among Global Constraints*

The rest of the definition of the model creates the activities and the resource constraints. Note how some evident symmetries of the problem are broken by ordering the subsets of activities of the same duration before solving. Note also that a delay *dmin* corresponding to the minimal duration of activities is inserted between each time point *ti*.

```
IloNumExprArray times(env, numberOfActivities + 1);
IloActivityArray acts(env, numberOfActivities);

// COMPUTE DURATION OF SHORTEST ACTIVITY
IloInt dmin = IloIntMax;
IloInt i, j, k, t;
```

```
for (i=0; i < numberOfActivities; i++)
  if (durations[i] < dmin)
    dmin = durations[i];

IloActivity lprod;
for (i=0; i < numberOfActivities; i++) {
  acts[i]= IloActivity(env, durations[i]);
  model.add(acts[i].requires(ures));
  model.add(acts[i].requires(reserv, 1, IloAfterEnd));
  model.add(acts[i].requires(reserv, 1, IloAfterStart));
  model.add(acts[i].endsBefore(makespan));
  IloActivity prod(env, 1);
  model.add(prod.provides(reserv, 2, IloAfterStart));
  times[i] = prod.getStartExpr();
  if (i > 0)
    // BUILD CHAIN OF PRODUCERS
    model.add(prod.startsAfterStart(lprod, dmin));
  lprod = prod;
}
times[numberOfActivities] = makespan;
model.add(times[numberOfActivities-1] + dmin <= makespan);

// BREAK EVIDENT SYMMETRIES BETWEEN ACTIVITIES WITH SAME DURATION
for (i=0; i < numberOfActivities - 1; i++)
  for (j=i+1; j < numberOfActivities; j++)
    if (durations[i] == durations[j]) {
      model.add(acts[j].startsAfterEnd(acts[i]));
      break;
    }

// ADD DELAY CONSTRAINTS
for (k=0; k < numberOfConstraints; k++) {
  i = constraints[3*k];
  j = constraints[(3*k) + 1];
  t = constraints[(3*k) + 2];
  model.add(times[j]-times[i] >= t);
}

return model;
```

Solving the Problem

Once the model has been created, we solve it using the predefined goal `IloRankForward` until an optimal schedule is found. Each solution found in the optimization process is displayed. For each solution, the slack time corresponding to the cumulative waiting time between activities is also displayed. Note that minimizing the makespan is the same as minimizing this slack time as the slack is equal to the makespan minus the total duration of activities.

```
IloNumVar makespan;
IloModel model = DefineModel(env,
                             numberOfActivities,
                             numberOfConstraints,
```

```
                                durations,
                                constraints,
                                makespan);

    model.add(IloMinimize(env, makespan));
    IloSolver solver(model);
    IlcSearch search = solver.newSearch(IloRankForward(env, makespan));

    IloInt sumd = 0;
    for (IloInt i=0; i < numberOfActivities; i++)
      sumd += durations[i];

    while (search.next()) {
      solver.out() << "SOLUTION WITH SLACK = "
                   << solver.getIntVar(makespan).getMin() - sumd << endl;
      PrintSolution(solver);
    }

void
PrintSolution(const IloSolver& solver)
{
  IlcScheduler sched(solver);
  IlcUnaryResource res = *(IlcUnaryResourceIterator(sched));
  for(IlcResourceConstraintIterator ite(res); ite.ok(); ++ite)
    solver.out() << (*ite).getActivity() << endl;
}
```

The complete program allows the solving of different predefined problems. The argument given to the executable corresponds to the instance of the problem to solve.

Complete Program and Output

You can see the entire program `balpos.cpp` here or view it in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

IloInt NumberOfActivities = 15;

//--------------------------- PROBLEM 1 -----------------------------------
IloInt Durations01 [] = {
  5, 5, 5, 5, 5, 7, 7, 8, 8, 9, 14, 14, 14, 14, 15 };

IloInt NumberOfConstraints01 = 7;

IloInt Constraints01 [] = {
  14, 15, 5,
  4,  6,  24,
  0,  3,  38,
  1,  11, 117,
  1,  8,  83,
  4,  8,  50,
  8,  14, 59 };
```

```
//--------------------------- PROBLEM 2 ----------------------------------
IloInt Durations02 [] = {
  5, 7, 7, 8, 8, 8, 8, 10, 10, 10, 11, 11, 13, 14, 15 };

IloInt NumberOfConstraints02 = 7;

IloInt Constraints02 [] = {
  3, 11, 88,
  6, 8,  27,
  1, 6,  48,
  5, 8,  38,
  9, 15, 55,
  1, 3,  17,
  1, 4,  27 };

//--------------------------- PROBLEM 3 ----------------------------------
IloInt Durations03 [] = {
  5, 6, 6, 7, 7, 7, 8, 9, 9, 9, 10, 11, 14, 14, 15 };

IloInt NumberOfConstraints03 = 7;

IloInt Constraints03 [] = {
  2, 3, 15,
  5, 7, 17,
  1, 7, 71,
  2, 5, 44,
  2, 6, 49,
  2, 4, 29,
  0, 3, 34 };


//----------------------- COMPUTE MAX MAKESPAN ---------------------------
IloNum getMakespanMax(const IloEnv& env,
                      IloInt numberOfActivities,
                      IloInt numberOfConstraints,
                      IloInt* durations,
                      IloInt* constraints) {
  // THIS FUNCTION COMPUTES AN UPPER BOUND ON THE MAKESPAN BY ASSUMING
  // ALL THE ACTIVITIES HAVE A  DURATION EQUAL TO THE LONGEST ACTIVITY
  // OF THE   PROBLEM.
  IloModel model(env);

  IloNumVar makespan(env, 0, IloInfinity, ILOINT);
  IloInt i, j, k, t;

  // COMPUTE MAXIMAL DURATION OF ACTIVITIES
  IloInt dmax = 0;
  for (i=0; i < numberOfActivities; i++)
    if (durations[i] > dmax)
      dmax = durations[i];

  IloActivity lact;
  IloNumExprArray times(env, numberOfActivities+1);
  for (i=0; i < numberOfActivities; i++) {
    // USE ACTIVITIES WITH MAXIMAL DURATION
    IloActivity act(env, dmax);
    times[i] = act.getStartExpr();
    if (i > 0)
```

```
      model.add(act.startsAfterEnd(lact));
    lact = act;
  }
  model.add(lact.endsBefore(makespan));
  times[numberOfActivities] = makespan;

  // ADD MINIMAL DELAY CONSTRAINTS
  for (k=0; k < numberOfConstraints; k++) {
    i = constraints[3*k];
    j = constraints[(3*k) + 1];
    t = constraints[(3*k) + 2];
    model.add(times[j]-times[i] >= t);
  }

  // JUST PROPAGATE THE PERT
  IloSolver solver(model);
  IlcSearch s = solver.newSearch(IloGoalTrue(env));
  s.next();
  IloNum makespanMax = solver.getMin(makespan);
  solver.end();

  return makespanMax;
}

//---------------------------- MODEL ----------------------------------
IloModel
DefineModel(const IloEnv& env,
            IloInt numberOfActivities,
            IloInt numberOfConstraints,
            IloInt* durations,
            IloInt* constraints,
            IloNumVar& makespan)
{
  IloModel model(env);
  IloUnaryResource ures(env);
  // THIS ENFORCEMENT LEVEL WILL POST THE BALANCE CONSTRAINT
  ures.setCapacityEnforcement(IloExtended);
  // THIS ENFORCEMENT LEVEL WILL POST THE PRECEDENCE GRAPH CONSTRAINT
  ures.setPrecedenceEnforcement(IloExtended);
  IloReservoir reserv(env, 2, 0);
  // THIS ENFORCEMENT LEVEL WILL POST THE BALANCE CONSTRAINT
  reserv.setCapacityEnforcement(IloExtended);
  // COMPUTES UPPER BOUND ON MAKESPAN
  IloNum makespanMax = getMakespanMax(env,
                                      numberOfActivities,
                                      numberOfConstraints,
                                      durations,
                                      constraints);
  makespan = IloNumVar(env, 0, makespanMax, ILOINT);

  IloNumExprArray times(env, numberOfActivities + 1);
  IloActivityArray acts(env, numberOfActivities);

  // COMPUTE DURATION OF SHORTEST ACTIVITY
  IloInt dmin = IloIntMax;
  IloInt i, j, k, t;
  for (i=0; i < numberOfActivities; i++)
    if (durations[i] < dmin)
```

```
      dmin = durations[i];

  IloActivity lprod;
  for (i=0; i < numberOfActivities; i++) {
    acts[i]= IloActivity(env, durations[i]);
    model.add(acts[i].requires(ures));
    model.add(acts[i].requires(reserv, 1, IloAfterEnd));
    model.add(acts[i].requires(reserv, 1, IloAfterStart));
    model.add(acts[i].endsBefore(makespan));
    IloActivity prod(env, 1);
    model.add(prod.provides(reserv, 2, IloAfterStart));
    times[i] = prod.getStartExpr();
    if (i > 0)
      // BUILD CHAIN OF PRODUCERS
      model.add(prod.startsAfterStart(lprod, dmin));
    lprod = prod;
  }
  times[numberOfActivities] = makespan;
  model.add(times[numberOfActivities-1] + dmin <= makespan);

  // BREAK EVIDENT SYMMETRIES BETWEEN ACTIVITIES WITH SAME DURATION
  for (i=0; i < numberOfActivities - 1; i++)
    for (j=i+1; j < numberOfActivities; j++)
      if (durations[i] == durations[j]) {
        model.add(acts[j].startsAfterEnd(acts[i]));
        break;
      }

  // ADD DELAY CONSTRAINTS
  for (k=0; k < numberOfConstraints; k++) {
    i = constraints[3*k];
    j = constraints[(3*k) + 1];
    t = constraints[(3*k) + 2];
    model.add(times[j]-times[i] >= t);
  }

  return model;
}

//------------------------- DISPLAY SOLUTION -----------------------------
void
PrintSolution(const IloSolver& solver)
{
  IlcScheduler sched(solver);
  IlcUnaryResource res = *(IlcUnaryResourceIterator(sched));
  for(IlcResourceConstraintIterator ite(res); ite.ok(); ++ite)
    solver.out() << (*ite).getActivity() << endl;
}

//---------------------- INITIALIZE PARAMETERS ----------------------------
void
InitParameters(int argc,
               char** argv,
               IloInt& numberOfConstraints,
               IloInt*& durations,
               IloInt*& constraints)
{
  IloInt p = 1;
```

```
  if (argc > 1)
    p = atol(argv[1]);

  switch (p) {
  case (1):
    numberOfConstraints = NumberOfConstraints01;
    durations   = Durations01;
    constraints = Constraints01;
    break;
  case (2):
    numberOfConstraints = NumberOfConstraints02;
    durations   = Durations02;
    constraints = Constraints02;
    break;
  case (3):
    numberOfConstraints = NumberOfConstraints03;
    durations   = Durations03;
    constraints = Constraints03;
    break;
  }
}

//------------------------ SOLVE PROBLEM ---------------------------------
int main(int argc, char* argv[])
{
  try {
    IloEnv env;

    IloInt numberOfActivities = NumberOfActivities;
    IloInt numberOfConstraints;
    IloInt* durations;
    IloInt* constraints;

    InitParameters(argc, argv,
                   numberOfConstraints,
                   durations, constraints);

    IloNumVar makespan;
    IloModel model = DefineModel(env,
                                 numberOfActivities,
                                 numberOfConstraints,
                                 durations,
                                 constraints,
                                 makespan);

    model.add(IloMinimize(env, makespan));
    IloSolver solver(model);
    IlcSearch search = solver.newSearch(IloRankForward(env, makespan));

    IloInt sumd = 0;
    for (IloInt i=0; i < numberOfActivities; i++)
      sumd += durations[i];

    while (search.next()) {
      solver.out() << "SOLUTION WITH SLACK = "
                   << solver.getIntVar(makespan).getMin() - sumd << endl;
      PrintSolution(solver);
    }
```

```
    solver.printInformation();
    solver.end();
    env.end();

  } catch (IloException& exc) {
    cout << exc << endl;
  }
  return 0;
}
```

An optimal solution for the first problem is described in Figure 22.4. It corresponds to a slack duration of 22 (5 between t_0 and t_1, 2 between t_5 and t_6, 6 between t_7 and t_8, 8 between t_{10} and t_{11} and 1 between t_{13} and t_{14}) for an optimal makespan of 157.

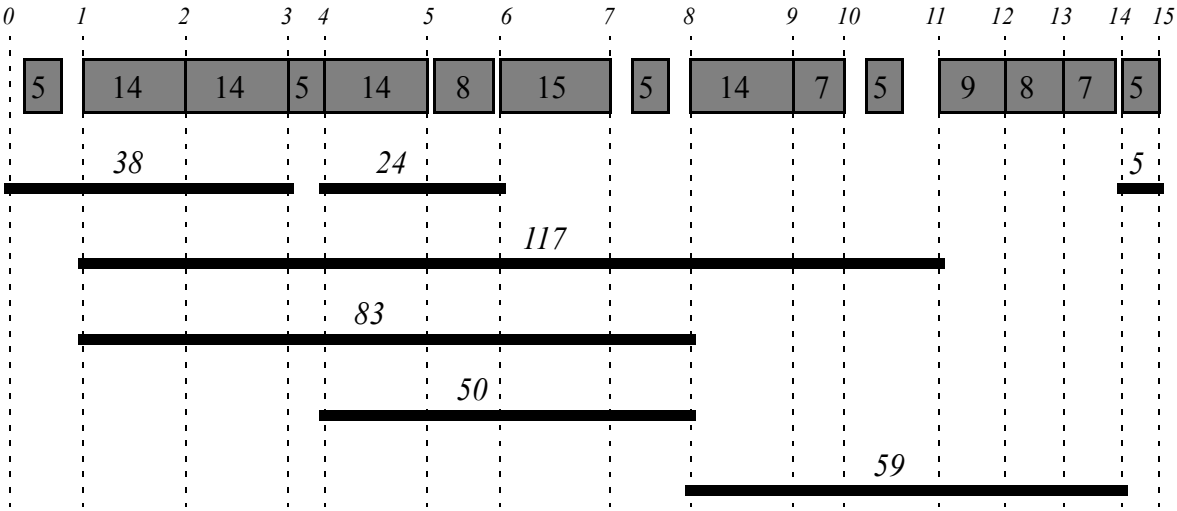


Figure 22.4 *Optimal Solution for Problem 1*

This optimal solution is quickly found after 2 iterations and 28 backtracks.

C H A P T E R

23

Scheduling with State Resources: the Trolley Problem

A *state resource* is one whose condition can vary over time. Each activity that uses a state resource may, throughout its execution, require the resource to be in a specific state or in any of a given set of states. Two activities may not overlap if they require incompatible states of the same state resource during their execution. A state resource does not constrain the number of activities that can be executed in parallel, provided that the overlapping activities require compatible states of the resource.

Typical examples of state resources are the color of a traffic light—the activity "going through the intersection" necessitates the state "green" of the traffic light—or the temperature of an oven—the activity "melting a piece of metal" requires the oven to be at a given temperature; several pieces of metal can simultaneously use the oven provided they need the same temperature.

The state resource concept includes a large range of resources. As we see in the two examples above, state transitions of a state resource can be imposed by the environment (the traffic light) or controlled by a schedule (the oven temperature). A state transition can be instantaneous or not—the oven needs time to warm or to cool. In this and other chapters, we illustrate how Scheduler allows us to represent complex state resources.

This chapter introduces a basic example of a state resource: a trolley in a production shop, used to carry items to the various machines. The position of the trolley represents its state.

Describing the Problem

The problem described in this chapter consists of *n* jobs to be performed in a shop equipped with *m* machines. A job corresponds to the processing of an item. Each item needs to be sequentially processed on *k* specified machines with known processing times.

In the problem, we want to model the physical distance between machines. The items to be processed are initially available in area *A* of the shop. Each time an item must be processed on a machine *M*, it must be brought to the area of this machine with a trolley. After the item has been processed, it must be stocked in area *S* of the shop.

Moving an item from an area *x* to an area *y* means (1) loading the item on the trolley at area *x*, (2) moving the trolley from area *x* to area *y* and, finally, (3) unloading the item at area *y*. We suppose that the activities of loading (and unloading) items can overlap, assuming that the trolley is in the same area as the items. We search for a solution that minimizes the makespan of the schedule.

In the first version of the problem, we do not consider the time taken by the trolley to go from one area to another. We suppose that there is only one trolley shared by all the jobs and that this trolley has no limit on the number of items it can carry. We want to solve an instance of the problem with six jobs and three machines (*M1*, *M2*, and *M3*) as shown in Figure 23.1. Each item requires processing on two machines. There are five areas in the shop (*A*, *M1*, *M2*, *M3*, and *S*) and each job consists of the following eight activities:

1. Load item on trolley at arrival area *A*.
2. Unload item from trolley at the area of the first machine required by the job.
3. Process item on this machine.
4. Load item on trolley at the area of this machine.
5. Unload item from trolley at the area of the second machine required by job.
6. Process item on this machine.
7. Load item on trolley at the area of this machine.
8. Unload item from trolley at the stock area *S*.

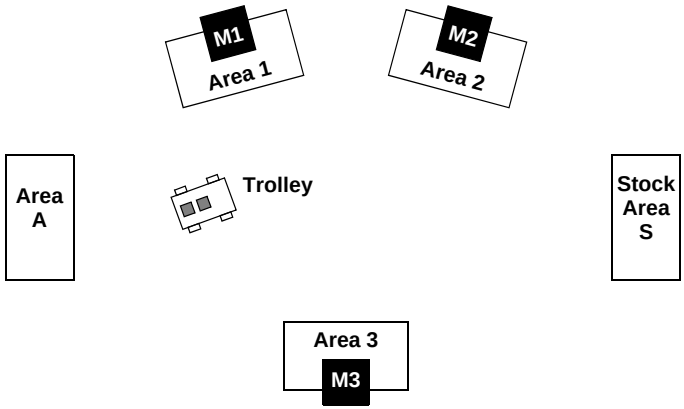


Figure 23.1 *Machines and Areas for the Trolley Problem*

The following C++ arrays provide the data defining the problem. The i^{th} column of arrays `JobNames`, `Machines1`, `Durations1`, `Machines2` and `Durations2` respectively define: the name of the i^{th} job, the number j of its first machine M_j, the processing time on this machine, the number k of its second machine M_k, and the processing time on this machine.

Two additional parameters are defined: the duration of loading (and unloading) activities and the number of jobs in the problem.

```
const IloNum loadDuration = 20;
const IloInt numberOfJobs = 6;

const char*  JobNames [] = { “J1”,   “J2”,   “J3”,   “J4”,   “J5”,   “J6”};
IloInt Machines1      [] = { 1,      2,      2,      1,      3,      2   };
IloNum Durations1     [] = { 80,     120,    80,     160,    180,    140 };
IloInt Machines2      [] = { 2,      3,      1,      3,      2,      3   };
IloNum Durations2     [] = { 60,     80,     60,     100,    80,     60  };
```

These parameters are passed to the function `DefineModel`. This function returns an instance of the class `IloModel` that corresponds to the problem and sets the constrained

variable makespan to be minimized. This function also stores an array of jobs (`Job*`) that is used later to print the schedule.

```
IloAnyArray jobs;
IloStateResource trolley;
IloArray<IloUnaryResource> machines;
IloNumVar makespan;
IloModel model = DefineModel(env,
    JobNames,
    Machines1, Durations1,
    Machines2, Durations2,
    jobs,
    trolley,
    machines,
    makespan);
```

Defining the Problem, Designing a Model

In this section, we'll take a look at the resources used and the jobs and activities to perform.

Resources

Two categories of resources are used in our problem: the trolley and the machines.

The trolley is a resource shared by all the jobs in the problem. Each activity consisting of loading/unloading an item on/from the trolley requires the trolley to be in the same area as the item. Notice that the trolley cannot be represented as a *unary resource* (class `IloUnaryResource`) because there is no limit on the number of load/unload activities that can be performed in parallel on the same trolley provided that the trolley is in a suitable area. We want to associate a state variable with the trolley (its position), and express the constraint that two loading or unloading activities that require the trolley to be at different positions cannot overlap.

An instance of the class `IloStateResource` is used to express this constraint. A state resource represents a resource, by default of infinite capacity, whose state can vary over time. Each activity that uses a state resource may require a state resource to be in a specific state or in any of a given set of states. Two activities cannot overlap if they require incompatible states during their execution.

To use state resources, we must first define the possible states of the resource. In this example, the states represent the possible positions of the trolley, and are defined as follows.

```
const IloInt MACH1 = 1; // Machine 1 area
const IloInt MACH2 = 2; // Machine 2 area
const IloInt MACH3 = 3; // Machine 3 area
const IloInt AREAA = 4; // Arrival area
const IloInt AREAS = 5; // Stock area
```

Next, the state resource representing the position of the trolley is created as follows in the `DefineModel` function.

```
  /* CREATE THE TROLLEY POSITION RESOURCE */
  IloAnyArray arrayOfPossibleAreas(env,
                                   5,
                                   (IloAny)AREAA,
                                   (IloAny)MACH1,
                                   (IloAny)MACH2,
                                   (IloAny)MACH3,
                                   (IloAny)AREAS);
  IloAnySet setOfPossibleAreas(env,arrayOfPossibleAreas);
  trolley = IloStateResource(env,setOfPossibleAreas, “trolley”);
```

As in the job-shop scheduling problem, each machine is represented as a unary resource (class `IloUnaryResource`).

```
  /* CREATE THE MACHINES RESOURCES */
  char buffer[256];
  machines = IloArray<IloUnaryResource>(env, 3);
  for (i = 0; i < 3; i++) {
    sprintf(buffer, “machine%ld”, i + 1);
    machines[i] = IloUnaryResource(env, buffer);
  }
```

Jobs and Activities

The code that follows defines a new class, `Job`, used to represent the activities associated with the processing of an item.

```
class Job {
private:
  const char* _name;
  IloActivity _loadA;
  IloActivity _unload1;
  IloActivity _process1;
  IloActivity _load1;
  IloActivity _unload2;
  IloActivity _process2;
  IloActivity _load2;
  IloActivity _unloadS;
  IloInt      _area1;
  IloInt      _area2;
  IloUnaryResource _machine1;
  IloUnaryResource _machine2;
public:
  Job(IloEnv env,
      const char*,
      IloNum,
      IloUnaryResource, IloNum, IloInt,
      IloUnaryResource, IloNum, IloInt);
  ~Job();
  void addToModel(IloModel model,
                  IloStateResource trolley,
```

```
                IloNumVar makespan);
  void printSolution(IlcScheduler scheduler, ILOSTD(ostream)& out);
  const char* getName() const { return  _name;}

  IloActivity getLoadA() const { return _loadA;}
  IloActivity getUnload1() const { return _unload1;}
  IloActivity getProcess1() const { return _process1;}
  IloActivity getLoad1() const { return _load1;}
  IloActivity getUnload2() const { return _unload2;}
  IloActivity getProcess2() const { return _process2;}
  IloActivity getLoad2() const { return _load2;}
  IloActivity getUnloadS() const { return _unloadS;}

};
```

As stated earlier, a job consists of a sequence of eight activities. The fields `_area1` and `_area2` respectively store the areas of the first and the second machine required to process the item.

The constructor of an instance of `Job` is implemented below. The code shows how to implement the constraint so that load/unload activities require the trolley to be in a given position. The constructor also constrains the makespan variable to be greater than or equal to the job completion time.

```
Job::Job(IloEnv env,
         const char*  name,
         IloNum loadDuration,
         IloUnaryResource machine1, IloNum duration1, IloInt area1,
         IloUnaryResource machine2, IloNum duration2, IloInt area2)
  : _name(name),
    _loadA(   env,loadDuration),
    _unload1( env,loadDuration),
    _process1(env,duration1),
    _load1(   env,loadDuration),
    _unload2( env,loadDuration),
    _process2(env,duration2),
    _load2(   env,loadDuration),
    _unloadS( env,loadDuration),
    _area1(area1),
    _area2(area2),
    _machine1(machine1),
    _machine2(machine2)
{
  char buffer[256];

  sprintf(buffer, “arrival_load_%s”, name);
  _loadA.setName(buffer);

  sprintf(buffer, “area%ld_unload_%s”, area1, name);
  _unload1.setName(buffer);

  sprintf(buffer, “machine%ld_%s”, area1, name);
  _process1.setName(buffer);

  sprintf(buffer, “area%ld_load_%s”, area1, name);
  _load1.setName(buffer);
```

```
  sprintf(buffer, “area%ld_unload_%s”, area2, name);
  _unload2.setName(buffer);

  sprintf(buffer, “machine%ld_%s”, area2, name);
  _process2.setName(buffer);

  sprintf(buffer, “area%ld_load_%s”, area2, name);
  _load2.setName(buffer);

  sprintf(buffer, “stock_load_%s”, name);
  _unloadS.setName(buffer);

}

Job::~Job()
{}

void Job::addToModel(IloModel model,
                     IloStateResource trolley,
                     IloNumVar makespan)
{
  /* ADD MACHINE REQUIREMENT CONSTRAINTS */
  model.add(_process1.requires(_machine1));
  model.add(_process2.requires(_machine2));

  /* ADD TEMPORAL CONSTRAINTS BETWEEN ACTIVITIES */
  model.add(_unload1.startsAfterEnd(_loadA));
  model.add(_process1.startsAfterEnd(_unload1));
  model.add(_load1.startsAfterEnd(_process1));
  model.add(_unload2.startsAfterEnd(_load1));
  model.add(_process2.startsAfterEnd(_unload2));
  model.add(_load2.startsAfterEnd(_process2));
  model.add(_unloadS.startsAfterEnd(_load2));

  /* ADD TROLLEY POSITION REQUIREMENTS  */
  model.add(_loadA.requires(trolley, (IloAny)AREAA));
  model.add(_unload1.requires(trolley, (IloAny)_area1));
  model.add(_load1.requires(trolley, (IloAny)_area1));
  model.add(_unload2.requires(trolley, (IloAny)_area2));
  model.add(_load2.requires(trolley, (IloAny)_area2));
  model.add(_unloadS.requires(trolley, (IloAny)AREAS));

  /* ADD MAKESPAN CONSTRAINT */
  model.add(_unloadS.endsBefore(makespan));
}
```

The function that follows displays a job on the `out()` stream of the solver.

```
void Job::printSolution(IlcScheduler scheduler, ILOSTD(ostream)& out)
{
  /* PRINT JOB */
  out
    << “JOB “ << _name << endl
    << “\t Load at area A: “
    << scheduler.getActivity( _loadA ) << endl
    << “\t Unload at area “ << (long)_area1 << “: “
    << scheduler.getActivity( _unload1 )   << endl
```

```
    << "\t Process on machine " << (long)_area1 << ": "
    << scheduler.getActivity( _process1 )  << endl
    << "\t Load at area "   << (long)_area1 << ": "
    << scheduler.getActivity( _load1 )     << endl
    << "\t Unload at area " << (long)_area2 << ": "
    << scheduler.getActivity( _unload2 )   << endl
    << "\t Process on machine " << (long)_area2 << ": "
    << scheduler.getActivity( _process2 )  << endl
    << "\t Load at area "   << (long)_area2 << " : "
    << scheduler.getActivity( _load2 )     << endl
    << "\t Unload at area S: "
    << scheduler.getActivity( _unloadS )   << endl;
}
```

That function is used by the function `PrintSolution` to print the schedule, along with a list of the chronological positions of the trolley.

```
void PrintSolution(IloSolver solver,
                   IloAnyArray jobs,
                   IloStateResource trolley,
                   IloArray<IloUnaryResource> machines,
                   IloNumVar makespan)
{
  IlcScheduler scheduler(solver);

  solver.out() << "Solution with makespan " << solver.getMin(makespan) << endl;
  IloInt numberOfJobs = jobs.getSize();
  for (IloInt i = 0; i < numberOfJobs; i++)
    ((Job*) jobs[i])->printSolution(scheduler, solver.out());

  // Output from algorithm
  solver.out() << "TROLLEY POSITIONS: " << endl;

  IlcAnyTimetable tt = scheduler.getStateResource(trolley).getTimetable();
  for (IlcAnyTimetableCursor cr(tt,0); cr.ok(); ++cr)
    if (cr.isBound())
      solver.out() << "\t [" << cr.getTimeMin()
                   << "," << cr.getTimeMax()
                   << "): Position " << (long)cr.getValue()
                   << endl;
  solver.printInformation();

#if defined(ILO_SDXLOUTPUT)
  IloSDXLTrolleyOutput output(solver.getEnv());
  ofstream outFile("trolley1.xml");
  output.writeTrolley(scheduler,
                      outFile,
                      jobs,
                      trolley,
                      (IloInt) 1000,
                      machines,
                      makespan);
  outFile.close();
#endif

}
```

In the definition of the scheduling problem, the jobs of the problem are created and stored in an array as follows.

```
/* CREATION OF JOB INSTANCES */
jobs = IloAnyArray(env, numberOfJobs);
for (i = 0; i < numberOfJobs; i++) {
  Job* job = new (env) Job(env,
                           jobNames[i],
                           loadDuration,
                           machines[machines1[i] - 1],
                           durations1[i], machines1[i],
                           machines[machines2[i] - 1],
                           durations2[i], machines2[i]);
  jobs[i] = job;
  job->addToModel(model, trolley, makespan);
}
```

Minimizing the Makespan

The Concert Technology function `IloMinimize` is used to set the objective of minimizing the `makespan` in the model.

```
/* WE LOOK FOR AN OPTIMAL SOLUTION */
model.add(IloMinimize(env, makespan));
```

Complete Program and Output

You can see the entire program `trolley1.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN
const IloInt MACH1 = 1; // Machine 1 area
const IloInt MACH2 = 2; // Machine 2 area
const IloInt MACH3 = 3; // Machine 3 area
const IloInt AREAA = 4; // Arrival area
const IloInt AREAS = 5; // Stock area
const IloNum loadDuration = 20;
const IloInt numberOfJobs = 6;

const char*  JobNames [] = { “J1”,   “J2”,   “J3”,   “J4”,   “J5”,   “J6”};
IloInt Machines1      [] = { 1,      2,      2,      1,      3,      2   };
IloNum Durations1     [] = { 80,     120,    80,     160,    180,    140 };
IloInt Machines2      [] = { 2,      3,      1,      3,      2,      3   };
IloNum Durations2     [] = { 60,     80,     60,     100,    80,     60  };
////////////////////////////////////////////////////////////////////////////
//
// DEFINITION OF THE JOB CLASS
```

```
//
////////////////////////////////////////////////////////////////////////////
class Job {
private:
  const char* _name;
  IloActivity _loadA;
  IloActivity _unload1;
  IloActivity _process1;
  IloActivity _load1;
  IloActivity _unload2;
  IloActivity _process2;
  IloActivity _load2;
  IloActivity _unloadS;
  IloInt      _area1;
  IloInt      _area2;
  IloUnaryResource _machine1;
  IloUnaryResource _machine2;
public:
  Job(IloEnv env,
      const char*,
      IloNum,
      IloUnaryResource, IloNum, IloInt,
      IloUnaryResource, IloNum, IloInt);
  ~Job();
  void addToModel(IloModel model,
                  IloStateResource trolley,
                  IloNumVar makespan);
  void printSolution(IlcScheduler scheduler, ILOSTD(ostream)& out);
  const char* getName() const { return  _name;}

  IloActivity getLoadA() const { return _loadA;}
  IloActivity getUnload1() const { return _unload1;}
  IloActivity getProcess1() const { return _process1;}
  IloActivity getLoad1() const { return _load1;}
  IloActivity getUnload2() const { return _unload2;}
  IloActivity getProcess2() const { return _process2;}
  IloActivity getLoad2() const { return _load2;}
  IloActivity getUnloadS() const { return _unloadS;}

};

Job::Job(IloEnv env,
         const char*  name,
         IloNum loadDuration,
         IloUnaryResource machine1, IloNum duration1, IloInt area1,
         IloUnaryResource machine2, IloNum duration2, IloInt area2)
  : _name(name),
    _loadA(   env,loadDuration),
    _unload1( env,loadDuration),
    _process1(env,duration1),
    _load1(   env,loadDuration),
    _unload2( env,loadDuration),
    _process2(env,duration2),
    _load2(   env,loadDuration),
    _unloadS( env,loadDuration),
    _area1(area1),
    _area2(area2),
    _machine1(machine1),
```

```
    _machine2(machine2)
{
  char buffer[256];

  sprintf(buffer, “arrival_load_%s”, name);
  _loadA.setName(buffer);

  sprintf(buffer, “area%ld_unload_%s”, area1, name);
  _unload1.setName(buffer);

  sprintf(buffer, “machine%ld_%s”, area1, name);
  _process1.setName(buffer);

  sprintf(buffer, “area%ld_load_%s”, area1, name);
  _load1.setName(buffer);

  sprintf(buffer, “area%ld_unload_%s”, area2, name);
  _unload2.setName(buffer);

  sprintf(buffer, “machine%ld_%s”, area2, name);
  _process2.setName(buffer);

  sprintf(buffer, “area%ld_load_%s”, area2, name);
  _load2.setName(buffer);

  sprintf(buffer, “stock_load_%s”, name);
  _unloadS.setName(buffer);

}

Job::~Job()
{}

void Job::addToModel(IloModel model,
                     IloStateResource trolley,
                     IloNumVar makespan)
{
  /* ADD MACHINE REQUIREMENT CONSTRAINTS */
  model.add(_process1.requires(_machine1));
  model.add(_process2.requires(_machine2));

  /* ADD TEMPORAL CONSTRAINTS BETWEEN ACTIVITIES */
  model.add(_unload1.startsAfterEnd(_loadA));
  model.add(_process1.startsAfterEnd(_unload1));
  model.add(_load1.startsAfterEnd(_process1));
  model.add(_unload2.startsAfterEnd(_load1));
  model.add(_process2.startsAfterEnd(_unload2));
  model.add(_load2.startsAfterEnd(_process2));
  model.add(_unloadS.startsAfterEnd(_load2));

  /* ADD TROLLEY POSITION REQUIREMENTS  */
  model.add(_loadA.requires(trolley, (IloAny)AREAA));
  model.add(_unload1.requires(trolley, (IloAny)_area1));
  model.add(_load1.requires(trolley, (IloAny)_area1));
  model.add(_unload2.requires(trolley, (IloAny)_area2));
  model.add(_load2.requires(trolley, (IloAny)_area2));
  model.add(_unloadS.requires(trolley, (IloAny)AREAS));
```

```
  /* ADD MAKESPAN CONSTRAINT */
  model.add(_unloadS.endsBefore(makespan));
}

void Job::printSolution(IlcScheduler scheduler, ILOSTD(ostream)& out)
{
  /* PRINT JOB */
  out
    << "JOB " << _name << endl
    << "\t Load at area A: "
    << scheduler.getActivity( _loadA ) << endl
    << "\t Unload at area " << (long)_area1 << ": "
    << scheduler.getActivity( _unload1 )   << endl
    << "\t Process on machine " << (long)_area1 << ": "
    << scheduler.getActivity( _process1 )  << endl
    << "\t Load at area "   << (long)_area1 << ": "
    << scheduler.getActivity( _load1 )     << endl
    << "\t Unload at area " << (long)_area2 << ": "
    << scheduler.getActivity( _unload2 )   << endl
    << "\t Process on machine " << (long)_area2 << ": "
    << scheduler.getActivity( _process2 )  << endl
    << "\t Load at area "   << (long)_area2 << " : "
    << scheduler.getActivity( _load2 )     << endl
    << "\t Unload at area S: "
    << scheduler.getActivity( _unloadS )   << endl;
}

////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////

#if defined(ILO_SDXLOUTPUT)
#include "sdxltrolley.h"
#endif

IloModel DefineModel(IloEnv env,
                     const char** jobNames,
                     IloInt*      machines1,
                     IloNum*      durations1,
                     IloInt*      machines2,
                     IloNum*      durations2,
                     IloAnyArray&  jobs,
                     IloStateResource& trolley,
                     IloArray<IloUnaryResource>& machines,
                     IloNumVar&   makespan)
{
  IloInt i;
  IloModel model(env);
  /* CREATE THE MAKESPAN VARIABLE. */
  makespan = IloNumVar(env,0,10000,ILOINT);
  /* CREATE THE TROLLEY POSITION RESOURCE */
  IloAnyArray arrayOfPossibleAreas(env,
                                   5,
                                   (IloAny)AREAA,
                                   (IloAny)MACH1,
                                   (IloAny)MACH2,
```

```
                                    (IloAny)MACH3,
                                    (IloAny)AREAS);
  IloAnySet setOfPossibleAreas(env,arrayOfPossibleAreas);
  trolley = IloStateResource(env,setOfPossibleAreas, "trolley");
  /* CREATE THE MACHINES RESOURCES */
  char buffer[256];
  machines = IloArray<IloUnaryResource>(env, 3);
  for (i = 0; i < 3; i++) {
    sprintf(buffer, "machine%ld", i + 1);
    machines[i] = IloUnaryResource(env, buffer);
  }
  /* CREATION OF JOB INSTANCES */
  jobs = IloAnyArray(env, numberOfJobs);
  for (i = 0; i < numberOfJobs; i++) {
    Job* job = new (env) Job(env,
                             jobNames[i],
                             loadDuration,
                             machines[machines1[i] - 1],
                             durations1[i], machines1[i],
                             machines[machines2[i] - 1],
                             durations2[i], machines2[i]);
    jobs[i] = job;
    job->addToModel(model, trolley, makespan);
  }
  /* WE LOOK FOR AN OPTIMAL SOLUTION */
  model.add(IloMinimize(env, makespan));
  /* RETURN THE CREATED MODEL */
  return model;
}


////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////////////////

void PrintSolution(IloSolver solver,
                   IloAnyArray jobs,
                   IloStateResource trolley,
                   IloArray<IloUnaryResource> machines,
                   IloNumVar makespan)
{
  IlcScheduler scheduler(solver);

  solver.out() << "Solution with makespan " << solver.getMin(makespan) << endl;
  IloInt numberOfJobs = jobs.getSize();
  for (IloInt i = 0; i < numberOfJobs; i++)
    ((Job*) jobs[i])->printSolution(scheduler, solver.out());

  // Output from algorithm
  solver.out() << "TROLLEY POSITIONS: " << endl;

  IlcAnyTimetable tt = scheduler.getStateResource(trolley).getTimetable();
  for (IlcAnyTimetableCursor cr(tt,0); cr.ok(); ++cr)
    if (cr.isBound())
      solver.out() << "\t [" << cr.getTimeMin()
                   << "," << cr.getTimeMax()
```

```
                  << "): Position " << (long)cr.getValue()
                  << endl;
  solver.printInformation();

#if defined(ILO_SDXLOUTPUT)
  IloSDXLTrolleyOutput output(solver.getEnv());
  ofstream outFile("trolley1.xml");
  output.writeTrolley(scheduler,
                      outFile,
                      jobs,
                      trolley,
                      (IloInt) 1000,
                      machines,
                      makespan);
  outFile.close();
#endif

}
////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////

int main() {
  try {
    IloEnv env;
    IloAnyArray jobs;
    IloStateResource trolley;
    IloArray<IloUnaryResource> machines;
    IloNumVar makespan;
    IloModel model = DefineModel(env,
        JobNames,
        Machines1, Durations1,
        Machines2, Durations2,
        jobs,
        trolley,
        machines,
        makespan);

     IloSolver solver(model);
     IlcScheduler scheduler(solver);
     IloGoal goal = IloSetTimesForward(env, makespan);

     if (solver.solve(goal)) {
       PrintSolution(solver,jobs,trolley,machines,makespan);
     }
     else
       solver.out() << "No Solution" << endl;

     env.end();

  } catch (IloException& exc) {
    cout << exc << endl;
  }
  return 0;
}
```

```
////////////////////////////////////////////////////////////////////////////
//
// RESULTS
//
////////////////////////////////////////////////////////////////////////////

/*
  Solution with makespan 560
  JOB J1
         Load at area A: [0 -- 20 --> 20]
         Unload at area 1: [60 -- 20 --> 80]
         Process on machine 1: [240 -- 80 --> 320]
         Load at area 1: [320 -- 20 --> 340]
         Unload at area 2: [360 -- 20 --> 380]
         Process on machine 2: [460 -- 60 --> 520]
         Load at area 2 : [520 -- 20 --> 540]
         Unload at area S: [540 -- 20 --> 560]
  JOB J2
         Load at area A: [0 -- 20 --> 20]
         Unload at area 2: [20 -- 20 --> 40]
         Process on machine 2: [40 -- 120 --> 160]
         Load at area 2: [160 -- 20 --> 180]
         Unload at area 3: [180 -- 20 --> 200]
         Process on machine 3: [240 -- 80 --> 320]
         Load at area 3 : [340 -- 20 --> 360]
         Unload at area S: [440 -- 20 --> 460]
  JOB J3
         Load at area A: [0 -- 20 --> 20]
         Unload at area 2: [20 -- 20 --> 40]
         Process on machine 2: [300 -- 80 --> 380]
         Load at area 2: [380 -- 20 --> 400]
         Unload at area 1: [400 -- 20 --> 420]
         Process on machine 1: [420 -- 60 --> 480]
         Load at area 1 : [500 -- 20 --> 520]
         Unload at area S: [540 -- 20 --> 560]
  JOB J4
         Load at area A: [0 -- 20 --> 20]
         Unload at area 1: [60 -- 20 --> 80]
         Process on machine 1: [80 -- 160 --> 240]
         Load at area 1: [240 -- 20 --> 260]
         Unload at area 3: [260 -- 20 --> 280]
         Process on machine 3: [320 -- 100 --> 420]
         Load at area 3 : [420 -- 20 --> 440]
         Unload at area S: [440 -- 20 --> 460]
  JOB J5
         Load at area A: [0 -- 20 --> 20]
         Unload at area 3: [40 -- 20 --> 60]
         Process on machine 3: [60 -- 180 --> 240]
         Load at area 3: [260 -- 20 --> 280]
         Unload at area 2: [280 -- 20 --> 300]
         Process on machine 2: [380 -- 80 --> 460]
         Load at area 2 : [460 -- 20 --> 480]
         Unload at area S: [540 -- 20 --> 560]
  JOB J6
         Load at area A: [0 -- 20 --> 20]
         Unload at area 2: [20 -- 20 --> 40]
         Process on machine 2: [160 -- 140 --> 300]
```

```
        Load at area 2: [300 -- 20 --> 320]
        Unload at area 3: [340 -- 20 --> 360]
        Process on machine 3: [420 -- 60 --> 480]
        Load at area 3 : [480 -- 20 --> 500]
        Unload at area S: [540 -- 20 --> 560]
  TROLLEY POSITIONS:
        [0,20): Position 4
        [20,40): Position 2
        [40,60): Position 3
        [60,80): Position 1
        [160,180): Position 2
        [180,200): Position 3
        [240,260): Position 1
        [260,280): Position 3
        [280,320): Position 2
        [320,340): Position 1
        [340,360): Position 3
        [360,400): Position 2
        [400,420): Position 1
        [420,440): Position 3
        [440,460): Position 5
        [460,480): Position 2
        [480,500): Position 3
        [500,520): Position 1
        [520,540): Position 2
        [540,560): Position 5
*/
```

This optimal solution is represented in Figure 23.2. The bottom line of the figure shows the positions of the trolley over time.

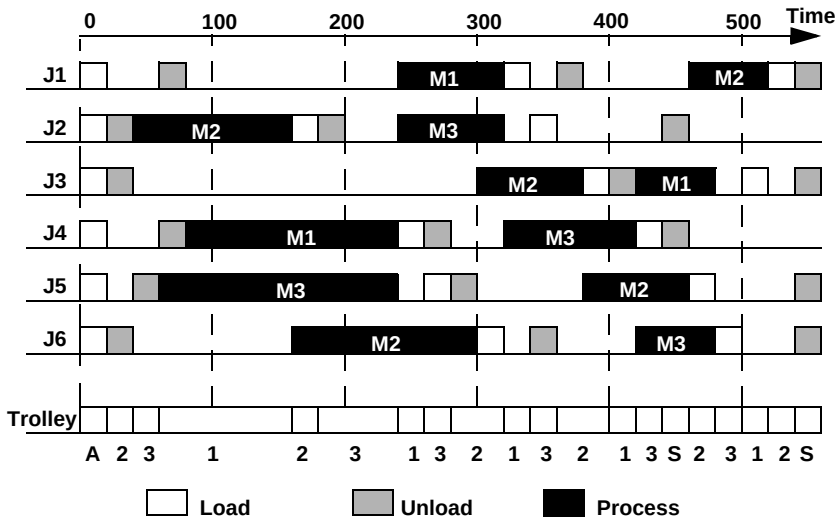


Figure 23.2 *An Optimal Solution for the First Version of the Trolley Problem*

C H A P T E R

24

Scheduling with Durable Resources

Scheduler offers an easy way to build applications where the same resources are used successively in different scheduling problems and information about resource availability is kept from one problem to another. Such resources are called *durable resources* and they are created on durable schedules.

This facility is particularly useful in applications which naturally decompose into parts between which no backtracking may occur, and where the sole information of interest from one part to another is the resource availability.

Scheduling with durable resources can also be a good approach for coping with large problems containing many activities and constraints. By choosing to decompose the problem into independent parts which are using the same durable resources, the global complexity may be drastically cut down. Of course, with such a decomposition you solve only a relaxed form of the initial problem, and thus you cannot get an optimal solution. However, decomposing the problem and using durable resources may be the only practical approach to tackling such large scheduling problems.

In this chapter we present an example illustrating the typical steps encountered when writing an application using durable resources. These steps are:

1. Create the durable resources.
2. Use the durable resources in different schedules to define and solve a particular scheduling problem.

3. Lock the particular durable resources needed by the schedule prior to using them.
4. After solving the specific scheduling problem, explicitly unlock the locked durable resources, thus making them available to other schedules.
5. Once a resource is unlocked by a schedule, any further computation done on that schedule will no longer interfere with the unlocked resource, except for backtracking. That is, backtracking a decision that modified the timetable of a resource will undo the modification, even if the resource has been unlocked.

Describing the Problem

Consider a small workshop employing eight workers and specialized in manufacturing two types of products, starting from raw materials provided by customers. Several customers can order one of the two types of product in advance for the next month. Each customer orders his product through a phone call during which he specifies the date at which he can provide the corresponding raw materials. The manager of the workshop replies by either:

- proposing the date at which the product will be delivered, or
- declining to deliver to the customer by the next month.

In the case where a delivery date is proposed, the customer can refuse the proposal. If the proposed date is accepted, it becomes a due date at which the manager commits to deliver the ordered product.

The workshop's workers are grouped into two teams, each dedicated to manufacturing one type of product. Inside each team, a worker specializes in one activity among the activities composing the manufacturing process of the corresponding product. Each activity takes one day to be performed and precedence constraints hold between the activities needed to build up a product.

The problem we are asked to solve is to determine the day by day timetable of each worker for the coming month, such that all promised products will be delivered on time and knowing that every five days in the month all workers have a two day break. The computed timetable will specify required working days for each worker.

Designing a Model

Each worker is modeled as a unary resource. Since the timetable information is of interest, timetable constraints should be explicitly added on these resources. The requirement that every worker has a two day break every five days is stated by setting a breaklist on each resource. In addition, each customer request is characterized by the team which

manufactures the requested product and by the date on which the team can start working on that product (we call that date the release date of the request).

This data can be easily translated into new activities and constraints as follows: For each team member a one day activity requiring the corresponding resource is created and all these activities are constrained to start after the release date. In addition, precedence constraints are added between these activities, according to the description of the manufacturing process of the product (in this example we suppose that both types of products impose the same precedences between activities). The solution of the request, the delivery date communicated to the customer, is represented by the end of the last activity in the manufacturing process.

Solving The Problem

Each time a new customer request arrives, new activities are created and new resource and precedence constraints are posted, according to the model description of the request.

A First Method

The following pseudocode illustrates how the whole problem can be solved iteratively by translating each arrived request into new activities and constraints. In this pseudocode, `AcceptOrNot` is a goal which fails if the proposed solution is refused by the customer.

- Create an environment '`env`'
- Create a model '`model`' in this environment
- Create the unary resources corresponding to the workers
- Add breaklists and timetable constraints on the unary resources
- While(new request has arrived) {
- let '`TreatRequest`' be a goal which adds new activities, posts new constraints according to the request, and then finds the solution of the request by using the goal '`IloSetTimesForward`'
- solve the goal '`IloAnd(TreatRequest(env, request, model), AcceptOrNot(env, model))`'
- the found solution (if any) becomes a commitment by calling '`env.commit`'
- }

After treating all requests, by simply reading the timetable associated with each unary resource we obtain the required information about the presence on the site of the corresponding worker during the coming month.

However, this proposed method has a major drawback: as new requests are treated, the created activities and constraints are accumulated in the system, thus increasing memory consumption and slowing down general responsiveness.

Using Durable Resources

In Scheduler there is another way of handling this kind of problem, thus avoiding the major drawback of accumulated activities and constraints. It is based on the observation that as we cannot backtrack over an already solved request, the only useful information needed from one iteration to another is the resource availability for each worker. This information is kept by the timetable associated with each resource.

Durable resources provide you with a natural way to keep timetable information from one iteration to another while deleting the activities and constraints which were used to compute the information.

In this approach, we consider the workers as durable unary resources and identify each customer request as a separate scheduling problem involving those durable resources.

Let us consider that the group of eight workers is represented by an array of `IloUnaryResource`, where the first four positions represent the first team and the last four positions represent the second team of workers.

```
const IloInt nbOfWorkers = 8;
IloUnaryResource workers[nbOfWorkers];
IloUnaryResource* firstTeam = workers;
IloUnaryResource* secondTeam = workers + 4;
```

Next the durable model and resources are created with following function.

```
IlcScheduler CreateDurableResources(IloSolver solver0,
                                    IloInt nbOfWorkers,
                                    IloUnaryResource* workers) {
  /* CREATE A DURABLE MODEL. */
  IloEnv env0 = solver0.getEnv();
  IloModel model0(env0);
  IloSchedulerEnv schedEnv0(env0);

  IloIntervalList blist(env0, 0, 100);
  blist.addPeriodicInterval(5, 2, 7, 100);
  schedEnv0.setBreakListParam(blist);

  IlcInt i=0;
  for (i = 0; i < nbOfWorkers; i++){
    workers[i] = IloUnaryResource(env0);
    model0.add(workers[i]);
  }

  IlcScheduler durSched(solver0);
  durSched.setDurable();
  solver0.extract(model0);
```

```
  /* CLOSE THE DURABLE SCHEDULER. */
  durSched.close();

  return durSched;
}
```

The following goal defines the scheduling problem associated with a given request and solves it. If the computed solution is not accepted, then the whole problem is backtracked, putting the system in the state it was before treating the request.

The schedule defined within this goal is associated with the given request. The durable resources representing the members of the team must be locked before using them, as shown in the following code.

```
ILCGOAL3(TreatRequestIlc,
         IlcInt, reqIndex,
         IlcUnaryResource*, team,
         IlcInt, releaseDate){

  /* CREATING A SCHEDULE. */
  IloSolver solver = getSolver();
  IlcSchedule schedule(solver, 0, Horizon);
  /* LOCKING THE NEEDED RESOURCES. */
  schedule.lock(4, team);


  /* ... */
  return 0;
}
```

Then, in the same goal, the scheduling problem is defined.

```
  /* DEFINING THE PROBLEM. */
  IlcActivity* act = new (solver.getHeap()) IlcActivity[4];
  for (IlcInt i = 0; i < 4; i++){
    act[i] = IlcActivity(schedule, 1);
    solver.add(act[i].requires(team[i]));
  }
  solver.add(act[0].startsAfter(releaseDate));
  solver.add(act[1].startsAfterEnd(act[0]));
  solver.add(act[2].startsAfterEnd(act[0]));
  solver.add(act[3].startsAfterEnd(act[1]));
  solver.add(act[3].startsAfterEnd(act[2]));
```

The goal returned by `IlcSetTimes` is used to solve the problem.

```
  /* SOLVING THE PROBLEM. */
  solver.startNewSearch(IlcSetTimes(schedule));
  IloBool solved = solver.next();
```

As the durable resources are no longer needed, they can now be unlocked, thus allowing other requests to lock them.

```
  /* UNLOCKING THE USED COMPUTATIONAL RESOURCES. */
  schedule.unlock(4, team);
```

The computed solution communicated to the customer is represented by the end time of the activity `act[3]`, which is the last in the manufacturing process. We call this date the due date of the request. For the sake of the example, the fact that the due date may not be accepted by the customer is simulated by comparing it with an acceptance date. This acceptance date is defined as being five days later than the release date. If the due date is greater than the acceptance date, then the proposed solution is refused and the whole problem associated to the request is backtracked by calling `solver.restartSearch()`.

```
  if (solved) {
  /* ACCEPTING OR NOT. */
    IlcInt acceptDate = releaseDate + 5;
    IlcInt dueDate = act[3].getEndMin();
    if (dueDate > acceptDate){
      // Not accepting :
      solver.out() << "request " << reqIndex << ": refused " << endl;
      solver.restartSearch();
    }
    else {
      // Accepting :
      solver.out() << "request " << reqIndex
              << ": promised at " << act[3].getEndMin()
              << endl;
    }
  }
  else
    solver.out() << "request " << reqIndex
            << ": not solved " << endl;
  solver.endSearch();
  return 0;
}
```

The main loop of the application reads the requests one by one from some input stream of requests that we here simulate by an array of pairs (type of product, release date), each pair defining a request.

```
const IloInt HowManyReqs = 20;

IloInt StreamOfReqs [20] [2] = { /* pairs type releaseDate */
  {1, 5},    // request 0
  {2, 15},   // request 1
  {2, 7},    // request 2
  {1, 24},   // request 3
  {2, 2},    // request 4
  {1, 10},   // request 5
  {1, 25},   // request 6
  {1, 17},   // request 7
  {2, 8},    // request 8
  {1, 15},   // request 9
```

```
  {2, 21},   // request 10
  {2, 5},    // request 11
  {1, 15},   // request 12
  {2, 17},   // request 13
  {2, 24},   // request 14
  {2, 2},    // request 15
  {1, 10},   // request 16
  {2, 28},   // request 17
  {1, 17},   // request 18
  {2, 10}    // request 19
};
```

At each iteration a new environment is created and used to treat the current request. The environment is deleted after the request treatment, as the scheduling problem it contains is no longer of interest.

```
  /* TREATING THE REQUESTS. */
  for (IloInt j = 0; j < HowManyReqs; j++){
    // Reading a request :
    IloInt requestType = StreamOfReqs[j][0];
    IloInt releaseDate = StreamOfReqs[j][1];

    IloEnv env;
    IloSolver solver(env);

    // Solving the specific problem associated with the request:
    IloUnaryResource* team = (requestType == 1 ?
                              firstTeam :
                              secondTeam);
   solver.solve(TreatRequest(env, durableSched, j, team, releaseDate));
    env.end();
  }
```

After all customer requests are processed, the timetable of the associated durable resource will contain a day by day timetable for the coming month for each worker. Timetables can be inspected using cursors.

```
  /* PRINTING THE SCHEDULE OF WORKERS. */
  IlcScheduler scheduler0(solver0);
  IloSchedulerSolution solution(env0);

  for (IloInt i = 0; i < nbOfWorkers; i++){
    solver0.out() << endl << "worker" << i << ": " << endl;
    solution.store(workers[i], scheduler0);
    for (IloNumToNumStepFunctionCursor curs(solution.getLevelMin(workers[i]));
         curs.ok();
         ++curs)
      solver0.out() << curs.getSegmentMin() << " .. "
                    << IloMin(Horizon, (curs.getSegmentMax() - 1)) << " : "
                    << (curs.getValue() == 1 ? "On" : "Off")
                    << endl;
  }
```

Complete Program and Output

You can see the entire program `durable.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

const IloInt Horizon = 30;

////////////////////////////////////////////
//
// CREATING THE DURABLE RESOURCES
//
////////////////////////////////////////////

IlcScheduler CreateDurableResources(IloSolver solver0,
                                    IloInt nbOfWorkers,
                                    IloUnaryResource* workers) {
  /* CREATE A DURABLE MODEL. */
  IloEnv env0 = solver0.getEnv();
  IloModel model0(env0);
  IloSchedulerEnv schedEnv0(env0);

  IloIntervalList blist(env0, 0, 100);
  blist.addPeriodicInterval(5, 2, 7, 100);
  schedEnv0.setBreakListParam(blist);

  IlcInt i=0;
  for (i = 0; i < nbOfWorkers; i++){
    workers[i] = IloUnaryResource(env0);
    model0.add(workers[i]);
  }

  IlcScheduler durSched(solver0);
  durSched.setDurable();
  solver0.extract(model0);

  /* CLOSE THE DURABLE SCHEDULER. */
  durSched.close();

  return durSched;
}

////////////////////////////////////////////
//
// TREATING A REQUEST
//
////////////////////////////////////////////

ILCGOAL3(TreatRequestIlc,
         IlcInt, reqIndex,
         IlcUnaryResource*, team,
         IlcInt, releaseDate){
```

```
  /* CREATING A SCHEDULE. */
  IloSolver solver = getSolver();
  IlcSchedule schedule(solver, 0, Horizon);
  /* LOCKING THE NEEDED RESOURCES. */
  schedule.lock(4, team);

  /* DEFINING THE PROBLEM. */
  IlcActivity* act = new (solver.getHeap()) IlcActivity[4];
  for (IlcInt i = 0; i < 4; i++){
    act[i] = IlcActivity(schedule, 1);
    solver.add(act[i].requires(team[i]));
  }
  solver.add(act[0].startsAfter(releaseDate));
  solver.add(act[1].startsAfterEnd(act[0]));
  solver.add(act[2].startsAfterEnd(act[0]));
  solver.add(act[3].startsAfterEnd(act[1]));
  solver.add(act[3].startsAfterEnd(act[2]));

  /* SOLVING THE PROBLEM. */
  solver.startNewSearch(IlcSetTimes(schedule));
  IloBool solved = solver.next();

  /* UNLOCKING THE USED COMPUTATIONAL RESOURCES. */
  schedule.unlock(4, team);

  if (solved) {
  /* ACCEPTING OR NOT. */
    IlcInt acceptDate = releaseDate + 5;
    IlcInt dueDate = act[3].getEndMin();
    if (dueDate > acceptDate){
      // Not accepting :
      solver.out() << "request " << reqIndex << ": refused " << endl;
      solver.restartSearch();
    }
    else {
      // Accepting :
      solver.out() << "request " << reqIndex
              << ": promised at " << act[3].getEndMin()
              << endl;
    }
  }
  else
    solver.out() << "request " << reqIndex
            << ": not solved " << endl;
  solver.endSearch();
  return 0;
}

ILOCPGOALWRAPPER4(TreatRequest, solver,
                  IlcScheduler, dSched,
                  IloInt, reqIndex,
                  IloUnaryResource*, team,
                  IloInt, releaseDate) {

  IlcUnaryResource* workers = new (solver.getHeap()) IlcUnaryResource[4];
  for (IlcInt i = 0; i < 4; i++)
    workers[i] = dSched.getResource(team[i]);
```

```
  return TreatRequestIlc(solver, reqIndex, workers, releaseDate);
}


////////////////////////////////////////
//
// DEFINING THE REQUESTS
//
////////////////////////////////////////
const IloInt HowManyReqs = 20;

IloInt StreamOfReqs [20] [2] = { /* pairs type releaseDate */
  {1, 5},    // request 0
  {2, 15},   // request 1
  {2, 7},    // request 2
  {1, 24},   // request 3
  {2, 2},    // request 4
  {1, 10},   // request 5
  {1, 25},   // request 6
  {1, 17},   // request 7
  {2, 8},    // request 8
  {1, 15},   // request 9
  {2, 21},   // request 10
  {2, 5},    // request 11
  {1, 15},   // request 12
  {2, 17},   // request 13
  {2, 24},   // request 14
  {2, 2},    // request 15
  {1, 10},   // request 16
  {2, 28},   // request 17
  {1, 17},   // request 18
  {2, 10}    // request 19
};

int main() {
  const IloInt nbOfWorkers = 8;
  IloUnaryResource workers[nbOfWorkers];
  IloUnaryResource* firstTeam = workers;
  IloUnaryResource* secondTeam = workers + 4;

  /* CREATING THE DURABLE RESOURCES. */
  IloEnv env0;
  IloSolver solver0(env0);
  IlcScheduler durableSched =
    CreateDurableResources(solver0, nbOfWorkers, workers);

  /* TREATING THE REQUESTS. */
  for (IloInt j = 0; j < HowManyReqs; j++){
    // Reading a request :
    IloInt requestType = StreamOfReqs[j][0];
    IloInt releaseDate = StreamOfReqs[j][1];

    IloEnv env;
    IloSolver solver(env);

    // Solving the specific problem associated with the request:
    IloUnaryResource* team = (requestType == 1 ?
                              firstTeam :
```

```
                                secondTeam);
    solver.solve(TreatRequest(env, durableSched, j, team, releaseDate));
    env.end();
  }

  /* PRINTING THE SCHEDULE OF WORKERS. */
  IlcScheduler scheduler0(solver0);
  IloSchedulerSolution solution(env0);

  for (IloInt i = 0; i < nbOfWorkers; i++){
    solver0.out() << endl << "worker" << i << ": " << endl;
    solution.store(workers[i], scheduler0);
    for (IloNumToNumStepFunctionCursor curs(solution.getLevelMin(workers[i]));
         curs.ok();
         ++curs)
      solver0.out() << curs.getSegmentMin() << " .. "
                    << IloMin(Horizon, (curs.getSegmentMax() - 1)) << " : "
                    << (curs.getValue() == 1 ? "On" : "Off")
                    << endl;
  }

  env0.end();
  return 0;
}

////////////////////////////////////////////
//
// RESULTS
//
////////////////////////////////////////////

/* durable
   request 0: promised at 10
   request 1: promised at 18
   request 2: promised at 10
   request 3: promised at 29
   request 4: promised at 5
   request 5: promised at 15
   request 6: promised at 30
   request 7: promised at 22
   request 8: promised at 11
   request 9: promised at 18
   request 10: promised at 24
   request 11: refused
   request 12: promised at 19
   request 13: promised at 22
   request 14: promised at 29
   request 15: refused
   request 16: refused
   request 17: not solved
   request 18: refused
   request 19: promised at 15

   worker0:
   0 .. 6 : Off
   7 .. 7 : On
   8 .. 9 : Off
   10 .. 10 : On
```

```
11 .. 14 : Off
15 .. 17 : On
18 .. 23 : Off
24 .. 25 : On
26 .. 30 : Off

worker1:
0 .. 7 : Off
8 .. 8 : On
9 .. 10 : Off
11 .. 11 : On
12 .. 15 : Off
16 .. 18 : On
19 .. 24 : Off
25 .. 25 : On
26 .. 27 : Off
28 .. 28 : On
29 .. 30 : Off

worker2:
0 .. 7 : Off
8 .. 8 : On
9 .. 10 : Off
11 .. 11 : On
12 .. 15 : Off
16 .. 18 : On
19 .. 24 : Off
25 .. 25 : On
26 .. 27 : Off
28 .. 28 : On
29 .. 30 : Off

worker3:
0 .. 8 : Off
9 .. 9 : On
10 .. 13 : Off
14 .. 14 : On
15 .. 16 : Off
17 .. 18 : On
19 .. 20 : Off
21 .. 21 : On
22 .. 27 : Off
28 .. 29 : On
30 .. 30 : Off

worker4:
0 .. 1 : Off
2 .. 2 : On
3 .. 6 : Off
7 .. 8 : On
9 .. 9 : Off
10 .. 10 : On
11 .. 14 : Off
15 .. 15 : On
16 .. 16 : Off
17 .. 17 : On
18 .. 20 : Off
21 .. 21 : On
```

```
   22 .. 23 : Off
   24 .. 24 : On
   25 .. 30 : Off

   worker5:
   0 .. 2 : Off
   3 .. 3 : On
   4 .. 7 : Off
   8 .. 9 : On
   10 .. 10 : Off
   11 .. 11 : On
   12 .. 15 : Off
   16 .. 16 : On
   17 .. 17 : Off
   18 .. 18 : On
   19 .. 21 : Off
   22 .. 22 : On
   23 .. 24 : Off
   25 .. 25 : On
   26 .. 30 : Off

   worker6:
   0 .. 2 : Off
   3 .. 3 : On
   4 .. 7 : Off
   8 .. 9 : On
   10 .. 10 : Off
   11 .. 11 : On
   12 .. 15 : Off
   16 .. 16 : On
   17 .. 17 : Off
   18 .. 18 : On
   19 .. 21 : Off
   22 .. 22 : On
   23 .. 24 : Off
   25 .. 25 : On
   26 .. 30 : Off

   worker7:
   0 .. 3 : Off
   4 .. 4 : On
   5 .. 8 : Off
   9 .. 10 : On
   11 .. 13 : Off
   14 .. 14 : On
   15 .. 16 : Off
   17 .. 17 : On
   18 .. 20 : Off
   21 .. 21 : On
   22 .. 22 : Off
   23 .. 23 : On
   24 .. 27 : Off
   28 .. 28 : On
   29 .. 30 : Off
*/
```

C H A P T E R

25

Using The Trace Facilities to Relax a Model

In this chapter we revisit the bridge problem from Chapter 14 and suppose that we have a targeted due date for each activity. This poses no specific problems, provided it is possible to find a schedule that satisfies all these targeted due dates. However, if it is impossible to simultaneously satisfy all the due dates, we say the problem is *overconstrained*. In this chapter, we present a fast way to build a schedule for an overconstrained problem. We assume that the major criterion for such a schedule is that it must be produced very quickly. In Chapter 31, *Handling an Overconstrained Problem: Adding Due Dates to the Bridge Problem*, an overconstrained problem is solved by using Solver functions to change the model, an approach that may yield a "better" solution, but that can also be more time-consuming for large problems.

The algorithm used here is based on *list scheduling*: at each step, an activity is selected from the list of non-scheduled activities and scheduled at its earliest possible time. If everything is all right, the process continues until all activities are scheduled. It may happen, however, that at some step in this procedure, the current, partial schedule becomes inconsistent with the set of constraints of the problem. Given such a situation, there is no hope to find a complete schedule by continuing to extend the partial schedule. In standard, non-deterministic search, a backtrack would occur at this point, undoing a previous decision in an attempt to re-establish a consistent partial schedule. Our approach here is different. Instead of backtracking, the algorithm analyses the reason that the partial schedule is inconsistent, called the "fail reason," and identifies a constraint that can be relaxed in order to restore consistency to the current partial solution. Note the contrast between this approach and standard, non-deterministic search. Rather than backtracking to search for a valid partial

schedule, we relax a problem constraint, changing the problem to one for which the current partial schedule may be valid.

The main goal of this chapter is to demonstrate the use of the trace facilities in the analysis of the fail reason. The chapter also presents how a problem can be relaxed using Concert Technology facilities and a solution object as well as how activities can be selected using predicates and evaluators.

Describing the Problem

The problem addressed in this chapter is a workshop scheduling problem. The workshop employs four workers that can manufacture two types of products, product A and B. Each customer orders a product, specifying a delivery date. We assume that a customer will take delivery of their product up to one week after its original delivery date. If a product cannot be delivered by that time, it is cancelled.

The manufacturing process of the first product is composed of four activities, each activity taking one day to be performed. Each worker specializes in one of these four activities. The second product is manufactured by two activities. The processing time of both activities is two days and each requires a single worker. The first activity requires one worker from one team of two workers and the second activity requires one worker from the other team of two workers.

The goal, then, is to assign the activities to the workers, to schedule them, and to identify the orders that will be delivered late or that will be cancelled.

Defining the Problem, Designing a Model

The scheduling model contains only workable days (five days per week) so that the one week delivery date extension is five days in this model.

```
const IloInt DeadlineExtension = 5;
```

Resources

As in the job-shop scheduling problem discussed in early chapters, each worker is represented as a unary resource (class `IloUnaryResource`). We define two teams of workers; one member of the first team processes the first activity for product B, while the second activity for product B is processed by one member of the second team. The precedence enforcement level is set to `IloMediumHigh` in order to create the precedence graph that is used later to analyze the partial schedule. The capacity enforcement level is left to its default value (`IloBasic`). In particular, we do not use the edge finder as it can identify

global inconsistencies that cannot easily be analyzed (see Analyzing a Fail Reason: Selecting the Guilty Activity).

```
/* DEFINING THE RESOURCES */
IloSchedulerEnv schedEnv(env);
schedEnv.getResourceParam().
  setPrecedenceEnforcement(IloMediumHigh);
IloUnaryResource* workers = new (env) IloUnaryResource[4];
workers[0] = IloUnaryResource(env, “Jim”);
workers[1] = IloUnaryResource(env, “Joe”);
workers[2] = IloUnaryResource(env, “Jack”);
workers[3] = IloUnaryResource(env, “John”);
for(i=0; i<4; i++) {
  solution.add(workers[i]);
}

/* DEFINING THE INITIAL OCCUPATION */
workers[0].setInitialOccupation(3,6,1);
workers[1].setInitialOccupation(22,23,1);
workers[2].setInitialOccupation(12,17,1);
workers[3].setInitialOccupation(7,11,1);

IloAltResSet* teams = new (env) IloAltResSet[2];
teams[0] = IloAltResSet(env, 2, workers[0], workers[1]);
teams[0].setName(“Jim or Joe”);
teams[1] = IloAltResSet(env, 2, workers[2], workers[3]);
teams[1].setName(“Jack or John”);
```

Jobs and Activities

A new class `Job` is used to represent the activities and the constraints associated with the processing of a product (either product A or B). The data member `_jobModel` stores all the precedence and time-bound constraints related to the activities of the job. This *job model* is very useful for adding and removing jobs—the activities plus all the related constraints—to and from the problem model. This is done with the member functions `addToModel` and `removeFromModel`.

```
class Job {
private:
  IloModel _jobModel;
  IloIntVar _deadline;
  IloBool _isDeadlineRelaxed;
public:
  Job(const IloEnv env, const char* name, IloInt deadline);
  void addToModel(IloModel model) { model.add(_jobModel); }
  void removeFromModel(IloModel model) { model.remove(_jobModel); }
  void relaxDeadline();
  IloBool isDeadlineRelaxed() const { return _isDeadlineRelaxed; }
  const char* getName() const { return _jobModel.getName(); }
protected:
  IloIntVar getDeadlineVar() const { return _deadline; }
  IloModel getJobModel() const { return _jobModel; }
};
```

When a job is created, its deadline (the delivery date of the product) is not relaxed. The member function `relaxDeadline` can be used to relax it in the job model.

```
Job::Job(const IloEnv env,
         const char* name,
         IloInt deadline)
 : _jobModel(env, name),
   _deadline(env),
   _isDeadlineRelaxed(IlcFalse)
{
  _deadline.setLB(deadline);
  _deadline.setUB(deadline);
}

void Job::relaxDeadline() {
  if (!_isDeadlineRelaxed) {
    IloInt deadline = (IloInt)_deadline.getUB() + DeadlineExtension;
    _isDeadlineRelaxed = IlcTrue;
    _deadline.setLB(deadline);
    _deadline.setUB(deadline);
  }
}
```

The two classes `JobA` and `JobB` are subclasses of `Job`. In their respective constructors, the classes specify the constraints that define the job and add those constraints to the job model. Some objects are also added to the `IloSchedulerSolution` object. These will be used later in the analysis of the partial solutions.

```
// JOB A CLASS

class JobA : public Job {
public:
  JobA(const IloEnv env, const char* name, IloInt deadline,
       IloSchedulerSolution solution, IloUnaryResource* workers);
};

JobA::JobA(const IloEnv env, const char* name, IloInt deadline,
           IloSchedulerSolution solution, IloUnaryResource* workers)
: Job(env, name, deadline) {
  IloInt i;
  IloModel jobModel = getJobModel();
  IloInt numberOfActivities = 4;
  IloActivity* act = new (env) IloActivity[numberOfActivities];
  char buffer[128];
  for(i=0; i < numberOfActivities; i++) {
    sprintf(buffer, "%s(Act%ld)", name, i);
    act[i] = IloActivity(env, 1, buffer);
    act[i].setObject(this);
    IloResourceConstraint ct = act[i].requires(workers[i]);
    jobModel.add(ct);
    solution.add(act[i]);
    solution.add(ct);
  }
  jobModel.add(act[3].endsBefore(getDeadlineVar()));
  jobModel.add(act[1].startsAfterEnd(act[0]));
  jobModel.add(act[2].startsAfterEnd(act[0]));
  jobModel.add(act[3].startsAfterEnd(act[1]));
```

```
  jobModel.add(act[3].startsAfterEnd(act[2]));
}

// JOB B CLASS

class JobB : public Job {
public:
  JobB(const IloEnv env, const char* name, IloInt deadline,
       IloSchedulerSolution solution, IloAltResSet* teams);
};

JobB::JobB(const IloEnv env, const char* name, IloInt deadline,
           IloSchedulerSolution solution, IloAltResSet* teams)
: Job(env, name, deadline) {
  IloInt i;
  IloModel jobModel = getJobModel();
  IloInt numberOfActivities = 2;
  IloActivity* act = new (env) IloActivity[numberOfActivities];
  char buffer[128];
  for(i=0; i < numberOfActivities; i++) {
    sprintf(buffer, "%s(Act%ld)", name, i);
    act[i] = IloActivity(env, 2, buffer);
    act[i].setObject(this);
    IloResourceConstraint ct = act[i].requires(teams[i]);
    jobModel.add(ct);
    solution.add(act[i]);
    solution.add(ct);
  }
  jobModel.add(act[1].endsBefore(getDeadlineVar()));
  jobModel.add(act[1].startsAfterEnd(act[0]));
}
```

Notice that the job instance is linked to each activity. When we handle an activity, it is easy to get the job to which it belongs by calling `getJob`.

```
Job* getJob(IloActivity act) {
  return (Job*)act.getObject();
}

Job* getJob(IlcActivity act) {
  return (Job*)act.getObject();
}
```

Solving the Problem: Analyzing a Fail

This section describes the list scheduling algorithm in detail.

List Scheduling

The list scheduling algorithm is a deterministic greedy algorithm: it creates no choice points. The heuristic rule to build the schedule is very simple: the activity that can start the earliest

time with the earliest maximum end time is selected and its start time is fixed to its minimum value. If the activity requires an alternative resource, one possible resource is arbitrarily selected.

```
ILCGOAL0(ListScheduling) {
  IloSolver solver = getSolver();
  IlcScheduler schedule(solver);

  IloSelector<IlcActivity,IlcSchedule> actSelector =
    IloBestSelector<IlcActivity,IlcSchedule>
    (!IlcActivityStartVarBoundPredicate(solver) ||
IsInUnselectedAltRCPredicate(solver),

IloComposeLexical(IlcActivityStartMinEvaluator(solver).makeLessThanComparator()
,
                  IlcActivityEndMaxEvaluator(solver).makeLessThanComparator()));

  IlcActivity act;
  if (actSelector.select(act, schedule)) {
    act.setStartTime(act.getStartMin());
    for(IlcAltResConstraintIterator ite(act); ite.ok(); ++ite) {
      IlcAltResConstraint altResCt = *ite;
      IlcAltResSet altResSet = altResCt.getAltResSet();
      IlcInt i;
      for(i=0; i<altResSet.getSize(); ++i) {
        IlcResource resource = altResSet[i];
        if (altResCt.isPossible(resource)) {
          altResCt.setSelected(resource);
          break;
        }
      }
    }
    return this;
  }
  else return 0;
}

ILOCPGOALWRAPPER0(IloListScheduling, solver) {
  return ListScheduling(solver);
}
```

What is a Fail Reason?

The list scheduling goal can fail before finding a solution. This fail means that the search engine has detected that the initial constraints and the decisions taken to build the partial schedule are inconsistent. It is impossible to continue extending the partial schedule to a consistent global schedule of the original problem. It is very difficult to identify the reason for a failure. While a failure is triggered by one particular object, the underlying reason for the failure generally results from the interaction of several Scheduler objects.

Scheduler trace facilities provide accessors for information related to a failure. For example, a failure is triggered when the domain of the start variable of an activity `act` becomes empty. In such a situation, the member function `getCurrentActivity1` of the class

`IlcSchedulerTraceI` returns `act`. We refer to `act` as the *guilty* activity. This does not mean that relaxing the constraints on this activity will make the problem solvable, but identification of the guilty activity does help us make a heuristic decision.

Analyzing a Fail Reason: Selecting the Guilty Activity

The fails are analyzed by the class `FailAnalysisI` that is a subclass of `IlcSchedulerTraceI`. The fail analysis must be initialized exactly as the printed trace (refer to *Part I, Getting Started with Scheduler* for additional information about printing the trace).

```
ILOCPTRACEWRAPPER1(ListSchedulingWithFailAnalysis, solver,
                   FailAnalysisResult&, analysisResult) {
  IlcScheduler sched(solver);
  solver.setTraceMode(IlcTrue);
  IlcSchedulerTrace analysis = FailAnalysis(sched, analysisResult);
  analysis.traceAllFailures();
  analysis.traceAllActivities();
  analysis.traceAllResources();
}
```

The functions `traceAllFailures`, `traceAllActivities`, `traceAllResources` initialize the search engine so that during the search, it maintains information related to a fail caused by an activity or resource. Each time a fail occurs, the virtual function `IlcSchedulerTraceI::failManager` is called. We overload this function in `FailAnalysisI` to implement the code that analyzes the fail.

A fail analysis object must be created when starting a new search and exists only during the search. Therefore, we create a class that stores the information found when analyzing the fail. The following class stores the guilty activity and the partial schedule at the time of failure.

```
class FailAnalysisResult {
  IloSchedulerSolution _solution;
  Job* _guiltyJob;
public:
  FailAnalysisResult(IloEnv env) : _solution(env), _guiltyJob(0) {}
  void setGuiltyJob(Job* job) { _guiltyJob = job; }
  void reset() { _guiltyJob = 0; }
  IloSchedulerSolution getSolution() const { return _solution; }
  IlcBool hasGuiltyJob() const { return _guiltyJob != 0; }
  Job* getGuiltyJob() const { return _guiltyJob; }
};
```

An instance of this class is given to the constructor of the class `FailAnalysisI`.

```
class FailAnalysisI : public IlcSchedulerTraceI {
  IloSolver _solver;
  IlcScheduler _schedule;
  FailAnalysisResult& _analysisResult;
public:
  FailAnalysisI(IlcScheduler sched,
```

```
                FailAnalysisResult& analysisResult)
    : IlcSchedulerTraceI(sched.getImpl()),
    _solver(sched.getSolver()),
    _schedule(sched),
    _analysisResult(analysisResult) {}
  virtual void failManager(IlcInt);
};
```

To find a guilty activity, we first look for an activity, a resource constraint, or an alternative resource constraint that could be related to the fail. If we find such an object, the guilty activity is directly determined. Otherwise, if there is a resource and time interval related to the failure, we look for an activity that can be processed by the resource during the time interval. Heuristically, we choose the guilty activity as the one with minimum slack. If there is no information related to the failure, the instance of `FailAnalysisResult` is not updated. The guilty job is the job to which the guilty activity belongs.

```
void FailAnalysisI::failManager(IlcInt) {
  IlcActivity act = getCurrentActivity1();
  if (act.getImpl() == 0) {
    IlcResourceConstraint rct = getCurrentResourceConstraint1();
    if (rct.getImpl())
      act = rct.getActivity();
    else {
      IlcAltResConstraint altRct = getCurrentAltResConstraint();
      if (altRct.getImpl()) {
        act = altRct.getActivity();
      }
      else {
        if (getCurrentResource().getImpl() == 0) return;
        IlcInt t1 = getCurrentTimeMin();
        IlcInt t2 = getCurrentTimeMax();

                        IloTranslator<IlcActivity, IlcResourceConstraint> ac =
                             IlcActivityResourceConstraintTranslator(_solver);
                      IloPredicate<IlcResourceConstraint> containsTimePeriod =
          (IlcActivityStartMinEvaluator(_solver) << ac) <= t1 &&
          (IlcActivityEndMaxEvaluator(_solver)  << ac) >= t2;
                              IloEvaluator<IlcResourceConstraint> slackEval =
          (IlcActivityEndMaxEvaluator(_solver) -
                                        IlcActivityStartMinEvaluator(_solver) -
                           IlcActivityDurationMinEvaluator(_solver)) << ac;
                        IloBestSelector<IlcResourceConstraint, IlcResource>
                  sel1(containsTimePeriod, slackEval.makeLessThanComparator());

        if (sel1.select(rct, getCurrentResource().getImpl())) {
          act = rct.getActivity();
        } else {
                     IloPredicate<IlcResourceConstraint> intersectsTimePeriod =
            (IlcActivityEndMaxEvaluator(_solver) << ac) >= t1 ||
                          (IlcActivityStartMinEvaluator(_solver) << ac) <= t2;
                           IloBestSelector<IlcResourceConstraint,IlcResource>
               sel2(intersectsTimePeriod, slackEval.makeLessThanComparator());

          if (sel2.select(rct, getCurrentResource().getImpl())) {
            act = rct.getActivity();
          }
```

```
        }
      }
    }
  }
  if (act.getImpl())
    _analysisResult.setGuiltyJob(getJob(act));
  _analysisResult.getSolution().store(_schedule);
}
```

Notice that we can generalize this fail analysis when the search goal is not deterministic and may meet several failures. However, in order to have a fast algorithm, a search limit should be used (see `IloLimitSearch` in the *IBM ILOG Solver Reference Manual*). All the guilty jobs found within this search limit can be stored in the analysis result object and the actual guilty job is the job that can be identified, for instance, as the one that is guilty the greatest number of times.

Relaxing the Problem

As long as a solution is not found, the list scheduling algorithm is followed. In case of a failure, the fail result object is used to identify the guilty job. The deadline of the guilty job is then relaxed if it has not been previously relaxed. If the deadline has been previously relaxed, the job is removed from the model. If no guilty job is found by the trace analysis, another heuristic decision should be taken. To keep this example simple, we have not implemented such a decision.

```
    IloCPTrace initListSchedulingWithFailAnalysis =
ListSchedulingWithFailAnalysis(env, analysisResult);
    solver.addTrace(initListSchedulingWithFailAnalysis);
    IloGoal goal = IloListScheduling(env);

    while (!solver.solve(goal)) {
      if (!analysisResult.hasGuiltyJob())
        throw Exception(“Cannot give a solution to the problem”);
      NewModelFromPartialSolution(analysisResult.getSolution());
      Job* guiltyJob = analysisResult.getGuiltyJob();
      if (!guiltyJob->isDeadlineRelaxed()) {
        guiltyJob->relaxDeadline();
        solver.out() << guiltyJob->getName()
                     << “ set at late deadline” << endl;
      }
      else {
        guiltyJob->removeFromModel(model);
        solver.out() << guiltyJob->getName()
                     << “ removed from the model” << endl;
      }
    }
```

Every time we fail, we relax the problem and start again from the existing partial solution. We implement the restart by storing the partial solution in an `IloSchedulerSolution` object and using it to initialize the list scheduling solution after relaxing the problem. This initialization is performed in the function `NewModelFromPartialSolution`.

The initialization speeds up the construction of the list schedule for the new, relaxed model. We could assign the start times from the partial schedule; however, as we might be removing an already scheduled activity, such a policy might overly constrain the new schedule. Removing an already scheduled activity, for example, would result in a “hole” in the schedule that could be used to decrease the start time of another already scheduled activity. To avoid this problem, only the maximum start times of the activities of the partial schedule are fixed.

```
void NewModelFromPartialSolution(const IloSchedulerSolution& solution) {
  for(IloSchedulerSolution::ResourceIterator ite(solution);
      ite.ok();
      ++ite) {
    IloResource resource = *ite;
    if (resource.isUnaryResource() &&
        solution.hasPrecedenceInformation(resource)) {
      if (solution.hasSetupRC(resource)) {
        IloResourceConstraint rct = solution.getSetupRC(resource);
        for(;;) {
          IloActivity act = rct.getActivity();
          IloNum t = solution.getStartMin(act);
          act.setStartMax(t);
          if (solution.hasNextRC(rct))
            rct = solution.getNextRC(rct);
          else break;
        }
      }
    }
  }
}
```

Complete Program and Output

You can see the entire program `relgreed.cpp` here or view it in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

const IloInt NumberOfJobsA = 20;
const IloInt NumberOfJobsB = 12;

IloInt DeadlinesA []  = {  5, 15,  7, 24,  9, 10, 25, 17,  8, 15,
                          29,  2, 15, 17, 24,  7, 10, 28, 17, 10};
IloInt DeadlinesB []  = {  7, 18,  9, 27,  7, 10,
                          21,  8, 18, 16, 26, 17};

const IloInt DeadlineExtension = 5;

////////////////////////////////////////
//
// JOB CLASSES
//
```

```
////////////////////////////////////////

class Job {
private:
  IloModel _jobModel;
  IloIntVar _deadline;
  IloBool _isDeadlineRelaxed;
public:
  Job(const IloEnv env, const char* name, IloInt deadline);
  void addToModel(IloModel model) { model.add(_jobModel); }
  void removeFromModel(IloModel model) { model.remove(_jobModel); }
  void relaxDeadline();
  IloBool isDeadlineRelaxed() const { return _isDeadlineRelaxed; }
  const char* getName() const { return _jobModel.getName(); }
protected:
  IloIntVar getDeadlineVar() const { return _deadline; }
  IloModel getJobModel() const { return _jobModel; }
};

Job::Job(const IloEnv env,
         const char* name,
         IloInt deadline)
 : _jobModel(env, name),
   _deadline(env),
   _isDeadlineRelaxed(IlcFalse)
{
  _deadline.setLB(deadline);
  _deadline.setUB(deadline);
}

void Job::relaxDeadline() {
  if (!_isDeadlineRelaxed) {
    IloInt deadline = (IloInt)_deadline.getUB() + DeadlineExtension;
    _isDeadlineRelaxed = IlcTrue;
    _deadline.setLB(deadline);
    _deadline.setUB(deadline);
  }
}

Job* getJob(IloActivity act) {
  return (Job*)act.getObject();
}

Job* getJob(IlcActivity act) {
  return (Job*)act.getObject();
}

// JOB A CLASS

class JobA : public Job {
public:
  JobA(const IloEnv env, const char* name, IloInt deadline,
       IloSchedulerSolution solution, IloUnaryResource* workers);
};

JobA::JobA(const IloEnv env, const char* name, IloInt deadline,
           IloSchedulerSolution solution, IloUnaryResource* workers)
: Job(env, name, deadline) {
```

```
  IloInt i;
  IloModel jobModel = getJobModel();
  IloInt numberOfActivities = 4;
  IloActivity* act = new (env) IloActivity[numberOfActivities];
  char buffer[128];
  for(i=0; i < numberOfActivities; i++) {
    sprintf(buffer, "%s(Act%ld)", name, i);
    act[i] = IloActivity(env, 1, buffer);
    act[i].setObject(this);
    IloResourceConstraint ct = act[i].requires(workers[i]);
    jobModel.add(ct);
    solution.add(act[i]);
    solution.add(ct);
  }
  jobModel.add(act[3].endsBefore(getDeadlineVar()));
  jobModel.add(act[1].startsAfterEnd(act[0]));
  jobModel.add(act[2].startsAfterEnd(act[0]));
  jobModel.add(act[3].startsAfterEnd(act[1]));
  jobModel.add(act[3].startsAfterEnd(act[2]));
}

// JOB B CLASS

class JobB : public Job {
public:
  JobB(const IloEnv env, const char* name, IloInt deadline,
       IloSchedulerSolution solution, IloAltResSet* teams);
};

JobB::JobB(const IloEnv env, const char* name, IloInt deadline,
           IloSchedulerSolution solution, IloAltResSet* teams)
: Job(env, name, deadline) {
  IloInt i;
  IloModel jobModel = getJobModel();
  IloInt numberOfActivities = 2;
  IloActivity* act = new (env) IloActivity[numberOfActivities];
  char buffer[128];
  for(i=0; i < numberOfActivities; i++) {
    sprintf(buffer, "%s(Act%ld)", name, i);
    act[i] = IloActivity(env, 2, buffer);
    act[i].setObject(this);
    IloResourceConstraint ct = act[i].requires(teams[i]);
    jobModel.add(ct);
    solution.add(act[i]);
    solution.add(ct);
  }
  jobModel.add(act[1].endsBefore(getDeadlineVar()));
  jobModel.add(act[1].startsAfterEnd(act[0]));
}

////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////

IloModel DefineModel(IloEnv& env,
                     IloInt numberOfJobsA,
```

```
                    IloInt numberOfJobsB,
                    IloInt* deadlinesA,
                    IloInt* deadlinesB,
                    IloSchedulerSolution solution) {
  IloInt i;
  IloModel model(env);

  /* DEFINING THE RESOURCES */
  IloSchedulerEnv schedEnv(env);
  schedEnv.getResourceParam().
    setPrecedenceEnforcement(IloMediumHigh);
  IloUnaryResource* workers = new (env) IloUnaryResource[4];
  workers[0] = IloUnaryResource(env, "Jim");
  workers[1] = IloUnaryResource(env, "Joe");
  workers[2] = IloUnaryResource(env, "Jack");
  workers[3] = IloUnaryResource(env, "John");
  for(i=0; i<4; i++) {
    solution.add(workers[i]);
  }

  /* DEFINING THE INITIAL OCCUPATION */
  workers[0].setInitialOccupation(3,6,1);
  workers[1].setInitialOccupation(22,23,1);
  workers[2].setInitialOccupation(12,17,1);
  workers[3].setInitialOccupation(7,11,1);

  IloAltResSet* teams = new (env) IloAltResSet[2];
  teams[0] = IloAltResSet(env, 2, workers[0], workers[1]);
  teams[0].setName("Jim or Joe");
  teams[1] = IloAltResSet(env, 2, workers[2], workers[3]);
  teams[1].setName("Jack or John");

  /* DEFINING THE JOBS */
  char buffer[128];
  for(i=0; i < numberOfJobsA; i++) {
    sprintf(buffer, "JobA%ld", i);
    JobA* job = new (env) JobA(env, buffer, deadlinesA[i],
                               solution, workers);
    job->addToModel(model);
  }
  for(i=0; i < numberOfJobsB; i++) {
    sprintf(buffer, "JobB%ld", i);
    JobB* job = new (env) JobB(env, buffer, deadlinesB[i],
                               solution, teams);
    job->addToModel(model);
  }
  return model;
}


////////////////////////////////////////
//
// MODIFYING THE MODEL
//
////////////////////////////////////////

void NewModelFromPartialSolution(const IloSchedulerSolution& solution) {
  for(IloSchedulerSolution::ResourceIterator ite(solution);
```

```
      ite.ok();
      ++ite) {
    IloResource resource = *ite;
    if (resource.isUnaryResource() &&
        solution.hasPrecedenceInformation(resource)) {
      if (solution.hasSetupRC(resource)) {
        IloResourceConstraint rct = solution.getSetupRC(resource);
        for(;;) {
          IloActivity act = rct.getActivity();
          IloNum t = solution.getStartMin(act);
          act.setStartMax(t);
          if (solution.hasNextRC(rct))
            rct = solution.getNextRC(rct);
          else break;
        }
      }
    }
  }
}


////////////////////////////////////////
//
// TRACE BASED FAIL DETECTION
//
////////////////////////////////////////

class FailAnalysisResult {
  IloSchedulerSolution _solution;
  Job* _guiltyJob;
public:
  FailAnalysisResult(IloEnv env) : _solution(env), _guiltyJob(0) {}
  void setGuiltyJob(Job* job) { _guiltyJob = job; }
  void reset() { _guiltyJob = 0; }
  IloSchedulerSolution getSolution() const { return _solution; }
  IlcBool hasGuiltyJob() const { return _guiltyJob != 0; }
  Job* getGuiltyJob() const { return _guiltyJob; }
};

class FailAnalysisI : public IlcSchedulerTraceI {
  IloSolver _solver;
  IlcScheduler _schedule;
  FailAnalysisResult& _analysisResult;
public:
  FailAnalysisI(IlcScheduler sched,
                FailAnalysisResult& analysisResult)
    : IlcSchedulerTraceI(sched.getImpl()),
    _solver(sched.getSolver()),
    _schedule(sched),
    _analysisResult(analysisResult) {}
  virtual void failManager(IlcInt);
};

void FailAnalysisI::failManager(IlcInt) {
  IlcActivity act = getCurrentActivity1();
  if (act.getImpl() == 0) {
    IlcResourceConstraint rct = getCurrentResourceConstraint1();
    if (rct.getImpl())
```

```
      act = rct.getActivity();
    else {
      IlcAltResConstraint altRct = getCurrentAltResConstraint();
      if (altRct.getImpl()) {
        act = altRct.getActivity();
      }
      else {
        if (getCurrentResource().getImpl() == 0) return;
        IlcInt t1 = getCurrentTimeMin();
        IlcInt t2 = getCurrentTimeMax();

                        IloTranslator<IlcActivity, IlcResourceConstraint> ac =
                             IlcActivityResourceConstraintTranslator(_solver);
                      IloPredicate<IlcResourceConstraint> containsTimePeriod =
          (IlcActivityStartMinEvaluator(_solver) << ac) <= t1 &&
          (IlcActivityEndMaxEvaluator(_solver)  << ac) >= t2;
                              IloEvaluator<IlcResourceConstraint> slackEval =
          (IlcActivityEndMaxEvaluator(_solver) -
                                      IlcActivityStartMinEvaluator(_solver) -
                            IlcActivityDurationMinEvaluator(_solver)) << ac;
                         IloBestSelector<IlcResourceConstraint, IlcResource>
                  sel1(containsTimePeriod, slackEval.makeLessThanComparator());

        if (sel1.select(rct, getCurrentResource().getImpl())) {
          act = rct.getActivity();
        } else {
                    IloPredicate<IlcResourceConstraint> intersectsTimePeriod =
            (IlcActivityEndMaxEvaluator(_solver) << ac) >= t1 ||
                          (IlcActivityStartMinEvaluator(_solver) << ac) <= t2;
                           IloBestSelector<IlcResourceConstraint,IlcResource>
                sel2(intersectsTimePeriod, slackEval.makeLessThanComparator());

          if (sel2.select(rct, getCurrentResource().getImpl())) {
            act = rct.getActivity();
          }
        }
      }
    }
  }
  if (act.getImpl())
    _analysisResult.setGuiltyJob(getJob(act));
  _analysisResult.getSolution().store(_schedule);
}

IlcSchedulerTrace FailAnalysis(IlcScheduler schedule,
                               FailAnalysisResult& analysisResult) {
  IloSolver solver = schedule.getSolver();
  analysisResult.reset();
  return new (solver.getHeap()) FailAnalysisI(schedule,
                                              analysisResult);
}


////////////////////////////////////////
//
// LIST SCHEDULING
//
////////////////////////////////////////
```

```
ILOPREDICATE0(IsInUnselectedAltRCPredicate,
                                                              IlcActivity, act) {
  for(IlcAltResConstraintIterator ite(act); ite.ok(); ++ite) {
    if (!(*ite).isResourceSelected())
      return IlcTrue;
  }
  return IlcFalse;
}

ILCGOAL0(ListScheduling) {
  IloSolver solver = getSolver();
  IlcScheduler schedule(solver);

  IloSelector<IlcActivity,IlcSchedule> actSelector =
    IloBestSelector<IlcActivity,IlcSchedule>
    (!IlcActivityStartVarBoundPredicate(solver) ||
IsInUnselectedAltRCPredicate(solver),

IloComposeLexical(IlcActivityStartMinEvaluator(solver).makeLessThanComparator()
,
                IlcActivityEndMaxEvaluator(solver).makeLessThanComparator()));

  IlcActivity act;
  if (actSelector.select(act, schedule)) {
    act.setStartTime(act.getStartMin());
    for(IlcAltResConstraintIterator ite(act); ite.ok(); ++ite) {
      IlcAltResConstraint altResCt = *ite;
      IlcAltResSet altResSet = altResCt.getAltResSet();
      IlcInt i;
      for(i=0; i<altResSet.getSize(); ++i) {
        IlcResource resource = altResSet[i];
        if (altResCt.isPossible(resource)) {
          altResCt.setSelected(resource);
          break;
        }
      }
    }
    return this;
  }
  else return 0;
}

ILOCPGOALWRAPPER0(IloListScheduling, solver) {
  return ListScheduling(solver);
}


ILOCPTRACEWRAPPER1(ListSchedulingWithFailAnalysis, solver,
                   FailAnalysisResult&, analysisResult) {
  IlcScheduler sched(solver);
  solver.setTraceMode(IlcTrue);
  IlcSchedulerTrace analysis = FailAnalysis(sched, analysisResult);
  analysis.traceAllFailures();
  analysis.traceAllActivities();
  analysis.traceAllResources();
}
```

```
////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////

void
PrintSolution(const IloSolver& solver)
{
  IlcScheduler scheduler(solver);
  for(IlcActivityIterator ite(scheduler); ite.ok(); ++ite) {
    IlcActivity act = *ite;
    solver.out() << act;
    for(IlcAltResConstraintIterator iteCt(act); iteCt.ok(); ++iteCt) {
      IlcAltResConstraint altResCt = *iteCt;
      solver.out() << " processed by " << altResCt.getSelected().getName();
    }
    solver.out() << endl;
  }
}


////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////

class Exception : public IloException {
public:
  Exception(const char* message) : IloException(message) {}
};

int main() {
  try {
    IloEnv env;

    FailAnalysisResult analysisResult(env);
    IloModel model = DefineModel(env,
                                 NumberOfJobsA,
                                 NumberOfJobsB,
                                 DeadlinesA,
                                 DeadlinesB,
                                 analysisResult.getSolution());

    IloSolver solver(env);
    solver.setFastRestartMode(IloTrue);
    solver.extract(model);



    IloCPTrace initListSchedulingWithFailAnalysis =
ListSchedulingWithFailAnalysis(env, analysisResult);
    solver.addTrace(initListSchedulingWithFailAnalysis);
    IloGoal goal = IloListScheduling(env);

    while (!solver.solve(goal)) {
      if (!analysisResult.hasGuiltyJob())
```

```
        throw Exception("Cannot give a solution to the problem");
      NewModelFromPartialSolution(analysisResult.getSolution());
      Job* guiltyJob = analysisResult.getGuiltyJob();
      if (!guiltyJob->isDeadlineRelaxed()) {
        guiltyJob->relaxDeadline();
        solver.out() << guiltyJob->getName()
                     << " set at late deadline" << endl;
      }
      else {
        guiltyJob->removeFromModel(model);
        solver.out() << guiltyJob->getName()
                     << " removed from the model" << endl;
      }
    }

    solver.out() << "Solution: " << endl;
    PrintSolution(solver);

    solver.printInformation();
    env.end();
  } catch (IloException& exc) {
    cout << exc << endl;
  }
  return 0;
}

////////////////////////////////////////////
//
// RESULTS
//
////////////////////////////////////////////
/*
JobA11 set at late deadline
JobB0 set at late deadline
JobA19 set at late deadline
JobA16 set at late deadline
JobA15 set at late deadline
JobA11 removed from the model
JobA8 set at late deadline
JobA5 set at late deadline
JobA4 set at late deadline
JobA2 set at late deadline
JobA15 removed from the model
JobA19 removed from the model
JobA16 removed from the model
JobA12 set at late deadline
JobA9 set at late deadline
JobA5 removed from the model
JobB9 set at late deadline
JobA18 set at late deadline
JobA13 set at late deadline
JobB1 set at late deadline
JobB0 removed from the model
JobA8 removed from the model
JobA4 removed from the model
JobB9 removed from the model
JobB6 set at late deadline
JobB1 removed from the model
```

```
JobB6 removed from the model
Solution:
JobB11(Act1) [14 -- 2 --> 16] processed by John
JobB11(Act0) [8 -- 2 --> 10] processed by Joe
JobB10(Act1) [24 -- 2 --> 26] processed by Jack
JobB10(Act0) [17 -- 2 --> 19] processed by Joe
JobB8(Act1) [16 -- 2 --> 18] processed by John
JobB8(Act0) [9 -- 2 --> 11] processed by Jim
JobB7(Act1) [4 -- 2 --> 6] processed by Jack
JobB7(Act0) [1 -- 2 --> 3] processed by Jim
JobB5(Act1) [7 -- 2 --> 9] processed by Jack
JobB5(Act0) [5 -- 2 --> 7] processed by Joe
JobB4(Act1) [2 -- 2 --> 4] processed by Jack
JobB4(Act0) [0 -- 2 --> 2] processed by Joe
JobB3(Act1) [25 -- 2 --> 27] processed by John
JobB3(Act0) [18 -- 2 --> 20] processed by Jim
JobB2(Act1) [5 -- 2 --> 7] processed by John
JobB2(Act0) [3 -- 2 --> 5] processed by Joe
JobA18(Act3) [20 -- 1 --> 21]
JobA18(Act2) [19 -- 1 --> 20]
JobA18(Act1) [14 -- 1 --> 15]
JobA18(Act0) [13 -- 1 --> 14]
JobA17(Act3) [27 -- 1 --> 28]
JobA17(Act2) [26 -- 1 --> 27]
JobA17(Act1) [21 -- 1 --> 22]
JobA17(Act0) [20 -- 1 --> 21]
JobA14(Act3) [22 -- 1 --> 23]
JobA14(Act2) [21 -- 1 --> 22]
JobA14(Act1) [16 -- 1 --> 17]
JobA14(Act0) [15 -- 1 --> 16]
JobA13(Act3) [21 -- 1 --> 22]
JobA13(Act2) [20 -- 1 --> 21]
JobA13(Act1) [15 -- 1 --> 16]
JobA13(Act0) [14 -- 1 --> 15]
JobA12(Act3) [18 -- 1 --> 19]
JobA12(Act2) [17 -- 1 --> 18]
JobA12(Act1) [12 -- 1 --> 13]
JobA12(Act0) [11 -- 1 --> 12]
JobA10(Act3) [28 -- 1 --> 29]
JobA10(Act2) [27 -- 1 --> 28]
JobA10(Act1) [23 -- 1 --> 24]
JobA10(Act0) [21 -- 1 --> 22]
JobA9(Act3) [19 -- 1 --> 20]
JobA9(Act2) [18 -- 1 --> 19]
JobA9(Act1) [13 -- 1 --> 14]
JobA9(Act0) [12 -- 1 --> 13]
JobA7(Act3) [13 -- 1 --> 14]
JobA7(Act2) [10 -- 1 --> 11]
JobA7(Act1) [11 -- 1 --> 12]
JobA7(Act0) [7 -- 1 --> 8]
JobA6(Act3) [24 -- 1 --> 25]
JobA6(Act2) [23 -- 1 --> 24]
JobA6(Act1) [20 -- 1 --> 21]
JobA6(Act0) [17 -- 1 --> 18]
JobA3(Act3) [23 -- 1 --> 24]
JobA3(Act2) [22 -- 1 --> 23]
JobA3(Act1) [19 -- 1 --> 20]
JobA3(Act0) [16 -- 1 --> 17]
```

```
JobA2(Act3) [11 -- 1 --> 12]
JobA2(Act2) [9 -- 1 --> 10]
JobA2(Act1) [7 -- 1 --> 8]
JobA2(Act0) [6 -- 1 --> 7]
JobA1(Act3) [12 -- 1 --> 13]
JobA1(Act2) [11 -- 1 --> 12]
JobA1(Act1) [10 -- 1 --> 11]
JobA1(Act0) [8 -- 1 --> 9]
JobA0(Act3) [3 -- 1 --> 4]
JobA0(Act2) [1 -- 1 --> 2]
JobA0(Act1) [2 -- 1 --> 3]
JobA0(Act0) [0 -- 1 --> 1]
*/
```

Part III

Local Search in Scheduler

This part of the manual examines ways to use, guide, and modify search within Scheduler.

Using the local search facilities of IBM© ILOG© Solver, it is possible to construct local search algorithms in Scheduler. In particular, rather than conducting your local search with *Solver* semantics (that is, referring to variables and values) you can conduct it using *Scheduler* semantics. This means that you can create neighborhoods and metaheuristics using Scheduler objects such as activities and resource constraints.

As the local search in Scheduler is based on the local search facilities in Solver, it is recommended that you have a strong understanding of the local search in Solver. See the *IBM ILOG Solver User's Manual*.

In Chapter 26, *Shuffling as Local Search in Scheduler,* we present an implementation of the shuffling technique using the local search infrastructure. This chapter demonstrates the creation of a custom neighborhood and its use within a standard pattern for local search implementation.

Chapter 27, *Tabu Search for the Jobshop Problem*, builds on the previous chapter by implementing a tabu search metaheuristic on top of one of the standard job shop scheduling neighborhoods from the research literature.

Chapter 28, *Tabu Search for the Jobshop Problem with Alternatives*, further extends the tabu search by introducing a second neighborhood consisting of moving a resource constraint from its existing resource to one of its other alternatives.

Chapter 29, *Large Neighborhood Search for the Jobshop Problem with Alternatives*, approaches the joshop problem with large neighborhood search. Two predefined and one custom neighborhood are used in the search.

C H A P T E R

26

Shuffling as Local Search in Scheduler

In this chapter, we implement the shuffling technique as a simple example of local search in Scheduler.

One of the standard, incomplete search techniques used in the Operations Research scheduling literature is called shuffling. Given a feasible solution to a jobshop scheduling problem, one variation of the shuffle technique randomly (with some probability) removes a subset of the edges on the critical path of the solution. The remaining edges define a partial solution and some standard constructive search technique is then used to try to extend the partial solution to a full solution. This is one iteration of the shuffle technique. Subsequent iterations begin with the solution found in the previous iteration.

Problem Description

The problem used here is the standard job shop scheduling problem as defined in Chapter 15, *Scheduling with Unary Resources: the Job-Shop Problem*. As we have seen the way in which the problem is modeled in previous chapters, we will move directly to the search mechanism.

Calculating the Critical Path

The shuffle technique requires a starting solution from which to calculate the first critical path. From this critical path, a subset of edges will be removed from the subsequent solution. We discuss finding the first solution in Finding the First Solution. In this section, we discuss the finding of the first solution, and assume that we have a feasible solution to the jobshop problem and need to find a randomly selected critical path in that solution.

One common way to model a solution to a job shop scheduling problem is as a set of nodes, corresponding to each activity, and a set of edges, corresponding to next relations in the precedence graph and to the precedence constraints in the original problem definition. The weight of each edge is the duration of the preceding activity. In such a model any path from a virtual source node to a virtual sink node of maximum length corresponds to a critical path of the solution.

Another way to visualize the critical path is using a Gantt chart such as the one in Figure 26.1. We represent the jobs by the activities with the same letter, so that activities B0, B1, and B2 are all in the same job. In this case, we have two critical paths: A0-B0-B1-B2 and C0-C1-B1-B2.

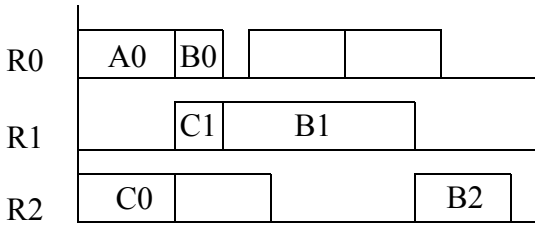


Figure 26.1 *Gantt Chart of Critical Activities*

Rather than making use of the graph representation, we calculate the critical path directly on the `IloActivity` instances stored in the `IloSchedulerSolution`. To do this we make use of the fact that any activity that is on the critical path must have its start time assigned when the makespan variable is assigned. We can easily identify all of these activities, sort them into an array in non-descending order of start time, and then, with a single pass through the array, pick a single critical path.

To implement this critical path calculation, we introduce a `CriticalPath` class, defined in the following code.

```
class CriticalPath {
private:
  IloInt _cpSize;
  IloArray<IloResourceConstraint> _cpArray;

  IloRandom _randGen;

  RCSortElement *getCriticalRCs(IloSchedulerSolution, IloInt&);
  void pickRandomCP(RCSortElement *, IloSchedulerSolution,IloInt);
  void addCPElement(IloResourceConstraint);

public:
  CriticalPath(IloEnv env) :
    _cpSize(0), _cpArray(env), _randGen(env, 98998) {}
  ~CriticalPath() {}

  IloArray<IloResourceConstraint>& computeCP(IloSchedulerSolution, IloInt&);
};
```

We proceed step-by-step, through the methods of this class. First, the primary public method is the `computeCP` method defined in the following code.

```
IloArray<IloResourceConstraint>& CriticalPath::computeCP(
       IloSchedulerSolution solution,
       IloInt& cpSize) {
  IloInt nbCriticalRCs = 0;
  RCSortElement *sortArray = getCriticalRCs(solution, nbCriticalRCs);

  // Pick the RCs in one (randomly selected) critical path
  pickRandomCP(sortArray, solution, nbCriticalRCs);
  delete [] sortArray;

  cpSize = _cpSize;
  return _cpArray;
}
```

The first call in this function, `getCriticalRCs`, returns an array of `IloResourceConstraint` instances sorted in non-descending order of start time. As this makes use of standard C++ techniques, we will not describe it in-depth (see the complete code for details).

The second function call, `pickRandomCP`, assigns the member data `_cpArray` to a randomly selected critical path. This function uses the sorted array of resource constraints to construct such a path. The code for `pickRandomCP` follows.

```
void CriticalPath::pickRandomCP(RCSortElement *sortArray,
IloSchedulerSolution solution,
IloInt nbCriticalRCs) {
  _cpSize = 0;
  IloNum endRC = 0;
  IloInt i = -1;
  while(i < nbCriticalRCs - 1) {
    // skip elements not on the same critical path as rc
    IloInt nextIndexStart = i + 1;
    while((nextIndexStart < nbCriticalRCs) &&
          (sortArray[nextIndexStart].getStartValue() < endRC))
      nextIndexStart++;

    // gather elements that are successors of rc on the critical
    // path. There may be more than one

    IloInt nextIndexEnd = nextIndexStart + 1;
    while((nextIndexEnd < nbCriticalRCs) &&
          (sortArray[nextIndexEnd].getStartValue() == endRC))
      nextIndexEnd++;

    if (nextIndexStart < nbCriticalRCs) {

      // At this point the elements between sortArray[nextIndexStart]
      // and sortArray[nextIndexEnd-1] inclusive are all successors to
      // rc on some critical path.
      // We randomly pick one of them.

      IloInt randIndex = nextIndexStart;
      IloInt indexDiff = nextIndexEnd - nextIndexStart;
      if (indexDiff > 1)
        randIndex += _randGen.getInt(indexDiff);

      IloResourceConstraint next = sortArray[randIndex].getRC();
      addCPElement(next);

      i = randIndex;
      endRC = solution.getEndMin(next.getActivity());
    }
    else
      i = nbCriticalRCs;
  }
}
```

Each iteration of the outer while-loop in `pickRandomCP` selects a single resource constraint from the `sortArray` to be a member of the randomly selected critical path. Essentially, given a resource constraint `rc` that has already been selected as part of the critical path, we

skip all the other resource constraints that start before the end time of `rc`. Clearly, these resource constraints cannot be on a critical path containing `rc`.

```
IloInt nextIndexStart = i + 1;
while((nextIndexStart < nbCriticalRCs) &&
      (sortArray[nextIndexStart].getStartValue() < endRC))
  nextIndexStart++;
```

We then gather the elements that have start times equal to the end time of `rc`.

```
IloInt nextIndexEnd = nextIndexStart + 1;
while((nextIndexEnd < nbCriticalRCs) &&
      (sortArray[nextIndexEnd].getStartValue() == endRC))
  nextIndexEnd++;
```

Finally, we randomly choose a resource constraint from those that start at the end of `rc`. This is the next resource constraint in the critical path.

```
IloInt randIndex = nextIndexStart;
IloInt indexDiff = nextIndexEnd - nextIndexStart;
if (indexDiff > 1)
  randIndex += _randGen.getInt(indexDiff);

IloResourceConstraint next = sortArray[randIndex].getRC();
addCPElement(next);
```

The outer while-loop then continues using the most recently selected resource constraint. We make use of the critical path to construct the moves in our neighborhood.

Solving the Problem

To solve the problem, we create an `IloSchedulerSolution`, find a first solution, and then create the shuffle neighborhood.

Creating an IloSchedulerSolution

In the local search infrastructure provided by IBM© ILOG© Solver, the `IloSolution` objects plays a key role as a data structure. The `IloSchedulerSolution` object plays the same role for local search in IBM ILOG Scheduler. In particular, it represents the Scheduler objects that define a solution as well as the changes to those objects that represent a neighbor.

Scheduler objects are richer than the variables that are the main elements of local search in Solver. For example, a resource constraint in a solution specifies the resource to which it is assigned, the set of successor resource constraints, the set of next resource constraints, etc. Because of this richer representation, it is necessary to specify precisely what is restored

when an IloSchedulerSolution is restored by the local search protocol (or by the IloRestoreSolution goal). In this case, we want to store the activities in the solution (because we need to access the start times for the critical path calculation) but we do not want to restore any of the information contained in the activities as we want our neighborhood to be based on the edges in the critical path, that is, on the next relations between resource constraints. We also want to store and restore those next relations between resource constraints. To do all this, we first add the activities to the solution in the DefineModel function.

```
solution.add(activity, IloRestoreNothing);
```

Note the second argument of the add function. This argument defines what is restored. All the information about the object will be stored in the solution, so it is still possible to inquire about the start times of activities but, because the second argument is IloRestoreNothing, no information about activities will be restored.

Rather than explicitly adding each activity to the local search solution, we simply create the local search solution by cloning the main solution object. Then we add each resource constraint, specifying that only the next relations should be restored. See the *IBM ILOG Scheduler Reference Manual* for full details about the parameterization of the restoration of IloSchedulerSolution instances.

```
/* CREATE LOCAL SEARCH SOLUTION */
IloSchedulerSolution lsSolution = globalSolution.makeClone(env);

for (IloIterator <IloResourceConstraint> iter(env); iter.ok(); ++iter)
  lsSolution.add(*iter, IloRestoreRCNext);
```

Finding the First Solution

As we are concerned primarily with illustrating local search in this example, we do not spend much effort in finding a good first solution. It should be noted however that in many cases, finding a good first solution is often important for good local search performance. We use the IloRankForward predefined goal and take the first feasible solution that is found.

```
IloGoal g = IloRankForward(env) && IloInstantiate(env, costVar);
solver.startNewSearch(g);

if(solver.next()) {
  IlcInt best = (IlcInt)solver.getValue(costVar);
  solver.out() << “Solution for Cost “ <<  best << endl;
  solver.printInformation();
  globalSolution.store(scheduler);
  lsSolution.store(scheduler);
}
```

After finding the solution, we call store on the `lsSolution` so that we are then ready to begin local search.

Creating the Shuffle Neighborhood

The primary part of implementing a local search technique in Scheduler is the creation of the appropriate neighborhood class (or classes) to describe the local search moves. The shuffle neighborhood is quite simple as it does not concern itself with making new search decisions, but rather with removing some of the decisions that define the critical path of the existing solution. Actually, in our implementation we take the opposite decision: we identify the set of next relations that we will preserve in the next search. This set defines a partial solution from which a constructive search will be done.

To construct the `ShuffleNHoodI` we subclass from the IBM ILOG Solver class `IloNHoodI`.

```
class ShuffleNHoodI : public IloNHoodI {
private:
  IloNumVar _costVar;
  IloNum _prob;

  CriticalPath _cp;

  IloSchedulerSolution _delta;
  IloRandom _randGen;

public:
  ShuffleNHoodI(IloEnv env, IloNumVar cost,
IloNum probability, const char *name = 0)
    : IloNHoodI (env, name),
      _costVar(cost), _prob(probability),
      _cp(env), _delta(0), _randGen(env, 98998) {}
  ~ShuffleNHoodI() {}

  void        start       (IloSolver solver, IloSolution soln);

  IloSolution define      (IloSolver solver, IloInt) {
    assert(_delta.getImpl() != 0);
    assert(_delta.getSolution().getImpl() != 0);
    return _delta.getSolution().makeClone(solver.getEnv());
  }

  IloInt      getSize     (IloSolver) const { return 1; }
};
```

As we have a relatively simple neighborhood, we only override three of the virtual functions of `IloNHoodI`.

The `define` method is called by the local search protocol and must return an `IloSolution` representing the move that should be performed. It is a good idea to make this function as small and as fast as possible as it is called many times during the local search. In our case,

we simply return a clone of the _delta solution, which we previously computed (in start; code follows).

The other function is getSize which returns the number of neighbors in the neighborhood. In our case, we have only one neighbor.

Most of the work in ShuffleNHoodI is done in the start method. The method begins by calculating the critical path, using the CriticalPath object presented above.

```
IloInt cpSize = 0;
IloArray<IloResourceConstraint> cpArray = _cp.computeCP(solution, cpSize);
```

The second step is to create an IloSchedulerSolution to represent the move that will be investigated. We do this simply by cloning the current solution.

```
if (_delta.getImpl() != 0)
  _delta.end();
_delta = solution.makeClone(env);
```

The critical path is represented as an array of IloResourceConstraint objects such that adjacent elements of the array are adjacent elements of the critical path. Recall, however, that there are two ways in which a pair of resource constraints can be adjacent on the critical path: they may execute consecutively on the same resource or they may correspond to consecutive activities in the same job. Clearly, we cannot remove edges of the latter type as they are part of the original problem definition. Therefore, we examine an adjacent pair of resource constraints on the critical path and ensure that they are also an adjacent pair on a resource. Then, based on a probability, we decide whether to preserve this adjacent pair in the next solution.

```
for(IloInt i = 0; i < cpSize - 1; ++i) {
  IloResourceConstraint before = cpArray[i];
  IloResourceConstraint after = cpArray[i + 1];

  if (solution.hasNextRC(before) &&
(after.getImpl() == solution.getNextRC(before).getImpl()) &&
(_randGen.getFloat() > _prob)) {
    // Remove resource constraint from delta. This has the effect of
    // ensuring that this edge will be in the next solution
    _delta.remove(before);
  }
}
```

Finally, all the other resource constraints in the solution, regardless of whether they are on the critical path or not, have their next relations removed from delta.

```
for(IloSchedulerSolution::ResourceConstraintIterator iter3(_delta);
    iter3.ok(); ++iter3) {
  IloResourceConstraint rc = *iter3;
  if (_delta.hasNextRC(rc))
    _delta.unsetNext(rc, _delta.getNextRC(rc));
}
```

It is important to understand the difference between removing the resource constraint from `_delta` and removing the next relation from a resource constraint in `_delta`. With the local search protocol, removing the resource constraint means that the values in the current solution for that resource constraint will be restored. Removing the next relation means that no next relation is restored. This is a critical distinction and can be best understood from the perspective of the local search protocol, which is faced with two solution objects: one containing the previous solution and the other containing the changes (the “delta”) that must be done to the previous solution to visit a specific neighbor. The protocol first restores the elements of the previous solution that are not in the delta. It then restores the objects in the delta. So the resource constraints that are not in the delta are restored to their previous values. The resource constraints that are in the delta, but have no specified next relation are restored without specifying the next relations. Because the next relations are not defined, we have space to subsequently search for a new solution.

Finally, since the delta contains the variable representing the cost, it is necessary to reset its lower bound to 0 or else no improved solution will be found. Note that we leave the upper bound of the cost variable as it is: this means that the subsequent constructive search goals are constrained to find solutions that are as good as or better than the current solution.

```
_delta.setMin(_costVar, 0);
```

Using the Shuffle Neighborhood

Once the neighborhood is defined, we use it to define a single move using the `IloSingeMove` functionality of IBM ILOG Solver.

```
IloNHood nhood = ShuffleNHood(env, makespan, 0.2);
IloGoal reschedGoal = IloLimitSearch(env,
     IloTextureSuccessorGoal(env) &&
     IloInstantiate(env, makespan),
     IloFailLimit(env, 100));
IloGoal move = IloSingleMove(env,
     lsSolution,
     nhood,
     reschedGoal);
```

The fourth argument of the `IloSingeMove` specifies a goal that is executed at the search state representing the local search neighbor. In our case, we specify a constructive search goal based on texture measurements and using a fail bound of 100. We use a fail bound because we are in a local search framework and so we do not want to fall into a situation where we cannot quickly find a constructive search extension of partial solution generated by the `ShuffleNHood`.

Also note that the probability that we provide to `ShuffleNHood` is 0.2. This means that on average 20% of the edges on the critical path will be preserved from one solution to the next.

Finally, in the same way that the `IloSingleMove` goal is often used in Solver, we enclose the solving algorithm within a loop. In this case, we attempt to find 100 solutions each of which must be as good or better than the preceding solution. In some cases, due to the fail bound, we will not be able to find a solution and so another neighbor will be visited based on the most recently found solution.

```
  IloInt maxIter = 100;
  for(IloInt i = 0; i < maxIter; ++i) {
    lsSolver.out() << “Move: “ << i << “:\t”;
    if (!lsSolver.solve(move))
      lsSolver.out() << “no solution” << endl;
    else {
      IloNum cost = lsSolution.getSolution().getObjectiveValue();
      lsSolver.out() << “solution at cost: “ << cost;
      if (cost < best) {
globalSolution.store(lsScheduler);
best = cost;
lsSolver.out() << “ **”;
      }
      lsSolver.out() << endl;
    }
  }

  env.out() << “Final solution Cost: “ << best << endl;
```

Complete Program

You can see the entire program `jobshopshuff.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>
#include <ilsolver/iimls.h>

ILOSTLBEGIN

IloInt ResourceNumbers06 [] = {2, 0, 1, 3, 5, 4,
                               1, 2, 4, 5, 0, 3,
                               2, 3, 5, 0, 1, 4,
                               1, 0, 2, 3, 4, 5,
                               2, 1, 4, 5, 0, 3,
```

```
                              1, 3, 5, 0, 4, 2};

IloInt Durations06 [] = { 1,  3,  6,  7,  3,  6,
                          8,  5, 10, 10, 10,  4,
                          5,  4,  8,  9,  1,  7,
                          5,  5,  5,  3,  8,  9,
                          9,  3,  5,  4,  3,  1,
                          3,  3,  9, 10,  4,  1};

IloInt ResourceNumbers10 [] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
                               0, 2, 4, 9, 3, 1, 6, 5, 7, 8,
                               1, 0, 3, 2, 8, 5, 7, 6, 9, 4,
                               1, 2, 0, 4, 6, 8, 7, 3, 9, 5,
                               2, 0, 1, 5, 3, 4, 8, 7, 9, 6,
                               2, 1, 5, 3, 8, 9, 0, 6, 4, 7,
                               1, 0, 3, 2, 6, 5, 9, 8, 7, 4,
                               2, 0, 1, 5, 4, 6, 8, 9, 7, 3,
                               0, 1, 3, 5, 2, 9, 6, 7, 4, 8,
                               1, 0, 2, 6, 8, 9, 5, 3, 4, 7};

IloInt Durations10 [] = {29, 78,  9, 36, 49, 11, 62, 56, 44, 21,
                         43, 90, 75, 11, 69, 28, 46, 46, 72, 30,
                         91, 85, 39, 74, 90, 10, 12, 89, 45, 33,
                         81, 95, 71, 99,  9, 52, 85, 98, 22, 43,
                         14,  6, 22, 61, 26, 69, 21, 49, 72, 53,
                         84,  2, 52, 95, 48, 72, 47, 65,  6, 25,
                         46, 37, 61, 13, 32, 21, 32, 89, 30, 55,
                         31, 86, 46, 74, 32, 88, 19, 48, 36, 79,
                         76, 69, 76, 51, 85, 11, 40, 89, 26, 74,
                         85, 13, 61,  7, 64, 76, 47, 52, 90, 45};

IloInt ResourceNumbers20 [] = {0, 1, 2, 3, 4,
                               0, 1, 3, 2, 4,
                               1, 0, 2, 4, 3,
                               1, 0, 4, 2, 3,
                               2, 1, 0, 3, 4,
                               2, 1, 4, 0, 3,
                               1, 0, 2, 3, 4,
                               2, 1, 0, 3, 4,
                               0, 3, 2, 1, 4,
                               1, 2, 0, 3, 4,
                               1, 3, 0, 4, 2,
                               2, 0, 1, 3, 4,
                               0, 2, 1, 3, 4,
                               2, 0, 1, 3, 4,
                               0, 1, 4, 2, 3,
                               1, 0, 3, 4, 2,
                               0, 2, 1, 3, 4,
                               0, 1, 4, 2, 3,
                               1, 2, 0, 3, 4,
                               0, 1, 2, 3, 4};

IloInt Durations20 [] = {29,  9, 49, 62, 44,
                         43, 75, 69, 46, 72,
                         91, 39, 90, 12, 45,
                         81, 71,  9, 85, 22,
                         14, 22, 26, 21, 72,
                         84, 52, 48, 47,  6,
```

```
                          46, 61, 32, 32, 30,
                          31, 46, 32, 19, 36,
                          76, 76, 85, 40, 26,
                          85, 61, 64, 47, 90,
                          78, 36, 11, 56, 21,
                          90, 11, 28, 46, 30,
                          85, 74, 10, 89, 33,
                          95, 99, 52, 98, 43,
                           6, 61, 69, 49, 53,
                           2, 95, 72, 65, 25,
                          37, 13, 21, 89, 55,
                          86, 74, 88, 48, 79,
                          69, 51, 11, 89, 74,
                          13,  7, 76, 52, 45};

////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////

IloModel
DefineModel(IloEnv& env,
            IloInt numberOfJobs,
            IloInt numberOfResources,
            IloInt* resourceNumbers,
            IloInt* durations,
            IloNumVar& makespan,
            IloSchedulerSolution& solution)
{
  IloModel model(env);

  /* CREATE THE MAKESPAN VARIABLE. */
  IloInt numberOfActivities = numberOfJobs * numberOfResources;
  IloInt horizon = 0;
  IloInt k;

  for (k = 0; k < numberOfActivities; k++)
    horizon += durations[k];

  makespan = IloNumVar(env, 0, horizon, ILOINT);
  solution = IloSchedulerSolution(env);

  /* CREATE THE RESOURCES. */
  IloSchedulerEnv schedEnv(env);
  schedEnv.getResourceParam().setCapacityEnforcement(IloMediumHigh);
  schedEnv.getResourceParam().setPrecedenceEnforcement(IloMediumHigh);

  IloTextureParam tParam = schedEnv.getTextureParam();

  IloRCTextureFactory rcFactory = IloRCTextureProbabilisticFactory(env);
  tParam.setRCTextureFactory(rcFactory);

  IloTextureCriticalityCalculator critCalc =
    IloProbabilisticCriticalityCalculator(env);
  tParam.setCriticalityCalculator(critCalc);

  IloRandom randGen(env, 98020981);
```

```
  tParam.setRandomGenerator(randGen);
  tParam.setHeuristicBeta(0.75);

  IloInt j;
  IloUnaryResource *resources =
    new (env) IloUnaryResource[numberOfResources];
  for (j = 0; j < numberOfResources; j++)
    resources[j] = IloUnaryResource(env);

  /* CREATE THE ACTIVITIES. */
  char buffer[128];
  k = 0;
  IloInt i;
  for (i = 0; i < numberOfJobs; i++) {
    IloActivity previousActivity;
    for (j = 0; j < numberOfResources; j++) {
      IloActivity activity(env, durations[k]);
      sprintf(buffer, "J%ldS%ldR%ld", i, j, resourceNumbers[k]);
      activity.setName(buffer);
      model.add(activity.requires(resources[resourceNumbers[k]]));
      if (j != 0)
        model.add(activity.startsAfterEnd(previousActivity));
      solution.add(activity, IloRestoreNothing);
      previousActivity = activity;
      k++;
    }
    model.add(previousActivity.endsBefore(makespan));
  }

  /* RETURN THE MODEL. */
  return model;
}


////////////////////////////////////////////////////////////////////////////
//
// PRINT SOLUTION
//
////////////////////////////////////////////////////////////////////////////
void PrintSolution(IloSchedulerSolution sol){
  IloEnv env = sol.getEnv();
  for(IloSchedulerSolution::ActivityIterator ite(sol); ite.ok(); ++ite){
    IloActivity act = *ite;
    env.out() << act.getName() << "[";

    IloNum startMin = sol.getStartMin(act);
    IloNum startMax = sol.getStartMax(act);
    env.out() << startMin;
    if (startMin != startMax)
      env.out() << ".." << startMax;

    env.out() << " -- ";

    IloNum pMin = sol.getProcessingTimeMin(act);
    IloNum pMax = sol.getProcessingTimeMax(act);
    env.out() << pMin;
    if (pMin != pMax)
      env.out() << ".." << pMax;
```

```
    env.out() << " --> ";

    IloNum endMin = sol.getEndMin(act);
    IloNum endMax = sol.getEndMax(act);
    env.out() << endMin;
    if (endMin != endMax)
      env.out() << ".." << endMax;

    env.out() << "]" << endl;
  }
}

////////////////////////////////////////////////////////////////////////////
//
// Critical Path Object: This object computes and returns a set of
// resource constraints that constitute a randomly selected critical
// path in the current solution
//
////////////////////////////////////////////////////////////////////////////

class RCSortElement {
private:
  IloResourceConstraint _rc;
  IloNum _startValue;
  IloInt _tieBreaker;

public:
  RCSortElement() : _rc(0), _startValue(0), _tieBreaker(0) {}
  RCSortElement(IloResourceConstraint rc, IloInt startValue, IloInt tieBreaker)
    : _rc(rc), _startValue(startValue), _tieBreaker(tieBreaker) {}

  IloResourceConstraint getRC() { return _rc; }
  IloNum getStartValue() { return _startValue; }
  IloInt getTieBreaker() { return _tieBreaker; }

  void setRC(IloResourceConstraint rc) { _rc = rc; }
  void setStartValue(IloNum st) { _startValue = st; }
  void setTieBreaker(IloInt tie) { _tieBreaker = tie; }
};

static
int RCSortElementAscendingSTCompare(const void* first,
                                    const void* second)
{
  RCSortElement *a = (RCSortElement *) first;
  RCSortElement *b = (RCSortElement *) second;

  if (a->getStartValue() > b->getStartValue())
    return 1;
  else if (a->getStartValue() < b->getStartValue())
    return -1;
  else if (a->getTieBreaker() > b->getTieBreaker())
    return 1;
  else if (a->getTieBreaker() < b->getTieBreaker())
    return -1;

  return 0;
```

```
}

class CriticalPath {
private:
  IloInt _cpSize;
  IloArray<IloResourceConstraint> _cpArray;

  IloRandom _randGen;

  RCSortElement *getCriticalRCs(IloSchedulerSolution, IloInt&);
  void pickRandomCP(RCSortElement *, IloSchedulerSolution,IloInt);
  void addCPElement(IloResourceConstraint);

public:
  CriticalPath(IloEnv env) :
    _cpSize(0), _cpArray(env), _randGen(env, 98998) {}
  ~CriticalPath() {}

  IloArray<IloResourceConstraint>& computeCP(IloSchedulerSolution, IloInt&);
};

RCSortElement *CriticalPath::getCriticalRCs(IloSchedulerSolution solution,
                                            IloInt& nbCriticalRCs) {
  // count the number of RCs with assigned start times
  for(IloSchedulerSolution::ResourceConstraintIterator iter(solution);
      iter.ok(); ++iter) {
    IloResourceConstraint rc = *iter;
    IloActivity act = rc.getActivity();
    if (solution.getStartMin(act) == solution.getStartMax(act))
      ++nbCriticalRCs;
  }

  // populate array
  RCSortElement *sortArray = new RCSortElement[nbCriticalRCs];
  IloInt index = 0;
  for(IloSchedulerSolution::ResourceConstraintIterator iter2(solution);
      iter2.ok(); ++iter2) {

    IloResourceConstraint rc = *iter2;
    IloActivity act = rc.getActivity();

    if (solution.getStartMin(act) == solution.getStartMax(act)) {
      sortArray[index].setRC(rc);
      sortArray[index].setStartValue(solution.getStartMin(act));
      sortArray[index].setTieBreaker(index);
      ++index;
    }
  }

  // order in ascending start time
  qsort(sortArray, nbCriticalRCs,
        sizeof(RCSortElement), RCSortElementAscendingSTCompare);

  return sortArray;
}

void CriticalPath::addCPElement(IloResourceConstraint rc) {
  if (_cpSize < _cpArray.getSize())
```

```
    // reuse
    _cpArray[_cpSize] = rc;
  else
    // add
    _cpArray.add(rc);

  _cpSize++;
}

void CriticalPath::pickRandomCP(RCSortElement *sortArray,
                                                IloSchedulerSolution solution,
                                                       IloInt nbCriticalRCs) {
  _cpSize = 0;
  IloNum endRC = 0;
  IloInt i = -1;
  while(i < nbCriticalRCs - 1) {
    // skip elements not on the same critical path as rc
    IloInt nextIndexStart = i + 1;
    while((nextIndexStart < nbCriticalRCs) &&
          (sortArray[nextIndexStart].getStartValue() < endRC))
      nextIndexStart++;

    // gather elements that are successors of rc on the critical
    // path. There may be more than one

    IloInt nextIndexEnd = nextIndexStart + 1;
    while((nextIndexEnd < nbCriticalRCs) &&
          (sortArray[nextIndexEnd].getStartValue() == endRC))
      nextIndexEnd++;

    if (nextIndexStart < nbCriticalRCs) {

      // At this point the elements between sortArray[nextIndexStart]
      // and sortArray[nextIndexEnd-1] inclusive are all successors to
      // rc on some critical path.
      // We randomly pick one of them.

      IloInt randIndex = nextIndexStart;
      IloInt indexDiff = nextIndexEnd - nextIndexStart;
      if (indexDiff > 1)
        randIndex += _randGen.getInt(indexDiff);

      IloResourceConstraint next = sortArray[randIndex].getRC();
      addCPElement(next);

      i = randIndex;
      endRC = solution.getEndMin(next.getActivity());
    }
    else
      i = nbCriticalRCs;
  }
}

IloArray<IloResourceConstraint>& CriticalPath::computeCP(
                                                IloSchedulerSolution solution,
                                                     IloInt& cpSize) {
  IloInt nbCriticalRCs = 0;
  RCSortElement *sortArray = getCriticalRCs(solution, nbCriticalRCs);
```

```
  // Pick the RCs in one (randomly selected) critical path
  pickRandomCP(sortArray, solution, nbCriticalRCs);
  delete [] sortArray;

  cpSize = _cpSize;
  return _cpArray;
}

////////////////////////////////////////////////////////////////////////////
//
// SHUFFLE NEIGHBORHOOD
//
////////////////////////////////////////////////////////////////////////////
class ShuffleNHoodI : public IloNHoodI {
private:
  IloNumVar _costVar;
  IloNum _prob;

  CriticalPath _cp;

  IloSchedulerSolution _delta;
  IloRandom _randGen;

public:
  ShuffleNHoodI(IloEnv env, IloNumVar cost,
                                    IloNum probability, const char *name = 0)
    : IloNHoodI (env, name),
      _costVar(cost), _prob(probability),
      _cp(env), _delta(0), _randGen(env, 98998) {}
  ~ShuffleNHoodI() {}

  void        start       (IloSolver solver, IloSolution soln);

  IloSolution define      (IloSolver solver, IloInt) {
    assert(_delta.getImpl() != 0);
    assert(_delta.getSolution().getImpl() != 0);
    return _delta.getSolution().makeClone(solver.getEnv());
  }

  IloInt      getSize     (IloSolver) const { return 1; }
};

void ShuffleNHoodI::start(IloSolver solver, IloSolution soln) {
  IloSchedulerSolution solution(soln);

  IloEnv env = solver.getEnv();

  IloInt cpSize = 0;
  IloArray<IloResourceConstraint> cpArray = _cp.computeCP(solution, cpSize);

  // Create the move
  if (_delta.getImpl() != 0)
    _delta.end();
  _delta = solution.makeClone(env);

  // For each edge in the critical path (between resource constraints
  // on the same resource) keep it in the solution with probability
```

```
  // (1 - _prob)
  for(IloInt i = 0; i < cpSize - 1; ++i) {
    IloResourceConstraint before = cpArray[i];
    IloResourceConstraint after = cpArray[i + 1];

    if (solution.hasNextRC(before) &&
                  (after.getImpl() == solution.getNextRC(before).getImpl()) &&
                                              (_randGen.getFloat() > _prob)) {
      // Remove resource constraint from delta. This has the effect of
      // ensuring that this edge will be in the next solution
      _delta.remove(before);
    }
  }

  // For everything remaining in the solution, get rid of the nexts.
  // This has the effect of leaving the next of each rc undefined and
  // so this defines the search space for the rescheduling goal.
  for(IloSchedulerSolution::ResourceConstraintIterator iter3(_delta);
      iter3.ok(); ++iter3) {
    IloResourceConstraint rc = *iter3;
    if (_delta.hasNextRC(rc))
      _delta.unsetNext(rc, _delta.getNextRC(rc));
  }

  // Reset the lower bound of the costVar

  _delta.setMin(_costVar, 0);
}


IloNHood ShuffleNHood(IloEnv env, IloNumVar cost,
                                            IloNum prob, const char *name = 0) {
  return new (env) ShuffleNHoodI(env, cost, prob, name);
}


////////////////////////////////////////////////////////////////////////////
//
// PROBLEM SOLVING
//
////////////////////////////////////////////////////////////////////////////

void FindInitialSolution(IloModel model,
                                            IloSchedulerSolution globalSolution,
                                                IloSchedulerSolution lsSolution,
                                                            IloNumVar costVar) {
  IloEnv env = model.getEnv();
  IloSolver solver(model);
  IlcScheduler scheduler(solver);

  IloGoal g = IloRankForward(env) && IloInstantiate(env, costVar);
  solver.startNewSearch(g);

  if(solver.next()) {
    IlcInt best = (IlcInt)solver.getValue(costVar);
    solver.out() << “Solution for Cost “ <<  best << endl;
    solver.printInformation();
    globalSolution.store(scheduler);
```

```
    lsSolution.store(scheduler);
  }

  solver.endSearch();
  solver.end();
}

void
SolveModel(IloModel model,
                                                    IloNumVar makespan,
                                          IloSchedulerSolution& globalSolution)
{
  IloEnv env = model.getEnv();
  IloObjective obj = IloMinimize(env, makespan);
  globalSolution.getSolution().add(makespan);

  /* CREATE LOCAL SEARCH SOLUTION */
  IloSchedulerSolution lsSolution = globalSolution.makeClone(env);

  for (IloIterator <IloResourceConstraint> iter(env); iter.ok(); ++iter)
    lsSolution.add(*iter, IloRestoreRCNext);

  lsSolution.getSolution().add(obj);

  /* GENERATE AN INITIAL SOLUTION. */
  FindInitialSolution(model, globalSolution, lsSolution, makespan);
  IloNum best = globalSolution.getMax(makespan);
  env.out() << "Initial solution:" << endl;
  PrintSolution(globalSolution);

  /* LOCAL SEARCH */
  IloSolver lsSolver(model);
  IlcScheduler lsScheduler(lsSolver);

  IloNHood nhood = ShuffleNHood(env, makespan, 0.2);
  IloGoal reschedGoal = IloLimitSearch(env,
                                          IloTextureSuccessorGoal(env) &&
                                            IloInstantiate(env, makespan),
                                                  IloFailLimit(env, 100));
  IloGoal move = IloSingleMove(env,
                                                        lsSolution,
                                                             nhood,
                                                       reschedGoal);

  IloInt maxIter = 100;
  for(IloInt i = 0; i < maxIter; ++i) {
    lsSolver.out() << "Move: " << i << ":\t";
    if (!lsSolver.solve(move))
      lsSolver.out() << "no solution" << endl;
    else {
      IloNum cost = lsSolution.getSolution().getObjectiveValue();
      lsSolver.out() << "solution at cost: " << cost;
      if (cost < best) {
                                          globalSolution.store(lsScheduler);
                                                       best = cost;
                                                lsSolver.out() << " **";
      }
      lsSolver.out() << endl;
```

```
    }
  }

  env.out() << "Final solution Cost: " << best << endl;

  PrintSolution(globalSolution);
  lsSolver.end();
}

////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////

void
InitParameters(int argc,
               char** argv,
               IloInt& numberOfJobs,
               IloInt& numberOfResources,
               IloInt*& resourceNumbers,
               IloInt*& durations)
{
  if (argc > 1) {
    IloInt number = atol(argv[1]);
    if (number == 10) {
      numberOfJobs = 10;
      numberOfResources = 10;
      resourceNumbers = ResourceNumbers10;
      durations = Durations10;
    }
    else if (number == 20) {
      numberOfJobs = 20;
      numberOfResources = 5;
      resourceNumbers = ResourceNumbers20;
      durations = Durations20;
    }
  }
}

int main(int argc, char** argv)
{
  try {
    IloEnv env;

    IloInt numberOfJobs = 6;
    IloInt numberOfResources = 6;
    IloInt* resourceNumbers = ResourceNumbers06;
    IloInt* durations = Durations06;
    InitParameters(argc,
                   argv,
                   numberOfJobs,
                   numberOfResources,
                   resourceNumbers,
                   durations);
    IloNumVar makespan;
    IloSchedulerSolution solution;
    IloModel model = DefineModel(env,
```

```
                                numberOfJobs,
                                numberOfResources,
                                resourceNumbers,
                                durations,
                                makespan,
                                solution);

    SolveModel(model,
                                                                  makespan,
                                                                 solution);

    env.end();
  }
  catch (IloException& exc) {
    cout << exc << endl;
  }

  return 0;
}
```

C H A P T E R

27

Tabu Search for the Jobshop Problem

The local search implementation of the shuffle technique in the previous chapter is a very simple use of the local search infrastructure provided by IBM© ILOG© Solver: it only requires the implementation of a new neighborhood class and the search is done by a constructive search goal.

Often, a key component of a local search technique is a metaheuristic. A metaheuristic is an additional object in the local search protocol that acts as a filter on one or more neighborhoods. A classic example of a metaheuristic is a tabu list, which stores information about the recently made moves and prohibits their reversal. Such a technique helps to avoid cycling in local search so that the same set of solutions are not repeatedly visited.

In this chapter, we implement a simple tabu search metaheuristic on top of a standard neighborhood from the jobshop scheduling literature.

Problem Description

As in the previous chapter, the problem used here is the standard job shop scheduling problem as defined in Chapter 15, *Scheduling with Unary Resources: the Job-Shop Problem*. As we have seen the way in which the problem is modeled in previous chapters, we will move directly to the search mechanism.

Solving the Problem

This section describes how the tabu search is incorporated into solving the problem.

The Critical Path

This example builds on top of the shuffle example implemented in the previous chapter. In particular, it reuses the `CriticalPath` object. See Calculating the Critical Path.

Calculating the Previous Resource Constraints

In addition to the critical path calculation, another requirement of the neighborhood implemented in this example is to be able to access both the next and previous resource constraints for each resource constraint in a solution. The next resource constraints are directly stored in an `IloSchedulerSolution`. There is, however, no direct representation of the previous resource constraints. Therefore, we need a function that, given a solution, calculates the previous resource constraints. Since we have the next resource constraints in the solution, this is easy to do.

```
void CalculatePrevs(IloSchedulerSolution soln) {
  // reset prev pointers
  for(IloSchedulerSolution::ResourceIterator resIter(soln);
      resIter.ok(); ++resIter) {
    assert(soln.hasSetupRC(*resIter));
    IloResourceConstraint rc = soln.getSetupRC(*resIter);

    // setup has no prev
    rc.setObject(0);

    while(soln.hasNextRC(rc)) {
      IloResourceConstraint next = soln.getNextRC(rc);
      next.setObject(rc.getImpl());
      rc = next;
    }
  }
}
```

Creating an IloSchedulerSolution

As with any local search in Scheduler implementation, it is necessary to construct an `IloSchedulerSolution` object that will be the central data structure in the local search protocol.

In this case, we create the `IloSchedulerSolution` as we did in the previous chapter with one exception. In the code above that calculates and stores the previous resource constraints, the main loop iterates over the resources that are stored in the solution. It is necessary, therefore, to actually add the resources to the solution.

In summary, we create the local search solution by cloning the existing solution, adding the resource constraints and specifying that they are to restore their next resource constraints, and by adding the resources.

```
IloSchedulerSolution CreateLSSolution(IloEnv env,
                                      IloSchedulerSolution globalSolution) {


  /* CREATE LOCAL SEARCH SOLUTION */
  IloSchedulerSolution lsSolution = globalSolution.makeClone(env);

  for (IloIterator <IloResourceConstraint> iter(env); iter.ok(); ++iter)
    lsSolution.add(*iter, IloRestoreRCNext);

  for (IloIterator <IloResource> resIter(env); resIter.ok(); ++resIter)
    lsSolution.add(*resIter);

  return lsSolution;
}
```

Finding the First Solution

The first solution is found with precisely the same algorithm as was used to find the first solution in the previous chapter.

Creating the SwapRC Move

As described below, the N1 neighborhood consists of the swapping of all pairs of adjacent activities that are both on the same resource and on the same randomly selected critical path. In order to represent a single swap, we use the class `SwapRC`.

This class has two `IloResourceConstraint` data members, `_rc` and `_after`, which represent the original ordering before the swap takes place. That is, in the solution from which an instance of the `SwapRC` class is created, `after` is the next resource constraint of `_rc`. When a move has been performed, this order is reversed.

Aside from the standard accessor functions in the `SwapRC` class, there are three important methods: `isReverseMove`, `createDelta`, and `findSwapFromDelta`. Each of these functions are called by the N1 neighborhood class that we define below. Before describing where these methods are called, we describe each of them in turn.

The `isReverseMove` function accepts an `IloSolution` object that represents a move (a "delta" solution) and returns `IloTrue` if the delta represents a move that is the opposite of the current `SwapRC` instance. For example, given a `SwapRC` instance with `_rc` equal to resource constraint A and `_after` equal to resource constraint B, the move that it represents is a swap from A $\rightarrow$ B to B $\rightarrow$ A. Therefore, if the `isReverseMove` method of this instance is passed a delta where `delta.getNextRC(A)` is equal to B, the delta represents a move

that reinserts the A → B relation. This is the reverse move and so the `isReverseMove` methods returns `IloTrue`.

```
IloBool isReverseMove(IloSolution delta) const {
  IloSchedulerSolution schedDelta(delta);
  return (schedDelta.contains(_rc) &&
          schedDelta.getNextRC(_rc).getImpl() == _after.getImpl());
}
```

The second important method is `createDelta`. This function returns the `IloSchedulerSolution` object that represents the change to the current solution corresponding to the current `SwapRC` instance. The difference is simply to remove the next relation from `_rc` to `_after` and to insert the next relation from `_after` to `_rc`. The code of this function is more complex because other objects in the solution must be updated to be coherent with this swap. For example, if, in the current solution, resource constraint C is previous to `_rc`, we must update the next pointer of C to point to `_after`. Otherwise, we will be in a situation where `_rc` is next to both `_after` and C, obviously an infeasible state on a unary resource.

The steps in `createDelta` are the following. First, `_rc` and `_after` are added to the delta and their current next values are unset.

```
swapDelta.add(_rc, current.getRestorable(_rc));
swapDelta.getSolution().copy(_rc, current);
swapDelta.unsetNext(_rc, current.getNextRC(_rc));

swapDelta.add(_after, current.getRestorable(_after));
swapDelta.getSolution().copy(_after, current);
if (current.hasNextRC(_after))
  swapDelta.unsetNext(_after, current.getNextRC(_after));
```

Then, if there is a resource constraint previous to `_rc` (like C in the example above), it is added to the delta and its next pointer is changed to be equal to `_after`.

```
//   IloResourceConstraint before = getBefore();
  IloResourceConstraint before = current.getPrevRC(_rc);
  if (before.getImpl() != 0) {
    swapDelta.add(before, current.getRestorable(before));
    swapDelta.getSolution().copy(before, current);
    swapDelta.unsetNext(before, current.getNextRC(before));
    swapDelta.setNext(before, _after);
  }
```

Finally, the next of `_after` is set to `_rc`, and the next of `_rc` is set to what used to be the next of `_after`.

```
swapDelta.setNext(_after, _rc);

// change _rc --> _after to _rc --> afterAfter
if (current.hasNextRC(_after))
  swapDelta.setNext(_rc, current.getNextRC(_after));
```

Figure 27.1 shows the next pointers for each resource constraint before and after the move is done. Note that all three pointers have changed, for example, the next of `before` is `_rc` before the move and is `_after` after the move.

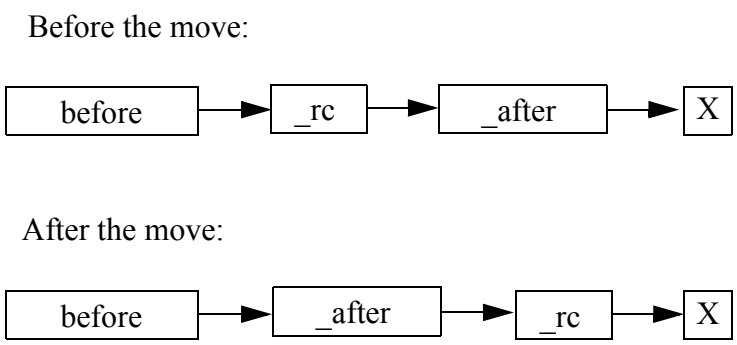


Figure 27.1 *Next Pointers Before and After the Move*

The resource constraints that do not change are not added to the delta. This ensures that they will be restored to their previous values by the local search protocol.

The final `SwapRC` method is `findSwapFromDelta`. This method, takes as input a delta solution and returns a corresponding `SwapRC` instance. This functionality is the inverse of `createDelta`: where `createDelta` returns a delta solution corresponding to a `SwapRC` instance, `findSwapFromDelta` returns a `SwapRC` instance corresponding to a delta solution.

Creating the N1 Neighborhood

The N1 neighborhood is a commonly used neighborhood from the Operations Research literature for local search in job shop scheduling. The neighborhood consists of the swapping of all pairs of adjacent activities that are both on the same resource and on the same randomly selected critical path. As noted above, we make use of the `CriticalPath` object introduced in the previous chapter to select a critical path.

To construct the N1NHoodI, we subclass from the IBM ILOG Solver class IloNHoodI.

```
class N1NHoodI : public IloNHoodI {
private:
  IloSchedulerSolution _solution;

  CriticalPath *_cp;

  IloArray<SwapRC*> _moves;
  IloInt _nbMoves;

  void addMove(IloSolver solver,
               IloResourceConstraint rc, IloResourceConstraint after);

public:
  N1NHoodI(IloEnv env, CriticalPath *cp, const char *name = 0 ) :
    IloNHoodI (env, name), _cp(cp), _moves(env), _nbMoves(0) {}
  ~N1NHoodI() {}

  void start(IloSolver solver, IloSolution soln);

  IloSolution define(IloSolver solver, IloInt i) {
    return (_moves[i])->createDelta(solver.getEnv(), _solution);
  }

  IloInt getSize(IloSolver) const { return _nbMoves; }

  void reset() { _solution = 0; _nbMoves = 0; }
};
```

The data members of N1NHoodI are the current solution (_solution), a pointer to the CriticalPath object (_cp), an array of pointers to SwapRC instances (_moves), and the number of elements currently in that array (_nbMoves).

Given the SwapRC class discussed above, three of the four neighborhood functions are very simple.

- define simply calls createDelta on an element in the _moves array
- getSize returns _nbMoves
- reset resets the _solution and _nbMoves to 0.

The main computation of the neighborhood is performed in the start method where, after some initialization, the previous pointers are set and the critical path is calculated by the CalculatePrevs function and CriticalPath class discussed above.

```
  IloInt cpSize = 0;
  IloArray<IloResourceConstraint> cpArray = _cp->computeCP(_solution, cpSize);
```

Then the moves are created. Consecutive elements on the critical path are examined and if they share a next relation, a move is created.

```
for(IloInt i = 0; i < cpSize - 1; ++i) {
  IloResourceConstraint rc = cpArray[i];
  IloResourceConstraint after = cpArray[i + 1];

  if ((_solution.getNextRC(rc).getImpl() != 0) &&
      (after.getImpl() == _solution.getNextRC(rc).getImpl())) {

    addMove(solver, rc, after);
  }
}
```

The Tabu Search Metaheuristic

The final component of this example is the metaheuristic that implements the tabu search. We implement a very simple tabu search where the `MyTabuI` class maintains a circular list of the last 10 moves that have been performed. A possible move suggested by the neighborhood is disallowed if it is the reverse of a move that is on the tabu list. The only exception to this rule is if the move results in a solution with a cost that is lower than the best solution that has been found so far.

Five of the virtual functions from the `IloMetaHeuristicI` class are redefined in our implementation. See the *IBM ILOG Solver Reference Manual* and *IBM ILOG Solver User's Manual* for a detailed description of the purpose of each of these methods.

Two of the redefined methods (`reset` and `start`) are very simple: the former resets the `_bestCost` to `IloInfinity` and empties the tabu list while the latter simply records the cost of the current solution if it is lower than any solution encountered so far.

We look more deeply at each of the other methods.

First, the `test` method is called by the local search protocol to determine if a move suggested by the neighborhood should be disallowed. If so, the function returns `IloFalse` and otherwise (that is, if the move is acceptable), `IloTrue` is returned. Our implementation first checks to see if the solution has a cost lower than any solution seen so far and, if so,

accepts the move. Otherwise the tabu list is checked to see if the delta solution is the reverse of a move that is on the tabu list. If so, the move is rejected. Otherwise the move is accepted.

```
IloBool MyTabuI::test(IloSolver solver, IloSolution delta) {
  if (solver.getMin(_costVar) < _bestCost)
    return IloTrue;

  for(IloInt i = 0; i < _nbTabuEles; ++i) {
    if ((_tabuList[i] != 0) && _tabuList[i]->isReverseMove(delta))
      return IloFalse;
  }

  return IloTrue;
}
```

Second, the `notify` method is called whenever the local search protocol accepts a new move. What we need to do therefore is to add the new move to our tabu list. To do this, we make use of the `findSwapFromDelta` method described above and then add the move to the circular tabu list.

```
void MyTabuI::notify(IloSolver solver, IloSolution delta) {
  SwapRC *move = SwapRC::findSwapFromDelta(solver, delta);
  assert(move != 0);

  _tabuList[_nextTabuEle] = move;

  _nextTabuEle++;
  _nextTabuEle %= _maxTabuSize;

  if (_nbTabuEles < _maxTabuSize)
    _nbTabuEles++;
}
```

The final metaheuristic function is `complete`. This function is called by the local search in the situation where all possible moves are either infeasible or rejected by the `test` method. If this method returns `IloTrue`, the local search is ended. In our case, we simply age the tabu list by replacing the oldest element in the list by the youngest. This will have the effect

of allowing one more possible neighbor than was allowed in the previous state. If all the elements in the tabu list are equal to the youngest element, we return `IloTrue`.

```
IloBool MyTabuI::complete() {
  // Remove the element that has been in the list the longest
  // time. Insert the youngest element in its place.
  IloInt youngest = _nextTabuEle - 1;
  if (youngest < 0)
    youngest = _nbTabuEles - 1;

  if ((youngest < 0) ||
      (_tabuList[youngest] == _tabuList[_nextTabuEle]))
    return IloTrue;

  _tabuList[_nextTabuEle] = _tabuList[youngest];

  _nextTabuEle++;
  _nextTabuEle %= _maxTabuSize;

  return IloFalse;
}
```

Scheduler Parameters for Local Search

All the moves in each neighborhood define a complete sequence of resource constraints on each resource. By definition, this must satisfy the unary capacity resource constraints. Furthermore, since each move is completely defined by the neighborhood object, there is no benefit from extra constraint propagation. We can therefore save time by ignoring the resource capacity constraints without losing soundness.

```
  IloSchedulerEnv schedEnv(env);
  schedEnv.getResourceParam().ignoreCapacityConstraints();
```

Hill-climbing with the N1 Neighborhood

It is often the case at the beginning of a local search that improvements to the current solution can be made simply by greedy hill-climbing without using a metaheuristic. We define a local search goal move (using `IloSingleMove` as in the previous chapter), that scans the neighbors until it finds one that has a lower cost than the current solution. It immediately accepts that solution.

Because we are only scanning the neighbors until we find an improvement and because we have no metaheuristic, performing a single move during the hill-climbing phase is typically

much faster that during the tabu phase. For move information about the `IloSingleMove` goal, see the *IBM ILOG Solver User's Manual* and *IBM ILOG Solver Reference Manual*.

```
CriticalPath cp(env);
IloNHood nhood = N1NHood(env, &cp);
IloGoal greedyMove = IloSingleMove(env,
                                   lsSolution,
                                   nhood,
                                   IloImprove(env),
                                   IloFirstSolution(env),
                                   IloInstantiate(env, makespan));

IloInt maxIter = 100;
IloInt movesDone = 0;
while((movesDone < maxIter) && lsSolver.solve(greedyMove)) {
  IloNum cost = lsSolution.getSolution().getObjectiveValue();
  lsSolver.out() << "Move: " << movesDone << ":\t";
  ++movesDone;
  lsSolver.out() << "solution at cost: " << cost
                 << " ** HC\n";
  best = cost;
  globalSolution.store(lsScheduler);
}
```

Tabu Search with the N1 Neighborhood

The hill-climbing phase continues until we are in a state such that all neighbors have a greater cost than that of the current solution. At that point, we transition to the tabu search phase. Using the same neighborhood, we simply create another `IloSingleMove` goal, this time including an instance of the `MyTabu` metaheuristic, described above.

```
IloMetaHeuristic mh = MyTabu(env, makespan);
IloGoal move = IloSingleMove(env,
                             lsSolution,
                             nhood,
                             mh,
                             IloMinimizeVar(env, makespan),
                             IloInstantiate(env, makespan));

for(IloInt i = movesDone; i < maxIter; ++i) {
  lsSolver.out() << "Move: " << i << ":\t";
  if (!lsSolver.solve(move)) {
    lsSolver.out() << "no solution" << endl;
    if ((nhood.getSize(lsSolver) == 0) || mh.complete())
      break;
  }
  else {
    IloNum cost = lsSolution.getSolution().getObjectiveValue();
    lsSolver.out() << "solution at cost: " << cost;
    if (cost < best) {
      globalSolution.store(lsScheduler);
      best = cost;
      lsSolver.out() << " **";
    }
    lsSolver.out() << endl;
```

```
    }
  }
```

Whenever there are no feasible, non-tabu neighbors, we call the `MyTabu::complete` method. We arbitrarily choose to search until the total number of moves (or non-solutions) we have found in the hill-climbing and tabu search phase together total 100.

Complete Program

You can see the entire program `jobshopn1.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>
#include <ilsolver/iimls.h>

ILOSTLBEGIN

IloInt ResourceNumbers06 [] = {2, 0, 1, 3, 5, 4,
                               1, 2, 4, 5, 0, 3,
                               2, 3, 5, 0, 1, 4,
                               1, 0, 2, 3, 4, 5,
                               2, 1, 4, 5, 0, 3,
                               1, 3, 5, 0, 4, 2};

IloInt Durations06 [] = { 1,  3,  6,  7,  3,  6,
                          8,  5, 10, 10, 10,  4,
                          5,  4,  8,  9,  1,  7,
                          5,  5,  5,  3,  8,  9,
                          9,  3,  5,  4,  3,  1,
                          3,  3,  9, 10,  4,  1};

IloInt ResourceNumbers10 [] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
                               0, 2, 4, 9, 3, 1, 6, 5, 7, 8,
                               1, 0, 3, 2, 8, 5, 7, 6, 9, 4,
                               1, 2, 0, 4, 6, 8, 7, 3, 9, 5,
                               2, 0, 1, 5, 3, 4, 8, 7, 9, 6,
                               2, 1, 5, 3, 8, 9, 0, 6, 4, 7,
                               1, 0, 3, 2, 6, 5, 9, 8, 7, 4,
                               2, 0, 1, 5, 4, 6, 8, 9, 7, 3,
                               0, 1, 3, 5, 2, 9, 6, 7, 4, 8,
                               1, 0, 2, 6, 8, 9, 5, 3, 4, 7};

IloInt Durations10 [] = {29, 78,  9, 36, 49, 11, 62, 56, 44, 21,
                         43, 90, 75, 11, 69, 28, 46, 46, 72, 30,
                         91, 85, 39, 74, 90, 10, 12, 89, 45, 33,
                         81, 95, 71, 99,  9, 52, 85, 98, 22, 43,
                         14,  6, 22, 61, 26, 69, 21, 49, 72, 53,
                         84,  2, 52, 95, 48, 72, 47, 65,  6, 25,
                         46, 37, 61, 13, 32, 21, 32, 89, 30, 55,
                         31, 86, 46, 74, 32, 88, 19, 48, 36, 79,
                         76, 69, 76, 51, 85, 11, 40, 89, 26, 74,
```

```
                          85, 13, 61,  7, 64, 76, 47, 52, 90, 45};

IloInt ResourceNumbers20 [] = {0, 1, 2, 3, 4,
                               0, 1, 3, 2, 4,
                               1, 0, 2, 4, 3,
                               1, 0, 4, 2, 3,
                               2, 1, 0, 3, 4,
                               2, 1, 4, 0, 3,
                               1, 0, 2, 3, 4,
                               2, 1, 0, 3, 4,
                               0, 3, 2, 1, 4,
                               1, 2, 0, 3, 4,
                               1, 3, 0, 4, 2,
                               2, 0, 1, 3, 4,
                               0, 2, 1, 3, 4,
                               2, 0, 1, 3, 4,
                               0, 1, 4, 2, 3,
                               1, 0, 3, 4, 2,
                               0, 2, 1, 3, 4,
                               0, 1, 4, 2, 3,
                               1, 2, 0, 3, 4,
                               0, 1, 2, 3, 4};

IloInt Durations20 [] = {29,  9, 49, 62, 44,
                         43, 75, 69, 46, 72,
                         91, 39, 90, 12, 45,
                         81, 71,  9, 85, 22,
                         14, 22, 26, 21, 72,
                         84, 52, 48, 47,  6,
                         46, 61, 32, 32, 30,
                         31, 46, 32, 19, 36,
                         76, 76, 85, 40, 26,
                         85, 61, 64, 47, 90,
                         78, 36, 11, 56, 21,
                         90, 11, 28, 46, 30,
                         85, 74, 10, 89, 33,
                         95, 99, 52, 98, 43,
                          6, 61, 69, 49, 53,
                          2, 95, 72, 65, 25,
                         37, 13, 21, 89, 55,
                         86, 74, 88, 48, 79,
                         69, 51, 11, 89, 74,
                         13,  7, 76, 52, 45};

////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////

IloModel
DefineModel(IloEnv& env,
            IloInt numberOfJobs,
            IloInt numberOfResources,
            IloInt* resourceNumbers,
            IloInt* durations,
            IloNumVar& makespan,
            IloSchedulerSolution& solution)
```

```
{
  IloModel model(env);

  /* CREATE THE MAKESPAN VARIABLE. */
  IloInt numberOfActivities = numberOfJobs * numberOfResources;
  IloInt horizon = 0;
  IloInt k;

  for (k = 0; k < numberOfActivities; k++)
    horizon += durations[k];

  makespan = IloNumVar(env, 0, horizon, ILOINT);
  solution = IloSchedulerSolution(env);

  /* CREATE THE RESOURCES. */
  IloSchedulerEnv schedEnv(env);
  schedEnv.getResourceParam().setPrecedenceEnforcement(IloMediumHigh);

  IloInt j;
  IloUnaryResource *resources =
    new (env) IloUnaryResource[numberOfResources];
  for (j = 0; j < numberOfResources; j++)
    resources[j] = IloUnaryResource(env);

  /* CREATE THE ACTIVITIES. */
  char buffer[128];
  k = 0;
  IloInt i;
  for (i = 0; i < numberOfJobs; i++) {
    IloActivity previousActivity;
    for (j = 0; j < numberOfResources; j++) {
      IloActivity activity(env, durations[k]);
      sprintf(buffer, "J%ldS%ldR%ld", i, j, resourceNumbers[k]);
      activity.setName(buffer);
      model.add(activity.requires(resources[resourceNumbers[k]]));
      if (j != 0)
        model.add(activity.startsAfterEnd(previousActivity));
      solution.add(activity, IloRestoreNothing);
      previousActivity = activity;
      k++;
    }
    model.add(previousActivity.endsBefore(makespan));
  }

  /* RETURN THE MODEL. */
  return model;
}


////////////////////////////////////////////////////////////////////////////
//
// PRINT SOLUTION
//
////////////////////////////////////////////////////////////////////////////
void PrintSolution(IloSchedulerSolution sol){
  IloEnv env = sol.getEnv();
  for(IloSchedulerSolution::ActivityIterator ite(sol); ite.ok(); ++ite){
    IloActivity act = *ite;
```

```
    env.out() << act.getName() << "[";

    IloNum startMin = sol.getStartMin(act);
    IloNum startMax = sol.getStartMax(act);
    env.out() << startMin;
    if (startMin != startMax)
      env.out() << ".." << startMax;

    env.out() << " -- ";

    IloNum pMin = sol.getProcessingTimeMin(act);
    IloNum pMax = sol.getProcessingTimeMax(act);
    env.out() << pMin;
    if (pMin != pMax)
      env.out() << ".." << pMax;

    env.out() << " --> ";

    IloNum endMin = sol.getEndMin(act);
    IloNum endMax = sol.getEndMax(act);
    env.out() << endMin;
    if (endMin != endMax)
      env.out() << ".." << endMax;

    env.out() << "]" << endl;
  }
}

////////////////////////////////////////////////////////////////////////////
//
// MOVE
//
////////////////////////////////////////////////////////////////////////////

class SwapRC {
  // This class represents the data to perform a swap of _rc and
  // _after in the sequence
private:
  IloResourceConstraint _rc;
  IloResourceConstraint _after;

public:
  SwapRC(IloResourceConstraint rc, IloResourceConstraint after)
    : _rc(rc), _after(after) {}
  ~SwapRC() {}

  IloResourceConstraint getRC() const { return _rc; }
  IloResourceConstraint getAfter() const { return _after; }

  void setRC(IloResourceConstraint rc) { _rc = rc; }
  void setAfter(IloResourceConstraint after) { _after = after; }

  IloBool isReverseMove(IloSolution delta) const {
    IloSchedulerSolution schedDelta(delta);
    return (schedDelta.contains(_rc) &&
            schedDelta.getNextRC(_rc).getImpl() == _after.getImpl());
  }
```

```
  IloSolution createDelta(IloEnv, IloSchedulerSolution);

  static SwapRC* findSwapFromDelta(IloSolver, IloSolution);
};

SwapRC* SwapRC::findSwapFromDelta(IloSolver solver,
                                  IloSolution delta) {
  // Try to generate a SwapRC move from the information in delta.
  IloSchedulerSolution solution(delta);

  // find the element that is not next of any element in the delta
  IloResourceConstraint first;
  for(IloSchedulerSolution::ResourceConstraintIterator iter(solution);
      iter.ok() && (0 == first.getImpl()); ++iter) {
    first = *iter;
    for(IloSchedulerSolution::ResourceConstraintIterator iter2(solution);
        iter2.ok() && (0 != first.getImpl()); ++iter2) {
      IloResourceConstraint rc = *iter2;
      if (rc.getImpl() != first.getImpl()) {
        if (solution.hasNextRC(rc) &&
            (solution.getNextRC(rc).getImpl() == first.getImpl()))
          first = 0;
      }
    }
  }

  assert(0 != first.getImpl());

  IloResourceConstraint second = solution.getNextRC(first);
  IloResourceConstraint third = solution.getNextRC(second);
  if ((0 == third.getImpl()) || !solution.contains(third))
    return new (solver.getEnv()) SwapRC(second, first);
  else
    return new (solver.getEnv()) SwapRC(third, second);
}

IloSolution SwapRC::createDelta(IloEnv env, IloSchedulerSolution current) {

  IloSchedulerSolution swapDelta(env);

  // old order: before -> _rc -> _after -> afterAfter
  // new order: before -> _after -> _rc -> afterAfter

  swapDelta.add(_rc, current.getRestorable(_rc));
  swapDelta.getSolution().copy(_rc, current);
  swapDelta.unsetNext(_rc, current.getNextRC(_rc));

  swapDelta.add(_after, current.getRestorable(_after));
  swapDelta.getSolution().copy(_after, current);
  if (current.hasNextRC(_after))
    swapDelta.unsetNext(_after, current.getNextRC(_after));

  // change beforeBefore --> _rc to beforeBefore --> _after
//   IloResourceConstraint before = getBefore();
  IloResourceConstraint before = current.getPrevRC(_rc);
  if (before.getImpl() != 0) {
    swapDelta.add(before, current.getRestorable(before));
    swapDelta.getSolution().copy(before, current);
```

```
    swapDelta.unsetNext(before, current.getNextRC(before));
    swapDelta.setNext(before, _after);
  }

  // change _after --> afterAfter to _after --> _rc
  swapDelta.setNext(_after, _rc);

  // change _rc --> _after to _rc --> afterAfter
  if (current.hasNextRC(_after))
    swapDelta.setNext(_rc, current.getNextRC(_after));

  return swapDelta;
}

////////////////////////////////////////////////////////////////////////////
//
// Critical Path Object: This object computes and returns a set of
// resource constraints that constitute a randomly selected critical
// path in the current solution
//
////////////////////////////////////////////////////////////////////////////

class RCSortElement {
private:
  IloResourceConstraint _rc;
  IloInt _startValue;
  IloInt _tieBreaker;

public:
  RCSortElement() : _rc(0), _startValue(0), _tieBreaker(0) {}
  RCSortElement(IloResourceConstraint rc, IloInt startValue, IloInt tieBreaker)
    : _rc(rc), _startValue(startValue), _tieBreaker(tieBreaker) {}

  IloResourceConstraint getRC() { return _rc; }
  IloInt getStartValue() { return _startValue; }
  IloInt getTieBreaker() { return _tieBreaker; }

  void setRC(IloResourceConstraint rc) { _rc = rc; }
  void setStartValue(IloInt st) { _startValue = st; }
  void setTieBreaker(IloInt tie) { _tieBreaker = tie; }
};

static
int RCSortElementAscendingSTCompare(const void* first,
                                    const void* second)
{
  RCSortElement *a = (RCSortElement *) first;
  RCSortElement *b = (RCSortElement *) second;

  if (a->getStartValue() > b->getStartValue())
    return 1;
  else if (a->getStartValue() < b->getStartValue())
    return -1;
  else if (a->getTieBreaker() > b->getTieBreaker())
    return 1;
  else if (a->getTieBreaker() < b->getTieBreaker())
    return -1;
```

```
  return 0;
}

class CriticalPath {
private:
  IloInt _cpSize;
  IloArray<IloResourceConstraint> _cpArray;

  IloRandom _randGen;

  RCSortElement *getCriticalRCs(IloSchedulerSolution, IloInt&);
  void pickRandomCP(RCSortElement *, IloSchedulerSolution,IloInt);
  void addCPElement(IloResourceConstraint);

public:
  CriticalPath(IloEnv env) :
    _cpSize(0), _cpArray(env), _randGen(env, 98998) {}
  ~CriticalPath() {}

  IloArray<IloResourceConstraint>& computeCP(IloSchedulerSolution, IloInt&);
};

RCSortElement *CriticalPath::getCriticalRCs(IloSchedulerSolution solution,
                                            IloInt& nbCriticalRCs) {
  // count the number of RCs with assigned start times
  for(IloSchedulerSolution::ResourceConstraintIterator iter(solution);
      iter.ok(); ++iter) {
    IloResourceConstraint rc = *iter;
    IloActivity act = rc.getActivity();
    if (solution.getStartMin(act) == solution.getStartMax(act))
      ++nbCriticalRCs;
  }

  // populate array
  RCSortElement *sortArray = new RCSortElement[nbCriticalRCs];
  IloInt index = 0;
  for(IloSchedulerSolution::ResourceConstraintIterator iter2(solution);
      iter2.ok(); ++iter2) {

    IloResourceConstraint rc = *iter2;
    IloActivity act = rc.getActivity();

    if (solution.getStartMin(act) == solution.getStartMax(act)) {
      sortArray[index].setRC(rc);
      sortArray[index].setStartValue( (IloInt)solution.getStartMin(act) );
      sortArray[index].setTieBreaker(index);
      ++index;
    }
  }

  // order in ascending start time
  qsort(sortArray, nbCriticalRCs,
        sizeof(RCSortElement), RCSortElementAscendingSTCompare);

  return sortArray;
}

void CriticalPath::addCPElement(IloResourceConstraint rc) {
```

```
  if (_cpSize < _cpArray.getSize())
    // reuse
    _cpArray[_cpSize] = rc;
  else
    // add
    _cpArray.add(rc);

  _cpSize++;
}

void CriticalPath::pickRandomCP(RCSortElement *sortArray,
                                IloSchedulerSolution solution,
                                IloInt nbCriticalRCs) {
  _cpSize = 0;
  IloInt endRC = 0;
  IloInt i = -1;
  while(i < nbCriticalRCs - 1) {
    // skip elements not on the same critical path as rc
    IloInt nextIndexStart = i + 1;
    while((nextIndexStart < nbCriticalRCs) &&
          (sortArray[nextIndexStart].getStartValue() < endRC))
      nextIndexStart++;

    // gather elements that are successors of rc on the critical
    // path. There may be more than one
    IloInt nextIndexEnd = nextIndexStart + 1;
    while((nextIndexEnd < nbCriticalRCs) &&
          (sortArray[nextIndexEnd].getStartValue() == endRC))
      nextIndexEnd++;

    if (nextIndexStart < nbCriticalRCs) {

      // At this point the elements between sortArray[nextIndexStart]
      // and sortArray[nextIndexEnd-1] inclusive are all successors to
      // rc on some critical path.
      // We randomly pick one of them.
      IloInt randIndex = nextIndexStart;
      IloInt indexDiff = nextIndexEnd - nextIndexStart;
      if (indexDiff > 1)
        randIndex += _randGen.getInt(indexDiff);

      IloResourceConstraint next = sortArray[randIndex].getRC();
      addCPElement(next);

      i = randIndex;
      endRC = (IloInt)solution.getEndMin(next.getActivity());
    }
    else
      i = nbCriticalRCs;
  }
}

IloArray<IloResourceConstraint>& CriticalPath::computeCP(
                                      IloSchedulerSolution solution,
                                      IloInt& cpSize) {
  IloInt nbCriticalRCs = 0;
  RCSortElement *sortArray = getCriticalRCs(solution, nbCriticalRCs);
```

```
  // Pick the RCs in one (randomly selected) critical path
  pickRandomCP(sortArray, solution, nbCriticalRCs);
  delete [] sortArray;

  cpSize = _cpSize;
  return _cpArray;
}

////////////////////////////////////////////////////////////////////////////
//
// NEIGHBORHOOD:  The N1 neighborhood found by swapping activities
//                   on the critical path
//
////////////////////////////////////////////////////////////////////////////
class N1NHoodI : public IloNHoodI {
private:
  IloSchedulerSolution _solution;

  CriticalPath *_cp;

  IloArray<SwapRC*> _moves;
  IloInt _nbMoves;

  void addMove(IloSolver solver,
               IloResourceConstraint rc, IloResourceConstraint after);

public:
  N1NHoodI(IloEnv env, CriticalPath *cp, const char *name = 0 ) :
    IloNHoodI (env, name), _cp(cp), _moves(env), _nbMoves(0) {}
  ~N1NHoodI() {}

  void start(IloSolver solver, IloSolution soln);

  IloSolution define(IloSolver solver, IloInt i) {
    return (_moves[i])->createDelta(solver.getEnv(), _solution);
  }

  IloInt getSize(IloSolver) const { return _nbMoves; }

  void reset() { _solution = 0; _nbMoves = 0; }
};

void N1NHoodI::addMove(IloSolver solver,
                       IloResourceConstraint rc, IloResourceConstraint after) {
  SwapRC *move = new (solver.getEnv()) SwapRC(rc, after);
  if (_nbMoves < _moves.getSize())
    _moves[_nbMoves] = move;
  else
    _moves.add(move);
  _nbMoves++;
}

void N1NHoodI::start(IloSolver solver, IloSolution soln) {

  _solution = IloSchedulerSolution(soln);
  _nbMoves = 0;

  IloInt cpSize = 0;
```

```
  IloArray<IloResourceConstraint> cpArray = _cp->computeCP(_solution, cpSize);

  // create the moves
  for(IloInt i = 0; i < cpSize - 1; ++i) {
    IloResourceConstraint rc = cpArray[i];
    IloResourceConstraint after = cpArray[i + 1];

    if ((_solution.getNextRC(rc).getImpl() != 0) &&
        (after.getImpl() == _solution.getNextRC(rc).getImpl())) {

      addMove(solver, rc, after);
    }
  }
}


IloNHood N1NHood(IloEnv env, CriticalPath *cp, const char *name = 0) {
  return new (env) N1NHoodI(env, cp, name);
}



////////////////////////////////////////////////////////////////////////////
//
// TABU LIST
//
////////////////////////////////////////////////////////////////////////////

class MyTabuI : public IloMetaHeuristicI {
private:
  IloSchedulerSolution _currentSolution;

  IloInt _maxTabuSize;
  IloArray<SwapRC*> _tabuList;
  IloInt _nbTabuEles;
  IloInt _nextTabuEle;

  IloNumVar _costVar;
  IloNum _bestCost;

public:
  MyTabuI(IloEnv env, IloNumVar costVar, const char *name = 0)
    : IloMetaHeuristicI(env, name)
    , _currentSolution(0)
    , _maxTabuSize(10)
    , _tabuList(env, _maxTabuSize)
    , _nbTabuEles(0)
    , _nextTabuEle(0)
    , _costVar(costVar)
    , _bestCost(IloInfinity)
  {
    for(IloInt i=0; i<_maxTabuSize; ++i)
      _tabuList[i] = 0;
  }

  virtual IloBool start(IloSolver, IloSolution);
  virtual IloBool test(IloSolver, IloSolution);
  virtual void notify(IloSolver, IloSolution);
```

```
  virtual IloBool complete();

  virtual void reset() {
    _bestCost = IloInfinity; _nbTabuEles = 0; _nextTabuEle = 0;
  }
};

IloBool MyTabuI::start(IloSolver, IloSolution solution) {
  _currentSolution = IloSchedulerSolution(solution);

  if (solution.isObjectiveSet()) {
    IloNumVar obj = solution.getObjectiveVar();
    if (obj.getImpl() && obj.getImpl() == _costVar.getImpl()) {
      if (solution.getObjectiveValue() < _bestCost)
        _bestCost = solution.getObjectiveValue();
    }
    else
      throw "MyTabuI::start - Solution objective must be a simple variable.";
  }
  else
    throw "MyTabuI::start - Solution has no objective.";

  return IloTrue;
}


IloBool MyTabuI::test(IloSolver solver, IloSolution delta) {
  if (solver.getMin(_costVar) < _bestCost)
    return IloTrue;

  for(IloInt i = 0; i < _nbTabuEles; ++i) {
    if ((_tabuList[i] != 0) && _tabuList[i]->isReverseMove(delta))
      return IloFalse;
  }

  return IloTrue;
}

void MyTabuI::notify(IloSolver solver, IloSolution delta) {
  SwapRC *move = SwapRC::findSwapFromDelta(solver, delta);
  assert(move != 0);

  _tabuList[_nextTabuEle] = move;

  _nextTabuEle++;
  _nextTabuEle %= _maxTabuSize;

  if (_nbTabuEles < _maxTabuSize)
    _nbTabuEles++;
}


IloBool MyTabuI::complete() {
  // Remove the element that has been in the list the longest
  // time. Insert the youngest element in its place.
  IloInt youngest = _nextTabuEle - 1;
  if (youngest < 0)
    youngest = _nbTabuEles - 1;
```

```
  if ((youngest < 0) ||
      (_tabuList[youngest] == _tabuList[_nextTabuEle]))
    return IloTrue;

  _tabuList[_nextTabuEle] = _tabuList[youngest];

  _nextTabuEle++;
  _nextTabuEle %= _maxTabuSize;

  return IloFalse;
}

IloMetaHeuristic MyTabu(IloEnv env, IloNumVar costVar,
                        const char *name = 0) {
  return (IloMetaHeuristicI *) new (env) MyTabuI(env, costVar, name);
}



////////////////////////////////////////////////////////////////////////////
//
// PROBLEM SOLVING
//
////////////////////////////////////////////////////////////////////////////

void FindInitialSolution(IloModel model,
                         IloSchedulerSolution globalSolution,
                         IloSchedulerSolution lsSolution,
                         IloNumVar costVar) {
  IloEnv env = model.getEnv();
  IloSolver solver(model);
  IlcScheduler scheduler(solver);

  IloGoal g = IloRankForward(env) && IloInstantiate(env, costVar);
  solver.startNewSearch(g);

  if(solver.next()) {
    IloNum best = solver.getValue(costVar);
    solver.out() << “Solution for Cost “ <<  best << endl;
    solver.printInformation();
    globalSolution.store(scheduler);
    lsSolution.store(scheduler);
  }

  solver.endSearch();
  solver.end();
}

IloSchedulerSolution CreateLSSolution(IloEnv env,
                                      IloSchedulerSolution globalSolution) {


  /* CREATE LOCAL SEARCH SOLUTION */
  IloSchedulerSolution lsSolution = globalSolution.makeClone(env);

  for (IloIterator <IloResourceConstraint> iter(env); iter.ok(); ++iter)
    lsSolution.add(*iter, IloRestoreRCNext);
```

```
  for (IloIterator <IloResource> resIter(env); resIter.ok(); ++resIter)
    lsSolution.add(*resIter);

  return lsSolution;
}

void
SolveModel(IloModel model,
           IloNumVar makespan,
           IloSchedulerSolution& globalSolution)
{
  IloEnv env = model.getEnv();

  /* CREATE LOCAL SEARCH SOLUTION */
  IloSchedulerSolution lsSolution = CreateLSSolution(env, globalSolution);
  IloObjective obj = IloMinimize(env, makespan);
  lsSolution.getSolution().add(obj);

  globalSolution.getSolution().add(makespan);

  /* GENERATE AN INITIAL SOLUTION. */
  FindInitialSolution(model, globalSolution, lsSolution, makespan);
  IloNum best = globalSolution.getMax(makespan);
  env.out() << "Initial solution:" << endl;
  PrintSolution(globalSolution);

  IloSchedulerEnv schedEnv(env);
  schedEnv.getResourceParam().ignoreCapacityConstraints();

  IloSolver lsSolver(model);
  IlcScheduler lsScheduler(lsSolver);

  /* HILL CLIMB */
  CriticalPath cp(env);
  IloNHood nhood = N1NHood(env, &cp);
  IloGoal greedyMove = IloSingleMove(env,
                                     lsSolution,
                                     nhood,
                                     IloImprove(env),
                                     IloFirstSolution(env),
                                     IloInstantiate(env, makespan));

  IloInt maxIter = 100;
  IloInt movesDone = 0;
  while((movesDone < maxIter) && lsSolver.solve(greedyMove)) {
    IloNum cost = lsSolution.getSolution().getObjectiveValue();
    lsSolver.out() << "Move: " << movesDone << ":\t";
    ++movesDone;
    lsSolver.out() << "solution at cost: " << cost
                   << " ** HC\n";
    best = cost;
    globalSolution.store(lsScheduler);
  }

  /* TABU */
  IloMetaHeuristic mh = MyTabu(env, makespan);
  IloGoal move = IloSingleMove(env,
```

```
                                lsSolution,
                                nhood,
                                mh,
                                IloMinimizeVar(env, makespan),
                                IloInstantiate(env, makespan));

  for(IloInt i = movesDone; i < maxIter; ++i) {
    lsSolver.out() << "Move: " << i << ":\t";
    if (!lsSolver.solve(move)) {
      lsSolver.out() << "no solution" << endl;
      if ((nhood.getSize(lsSolver) == 0) || mh.complete())
        break;
    }
    else {
      IloNum cost = lsSolution.getSolution().getObjectiveValue();
      lsSolver.out() << "solution at cost: " << cost;
      if (cost < best) {
        globalSolution.store(lsScheduler);
        best = cost;
        lsSolver.out() << " **";
      }
      lsSolver.out() << endl;
    }
  }


  env.out() << "Final solution Cost: " << best << endl;

  PrintSolution(globalSolution);
  lsSolver.end();
}

////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////

void
InitParameters(int argc,
               char** argv,
               IloInt& numberOfJobs,
               IloInt& numberOfResources,
               IloInt*& resourceNumbers,
               IloInt*& durations)
{
  if (argc > 1) {
    IloInt number = atol(argv[1]);
    if (number == 10) {
      numberOfJobs = 10;
      numberOfResources = 10;
      resourceNumbers = ResourceNumbers10;
      durations = Durations10;
    }
    else if (number == 20) {
      numberOfJobs = 20;
      numberOfResources = 5;
      resourceNumbers = ResourceNumbers20;
```

```
      durations = Durations20;
    }
  }
}

int main(int argc, char** argv)
{
  try {
    IloEnv env;

    IloInt numberOfJobs = 6;
    IloInt numberOfResources = 6;
    IloInt* resourceNumbers = ResourceNumbers06;
    IloInt* durations = Durations06;
    InitParameters(argc,
                   argv,
                   numberOfJobs,
                   numberOfResources,
                   resourceNumbers,
                   durations);
    IloNumVar makespan;
    IloSchedulerSolution solution;
    IloModel model = DefineModel(env,
                                 numberOfJobs,
                                 numberOfResources,
                                 resourceNumbers,
                                 durations,
                                 makespan,
                                 solution);

    SolveModel(model,
               makespan,
               solution);
    env.end();
  }
  catch (IloException& exc) {
    cout << exc << endl;
  }

  return 0;
}
```

C H A P T E R

28

Tabu Search for the Jobshop Problem with Alternatives

The previous two chapters introduced the use of the neighborhood and metaheuristic objects provided in IBM© ILOG© Solver in the context of scheduling. In both of those examples there was a single type of decision to be made: the sequence of activities on each resource. In more realistic scheduling problems, it is likely that there will be a wider space of possible decisions. As noted in *Part I, Getting Started with Scheduler*, there are joint scheduling and resource allocation problems where both the resources assigned and the timing of the activities must be decided. In this chapter, we take such a problem and construct a local search algorithm.

Problem Description

The problems used in this chapter are the standard job shop scheduling problems, extended so that each activity requires one of a set of two resources rather than having a predefined resource requirement. Furthermore, the processing time of the activity depends on the resource to which it is assigned. This problem is also addressed in Chapter 36, *Scheduling with Alternative Resources: Interleaving Resource Allocation and Scheduling Decisions*.

Solving the Problem

The local search technique implemented in this chapter builds on top of the techniques introduced in the previous two chapters. In particular, we use the same tabu search implementation, the `N1NHood` neighborhood, and the same critical path calculation. In fact, the central change in the implementation for this chapter is the creation of a new neighborhood to allow activities to be relocated from one resource to another. The new neighborhood implementation requires a number of minor changes in the other objects.

Calculating the Critical Path

The critical path calculation is the same as that performed in the previous chapter. However, both the `N1NHood` and the new neighborhood (see Creating the Relocate Neighborhood) require the critical path to be calculated in order to generate moves. With our previous implementation, we would unnecessarily calculate the critical path twice, once for each neighborhood. To avoid this we add a `_needsRecomputing` flag to the `CriticalPath` object. This flag can be set using a public method:

```
  void needsRecomputing() { _needsRecomputing = IloTrue; }
```

The `CriticalPath::computeCP` method is modified to check this flag and only recompute if it is `IloTrue`.

```
IloArray<IloResourceConstraint>& CriticalPath::computeCP(
                                      IloSchedulerSolution solution,
                                      IloInt& cpSize) {
  if (_needsRecomputing) {
    IloInt nbCriticalRCs = 0;
    RCSortElement *sortArray = getCriticalRCs(solution, nbCriticalRCs);

    // pick one (randomly selected) critical path
    pickRandomCP(sortArray, solution, nbCriticalRCs);
    delete [] sortArray;
    _needsRecomputing = IloFalse;
  }

  cpSize = _cpSize;
  return _cpArray;
}
```

The `CriticalPath` implementation is otherwise identical to that used in the previous two chapters.

Creating an IloSchedulerSolution

The `IloSchedulerSolution` is created in the same way as it was created in the previous chapter, with one difference. Since we have two types of decision variables—the next

resource constraint and the selected resource of each resource constraint—we need to restore both. To do this we set the restorable status of the resource constraints in the solution to a value that specifies that both the next and the resource selection should be restored.

```
for (IloIterator <IloResourceConstraint> iter(env); iter.ok(); ++iter)
  lsSolution.setRestorable(*iter,
                           IloRestoreRCNext | IloRestoreRCSelected);
```

We do not add the resource constraint to the solution because it was already added to the solution during the model definition.

Finding the First Solution

The first solution is found with a simple predefined goal that first assigns each resource constraint to a resource, then ranks each resource, and finally instantiates the makespan variable.

```
IloGoal g = IloAssignAlternative(env) &&
  IloRankForward(env) &&
  IloInstantiate(env, costVar);
```

Modifying the SwapRC Move

As we did in the previous chapter, we define special move classes to organize the information about a single local search move. Instances of these classes are then created and manipulated by the neighborhood classes.

Since we have two types of moves and we want to treat them equivalently from the perspective of the metaheuristic, we implement an abstract class `LSMove`.

```
class LSMove {
public:
  LSMove() {}
  virtual ~LSMove() {}

  virtual IloBool isReverseMove(IloSolution delta) const = 0;
  virtual IloSolution createDelta(IloEnv, IloSchedulerSolution) = 0;
};
```

We redefine the `SwapRC` class from the previous chapter to inherit from `LSMove`. The only change to the `SwapRC` class is an additional test in the `findSwapFromDelta` method. For each resource constraint in the delta solution we ensure that it selects the same resource as in

the current solution. If a resource selection has changed, we know that the delta solution does not represent a SwapRC move.

```
for(IloSchedulerSolution::ResourceConstraintIterator preiter(solution);
    preiter.ok(); ++preiter) {
  IloResourceConstraint rc = *preiter;
  if (solution.getSelected(rc).getImpl() !=
      current.getSelected(rc).getImpl())
    return 0;
}
```

The other components of the SwapRC move are identical to what was presented in the previous chapter.

Creating the RelocateRC Move

The RelocateRC move represents the movement of a resource constraint from one resource in its alternative resource set to another and the insertion of the resource constraint between two other resource constraints on the latter resource. The class represents the resource constraint that is relocated (_relocated), the resource to which it is moved to (_to), and the resource constraint that it is next to on the new resource (_beforeInsertion).

Accessors are defined for the data members and the class as a whole inherits from `LSMove`.

```
class RelocateRC : public LSMove {
  // This class represents the data to perform a relocate of _and
  // _after in the sequence
private:
  IloResourceConstraint _relocated;
  IloResource _to;
  IloResourceConstraint _beforeInsertion;

public:
  RelocateRC() : _relocated(0),_to(0),_beforeInsertion(0) {}
  RelocateRC(IloResourceConstraint rc,
         IloResource to,
         IloResourceConstraint before)
    : _relocated(rc), _to(to), _beforeInsertion(before) {}
  virtual ~RelocateRC() {}

  IloResourceConstraint getRC() const { return _relocated; }
  IloResource getTo() const { return _to; }

  void setRC(IloResourceConstraint rc) { _relocated = rc; }
  void setTo(IloResource res) { _to = res; }

  void setBeforeInsertion(IloResourceConstraint before) {
    _beforeInsertion = before;
  }

  virtual IloBool isReverseMove(IloSolution delta) const {
    IloSchedulerSolution schedDelta(delta);
    return (schedDelta.contains(_relocated) &&
            (schedDelta.getSelected(_relocated).getImpl() != _to.getImpl()));
  }

  virtual IloSolution createDelta(IloEnv, IloSchedulerSolution);
  static RelocateRC* findRelocateFromDelta(IloSolver,
                                           IloSolution, IloSchedulerSolution);
};
```

The `isReverseMove` method returns `IloTrue` if the delta solution places the `_relocated` resource constraint on a resource other than `_to`.

Two additional functions are defined:

- `createDelta` and
- `findRelocateFromDelta`.

Recall that the `createDelta` function creates an `IloSchedulerSolution` object that represents the changes necessary to implement a move. In detail, we do the following in the `createDelta` method.

First, we reassign the selected resource of the _relocated resource constraint and unset its next pointer if it has one in the current solution.

```
relocateDelta.add(_relocated, current.getRestorable(_relocated));
relocateDelta.setSelected(_relocated, _to);
if (current.hasNextRC(_relocated))
  relocateDelta.unsetNext(_relocated, current.getNextRC(_relocated));
```

We then set the next pointer of _relocated to be either the setup resource constraint on the _to resource or to the resource constraint that is next after _beforeInsertion in the current solution.

```
IloResourceConstraint afterInsertion;
if (_beforeInsertion.getImpl() == 0)
  // insert _relocated as the first rc on _to
  afterInsertion = current.getSetupRC(_to);
else
  afterInsertion = current.getNextRC(_beforeInsertion);

if (afterInsertion.getImpl() != 0)
  relocateDelta.setNext(_relocated, afterInsertion);
```

It is possible that _relocated will have no next resource constraint in the case where it is inserted as the last resource constraint on the _to resource.

We then add the _beforeInsertion resource constraint to the delta solution and set its next to be _relocated. Again, it is possible that _beforeInsertion does not exist when _relocated is inserted as the first resource constraint on the _to resource.

```
if (_beforeInsertion.getImpl() != 0) {
  relocateDelta.add(_beforeInsertion,
                    current.getRestorable(_beforeInsertion));
relocateDelta.getSolution().copy(_beforeInsertion, current);
  if (relocateDelta.hasNextRC(_beforeInsertion))
    relocateDelta.unsetNext(_beforeInsertion,
                            current.getNextRC(_beforeInsertion));
  relocateDelta.setNext(_beforeInsertion, _relocated);
}
```

The final step is to change the resource constraint that comes before `_relocated` in the current solution. If it exists, we need to change its next pointer to point to the resource constraint that comes after `_relocated` in the current solution.

```
  IloResourceConstraint oldBefore =
    current.getPrevRC(_relocated);
  if (oldBefore.getImpl() != 0) {
    relocateDelta.add(oldBefore,
                      current.getRestorable(oldBefore));
relocateDelta.getSolution().copy(oldBefore, current);
    relocateDelta.unsetNext(oldBefore, _relocated);
    if (current.hasNextRC(_relocated))
      relocateDelta.setNext(oldBefore, current.getNextRC(_relocated));
  }
```

The `RelocateRC::findRelocateFromDelta` creates an instance of `RelocateRC` from a corresponding `IloSchedulerSolution`. If the `IloSchedulerSolution` does not actually represent a `RelocateRC` a NULL pointer is returned.

This method first finds a resource constraint whose assigned resource has been changed in the delta solution and then finds the resource constraint that is not next to any of the resource constraints in the delta. By comparing these two resource constraints, it is possible to recreate the `RelocateRC` move.

```
RelocateRC* RelocateRC::findRelocateFromDelta(IloSolver solver,
                                  IloSolution delta,
                                  IloSchedulerSolution current) {
  // Try to generate a RelocateRC move from the information in delta.
  // This may fail as delta does not necessarily represent a SwapRC
  // move
  IloSchedulerSolution solution(delta);
  IloResourceConstraint resChange;
  for(IloSchedulerSolution::ResourceConstraintIterator iter2(solution);
      iter2.ok() && (resChange.getImpl() == 0); ++iter2) {
    resChange = *iter2;
    if (solution.getSelected(resChange).getImpl() ==
        current.getSelected(resChange).getImpl())
      resChange = 0;
  }

  if (0 == resChange.getImpl())
    return 0;

  // find the element that is not next of any element in the delta
  IloResourceConstraint first;
  for(IloSchedulerSolution::ResourceConstraintIterator iter(solution);
      iter.ok() && (0 == first.getImpl()); ++iter) {
    first = *iter;
    for(IloSchedulerSolution::ResourceConstraintIterator iter3(solution);
        iter3.ok() && (0 != first.getImpl()); ++iter3) {
      IloResourceConstraint rc = *iter3;
      if (rc.getImpl() != first.getImpl()) {
        if (solution.hasNextRC(rc) &&
            (solution.getNextRC(rc).getImpl() == first.getImpl()))
          first = 0;
```

```
      }
    }
  }

  assert(0 != first.getImpl());

  if (first.getImpl() == resChange.getImpl())
    return new (solver.getEnv()) RelocateRC(first,
                                            solution.getSelected(first),
                                            0);
  else
    return new (solver.getEnv()) RelocateRC(resChange,
                                            solution.getSelected(resChange),
                                            first);
}
```

Creating an Abstract Neighborhood Class

Since we have two neighborhoods (one for swapping and one for relocating), we first create an abstract neighborhood class that inherits from `IloNHoodI` and which contains the functionality that is common to both concrete neighborhoods. In this case, the `MyNHoodI` class contains private and protected methods and data to manage the current solution, the set of possible moves, and the recalculation of the critical path and the previous pointers.

```
private:
  void resetCache()  { _cp->needsRecomputing(); }

protected:
  IloSchedulerSolution _solution;

  CriticalPath *_cp;

  IloArray<LSMove*> _moves;
  IloInt _nbMoves;

  void realAddMove(LSMove *m) {
    if (_nbMoves < _moves.getSize()) _moves[_nbMoves] = m;
    else _moves.add(m);
    _nbMoves++;
  }
```

The virtual methods inherited from `IloNHoodI` are also redefined. The `start` method stores the new solution and recalculates the previous pointers.

```
  void start(IloSolver, IloSolution soln) {
    _solution = IloSchedulerSolution(soln);
    _nbMoves = 0;
  }
```

The `define` and `getSize` methods are identical to what was implemented in the previous chapter as the `LSMove` subclass creates the delta solution and the size of the neighborhood is stored in the `_nbMoves` data member.

```
IloSolution define(IloSolver solver, IloInt i) {
  return (_moves[i])->createDelta(solver.getEnv(), _solution);
}

IloInt getSize(IloSolver) const { return _nbMoves; }
```

Finally, the set of methods for maintaining the state of the neighborhood make use of the `resetCache` method described above.

```
void notify(IloSolver, IloInt) { resetCache(); }
void notifyOther (IloSolver, IloSolution) { resetCache(); }
void reset() { _solution = 0; _nbMoves = 0; resetCache(); }
```

Modifying the N1 Neighborhood

Given that much of the implementation of the N1 neighborhood in the previous chapter is now implemented in the abstract `MyNHoodI` class, the `N1NHoodI` is quite straightforward. Two methods are defined. The `addMove` method creates an instance of the `SwapRC` class and calls the `MyNHoodI` function to add it to the list of moves.

```
void addMove(IloSolver solver,
             IloResourceConstraint rc, IloResourceConstraint after) {
  SwapRC *move = new (solver.getEnv()) SwapRC(rc, after);
  MyNHoodI::realAddMove(move);
}
```

The `start` method has the same functionality as the `N1HoodI::start` method presented in the previous chapter, but the fact that some of the functionality is performed in the parent method means that the code is shorter. Refer to Creating the N1 Neighborhood for more details.

```
void start(IloSolver, IloSolution soln) {
  _solution = IloSchedulerSolution(soln);
  _nbMoves = 0;
}
```

Creating the Relocate Neighborhood

The relocate neighborhood defines a set of moves consisting of assigning each resource constraint on a randomly selected critical path to an alternative resource, inserting it between each pair of consecutive activities on that resource.

The implementation follows the same pattern as the N1 neighborhood implementation, inheriting from `MyNHoodI` and defining two methods: `addMove` and `start`. The former function is simple, as it creates an instance of `RelocateRC` and calls the `MyNHoodI::realAddMove` method.

```
  void addMove(IloSolver solver,
               IloResourceConstraint relocate, IloResource to,
               IloResourceConstraint beforeInsertion) {
    RelocateRC *move = new (solver.getEnv())
      RelocateRC(relocate, to, beforeInsertion);
    MyNHoodI::realAddMove(move);
  }
```

The `start` method creates the set of neighbors from the given solution. For each resource constraint on the critical path, the set of alternative resources is iterated over and for each one, a set of moves is created. Each move inserts the original resource constraint between a pair of consecutive resource constraints on the alternative resource or as the first or last resource constraint on the resource.

```
void RelocateNHoodI::start(IloSolver solver, IloSolution soln) {
  MyNHoodI::start(solver, soln);

  IloInt cpSize = 0;
  IloArray<IloResourceConstraint> cpArray = _cp->computeCP(_solution, cpSize);

  // create the moves
  for(IloInt i = 0; i < cpSize; ++i) {
    IloResourceConstraint rc = cpArray[i];
    IloResource selected = _solution.getSelected(rc);
    if (rc.hasAltResSet()) {
      for(IloAltResSet::Iterator iter(rc.getAltResSet());
          iter.ok(); ++iter) {
        IloResource res = *iter;
        if (selected.getImpl() != res.getImpl()) {

          // add move to insert rc first on res
          addMove(solver, rc, res, 0);
          IloResourceConstraint prev = _solution.getSetupRC(res);
          while(prev.getImpl() != 0) {
            addMove(solver, rc, res, prev);
            prev = _solution.getNextRC(prev);
          }
        }
      }
    }
  }
}
```

Figure 28.1 shows part of the neighborhood of a solution, assuming that C1 and B1 are on the critical path that is chosen and that C1 can execute on either R0 or R1 and B1 can execute on either R1 or R2. There are five possible relocations of C1: between each consecutive pair of activities on R0 and as the first or last resource constraint on R0. Similarly, there are four possible relocations of B1. The neighborhood also contains the

relocation moves of the other resource constraints on the chosen critical path (that is, C0 and B2).

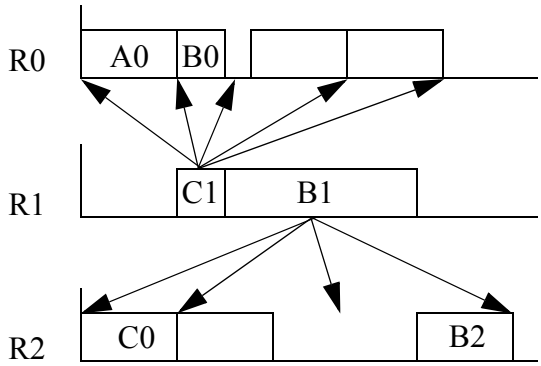


Figure 28.1 *A Partial Neighborhood of a Solution*

The Tabu Search Metaheuristic

The tabu search metaheuristic is identical to that implemented in the previous chapter with two exceptions. First, the list of tabu moves is now an array of pointers to `LSMove` instances rather than to `SwapRC` instances.

```
IloArray<LSMove*> _tabuList;
```

Second, the `notify` method has to deal with the fact that it could be called when either a `SwapRC` move or a `RelocateRC` move is chosen. To account for this we modify the first few lines of the method so that it can identify the correct move represented by the delta solution.

```
LSMove *move = 0;
RelocateRC *relocateMove =
  RelocateRC::findRelocateFromDelta(solver, delta, _currentSolution);
if (0 != relocateMove)
  move = relocateMove;
else {
  SwapRC *swapMove = SwapRC::findSwapFromDelta(solver,
                                               delta, _currentSolution);
  if (0 != swapMove)
    move = swapMove;
}
```

Other than these two minor modifications, the implementation of the tabu metaheuristic is identical to that in the previous chapter.

Scheduler Parameters for Local Search

Since the relocate neighborhood involves moving resource constraints between resources, it is possible that a move will create a long cycle of temporal constraints. If the problem has a long horizon, it may take a significant amount of time for this cycle to be detected with the standard levels of constraint propagation. To avoid such a situation, we set the precedence enforcement on the Scheduler environment to `IloHigh`. This level of enforcement uses the schedule precedence graph to maintain the precedence graph among the activities in the schedule. A loop in this graph is detected quickly.

As in the previous chapter, we do not need the propagation of the resource capacity constraints to ensure soundness because each move defines a sequence of resource constraints on each resource. Therefore, for the local search, we ignore the resource capacity constraints.

```
IloSchedulerEnv schedEnv(env);
schedEnv.setPrecedenceEnforcement(IloHigh);
schedEnv.getResourceParam().ignoreCapacityConstraints();
```

The Local Search

The code that defines the local search, using the objects presented above, has only a few minor changes from the code of the previous chapter. Because we now have two neighborhoods, the definition of the hill-climbing move is as follows:

```
CriticalPath cp(env);
IloNHood nhood = N1NHood(env, &cp) + RelocateNHood(env, &cp);
IloGoal greedyMove = IloSingleMove(env,
                                   lsSolution,
                                   nhood,
                                   IloImprove(env),
                                   IloFirstSolution(env),
                                   IloInstantiate(env, makespan));
```

The `IloNHood` instance is composed of two sub-neighborhoods (`N1Hood` and `RelocateNHood`) and both of these neighborhoods are created with the same `CriticalPath` instances.

The actual hill-climbing loop is identical to that used in the previous chapter.

Similarly, the tabu search move is the same as defined in the previous chapter with the only exception being that the neighborhood used is composed of our two sub-neighborhoods.

```
IloMetaHeuristic mh = MyTabu(env, makespan);
IloGoal move = IloSingleMove(env,
                             lsSolution,
                             nhood,
                             mh,
                             IloMinimizeVar(env, makespan),
                             IloInstantiate(env, makespan));
```

As with the hill-climbing code, the while-loop that implements the search is identical to that in the previous chapter.

Complete Program

You can see the entire program `altls.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>
#include <ilsolver/iimls.h>

ILOSTLBEGIN

IloInt ResourceNumbers06 [] = {2, 0, 1, 3, 5, 4,
                               1, 2, 4, 5, 0, 3,
                               2, 3, 5, 0, 1, 4,
                               1, 0, 2, 3, 4, 5,
                               2, 1, 4, 5, 0, 3,
                               1, 3, 5, 0, 4, 2};

IloNum Durations06 [] = { 1,  3,  6,  7,  3,  6,
                          8,  5, 10, 10, 10,  4,
                          5,  4,  8,  9,  1,  7,
                          5,  5,  5,  3,  8,  9,
                          9,  3,  5,  4,  3,  1,
                          3,  3,  9, 10,  4,  1};

IloInt ResourceNumbers10 [] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
                               0, 2, 4, 9, 3, 1, 6, 5, 7, 8,
                               1, 0, 3, 2, 8, 5, 7, 6, 9, 4,
                               1, 2, 0, 4, 6, 8, 7, 3, 9, 5,
                               2, 0, 1, 5, 3, 4, 8, 7, 9, 6,
                               2, 1, 5, 3, 8, 9, 0, 6, 4, 7,
                               1, 0, 3, 2, 6, 5, 9, 8, 7, 4,
                               2, 0, 1, 5, 4, 6, 8, 9, 7, 3,
                               0, 1, 3, 5, 2, 9, 6, 7, 4, 8,
                               1, 0, 2, 6, 8, 9, 5, 3, 4, 7};

IloNum Durations10 [] = {29, 78,  9, 36, 49, 11, 62, 56, 44, 21,
                         43, 90, 75, 11, 69, 28, 46, 46, 72, 30,
                         91, 85, 39, 74, 90, 10, 12, 89, 45, 33,
                         81, 95, 71, 99,  9, 52, 85, 98, 22, 43,
```

```
                          14,  6, 22, 61, 26, 69, 21, 49, 72, 53,
                          84,  2, 52, 95, 48, 72, 47, 65,  6, 25,
                          46, 37, 61, 13, 32, 21, 32, 89, 30, 55,
                          31, 86, 46, 74, 32, 88, 19, 48, 36, 79,
                          76, 69, 76, 51, 85, 11, 40, 89, 26, 74,
                          85, 13, 61,  7, 64, 76, 47, 52, 90, 45};

IloInt ResourceNumbers20 [] = {0, 1, 2, 3, 4,
                               0, 1, 3, 2, 4,
                               1, 0, 2, 4, 3,
                               1, 0, 4, 2, 3,
                               2, 1, 0, 3, 4,
                               2, 1, 4, 0, 3,
                               1, 0, 2, 3, 4,
                               2, 1, 0, 3, 4,
                               0, 3, 2, 1, 4,
                               1, 2, 0, 3, 4,
                               1, 3, 0, 4, 2,
                               2, 0, 1, 3, 4,
                               0, 2, 1, 3, 4,
                               2, 0, 1, 3, 4,
                               0, 1, 4, 2, 3,
                               1, 0, 3, 4, 2,
                               0, 2, 1, 3, 4,
                               0, 1, 4, 2, 3,
                               1, 2, 0, 3, 4,
                               0, 1, 2, 3, 4};

IloNum Durations20 [] = {29,  9, 49, 62, 44,
                         43, 75, 69, 46, 72,
                         91, 39, 90, 12, 45,
                         81, 71,  9, 85, 22,
                         14, 22, 26, 21, 72,
                         84, 52, 48, 47,  6,
                         46, 61, 32, 32, 30,
                         31, 46, 32, 19, 36,
                         76, 76, 85, 40, 26,
                         85, 61, 64, 47, 90,
                         78, 36, 11, 56, 21,
                         90, 11, 28, 46, 30,
                         85, 74, 10, 89, 33,
                         95, 99, 52, 98, 43,
                          6, 61, 69, 49, 53,
                          2, 95, 72, 65, 25,
                         37, 13, 21, 89, 55,
                         86, 74, 88, 48, 79,
                         69, 51, 11, 89, 74,
                         13,  7, 76, 52, 45};


////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////

void PrintRange(IloEnv& env, IloNum min, IloNum max) {
```

```
  if (min == max)
    env.out() << (IlcInt)min;
  else
    env.out() << (IlcInt)min << ".." << (IlcInt)max;
}

void PrintSolution(IloEnv& env,
                   const IloSchedulerSolution solution,
                   const IloNumVar makespan)
{
  env.out() << "Solution with makespan ["
            << solution.getMin(makespan) << ".."
            << solution.getMax(makespan) << "]" << endl;

  for (IloSchedulerSolution::ResourceConstraintIterator
                                          iter(solution);
       iter.ok();
       ++iter)
  {
    IloResourceConstraint rc = *iter;
    if (!solution.isResourceSelected(rc))
      IloSchedulerException("No resource assigned!");

    IloActivity activity = rc.getActivity();
    env.out() << activity.getName() << "[";
    PrintRange(env,
               solution.getStartMin(activity),
               solution.getStartMax(activity));
    env.out() << " -- ";
    PrintRange(env,
               solution.getDurationMin(activity),
               solution.getDurationMax(activity));
    env.out() << " --> ";
    PrintRange(env,
               solution.getEndMin(activity),
               solution.getEndMax(activity));
    env.out() << "]: " << solution.getSelected(rc).getName()
              << endl;
  }
}

////////////////////////////////////////////////////////////////////////////
//
// MOVES
//
// These are classes that will be associated with a specific
// neighborhood and represent a particular type of local search move
//
////////////////////////////////////////////////////////////////////////////

// Abstract move object
class LSMove {
public:
  LSMove() {}
  virtual ~LSMove() {}

  virtual IloBool isReverseMove(IloSolution delta) const = 0;
  virtual IloSolution createDelta(IloEnv, IloSchedulerSolution) = 0;
```

```
};

class SwapRC : public LSMove {
  // This class represents the data to perform a swap of _rc and
  // _after in the sequence
private:
  IloResourceConstraint _rc;
  IloResourceConstraint _after;

public:
  SwapRC() : _rc(0), _after(0) {}
  SwapRC(IloResourceConstraint rc, IloResourceConstraint after)
    : _rc(rc), _after(after) {}
  virtual ~SwapRC() {}

  IloResourceConstraint getRC() const { return _rc; }
  IloResourceConstraint getAfter() const { return _after; }

  void setRC(IloResourceConstraint rc) { _rc = rc; }
  void setAfter(IloResourceConstraint after) { _after = after; }

  virtual IloBool isReverseMove(IloSolution delta) const {
    IloSchedulerSolution schedDelta(delta);
    return (schedDelta.contains(_rc) &&
            schedDelta.getNextRC(_rc).getImpl() == _after.getImpl());
  }

  virtual IloSolution createDelta(IloEnv, IloSchedulerSolution);

  static SwapRC* findSwapFromDelta(IloSolver, IloSolution,
                                   IloSchedulerSolution);
};

SwapRC* SwapRC::findSwapFromDelta(IloSolver solver,
                                  IloSolution delta,
                                  IloSchedulerSolution current) {
  // Try to generate a SwapRC move from the information in delta.  This may
  // fail as delta does not necessarily represent a SwapRC move
  IloSchedulerSolution solution(delta);

  for(IloSchedulerSolution::ResourceConstraintIterator preiter(solution);
      preiter.ok(); ++preiter) {
    IloResourceConstraint rc = *preiter;
    if (solution.getSelected(rc).getImpl() !=
        current.getSelected(rc).getImpl())
      return 0;
  }

  // find the element that is not next of any element in the delta
  IloResourceConstraint first;
  for(IloSchedulerSolution::ResourceConstraintIterator iter(solution);
      iter.ok() && (0 == first.getImpl()); ++iter) {
    first = *iter;
    for(IloSchedulerSolution::ResourceConstraintIterator iter2(solution);
        iter2.ok() && (0 != first.getImpl()); ++iter2) {
      IloResourceConstraint rc = *iter2;
      if (rc.getImpl() != first.getImpl()) {
        if (solution.hasNextRC(rc) &&
```

```
          (solution.getNextRC(rc).getImpl() == first.getImpl()))
        first = 0;
    }
  }
}

assert(0 != first.getImpl());

IloResourceConstraint second = solution.getNextRC(first);
IloResourceConstraint third = solution.getNextRC(second);
if (0 == third.getImpl())
  return new (solver.getEnv()) SwapRC(second, first);
else
  return new (solver.getEnv()) SwapRC(third, second);
}

IloSolution SwapRC::createDelta(IloEnv env, IloSchedulerSolution current) {

IloSchedulerSolution swapDelta(env);

// old order: before -> _rc -> _after -> afterAfter
// new order: before -> _after -> _rc -> afterAfter

swapDelta.add(_rc, current.getRestorable(_rc));
swapDelta.getSolution().copy(_rc, current);
swapDelta.unsetNext(_rc, current.getNextRC(_rc));

swapDelta.add(_after, current.getRestorable(_after));
swapDelta.getSolution().copy(_after, current);
if (current.hasNextRC(_after))
  swapDelta.unsetNext(_after, current.getNextRC(_after));

// change beforeBefore --> _rc to beforeBefore --> _after
//   IloResourceConstraint before = getBefore();
IloResourceConstraint before = current.getPrevRC(_rc);
if (before.getImpl() != 0) {
  swapDelta.add(before, current.getRestorable(before));
  swapDelta.getSolution().copy(before, current);
  swapDelta.unsetNext(before, current.getNextRC(before));
  swapDelta.setNext(before, _after);
}

// change _after --> afterAfter to _after --> _rc
swapDelta.setNext(_after, _rc);

// change _rc --> _after to _rc --> afterAfter
if (current.hasNextRC(_after))
  swapDelta.setNext(_rc, current.getNextRC(_after));

return swapDelta;
}

class RelocateRC : public LSMove {
// This class represents the data to perform a relocate of _and
// _after in the sequence
private:
IloResourceConstraint _relocated;
IloResource _to;
```

```
  IloResourceConstraint _beforeInsertion;

public:
  RelocateRC() : _relocated(0),_to(0),_beforeInsertion(0) {}
  RelocateRC(IloResourceConstraint rc,
         IloResource to,
         IloResourceConstraint before)
    : _relocated(rc), _to(to), _beforeInsertion(before) {}
  virtual ~RelocateRC() {}

  IloResourceConstraint getRC() const { return _relocated; }
  IloResource getTo() const { return _to; }

  void setRC(IloResourceConstraint rc) { _relocated = rc; }
  void setTo(IloResource res) { _to = res; }

  void setBeforeInsertion(IloResourceConstraint before) {
    _beforeInsertion = before;
  }

  virtual IloBool isReverseMove(IloSolution delta) const {
    IloSchedulerSolution schedDelta(delta);
    return (schedDelta.contains(_relocated) &&
            (schedDelta.getSelected(_relocated).getImpl() != _to.getImpl()));
  }

  virtual IloSolution createDelta(IloEnv, IloSchedulerSolution);
  static RelocateRC* findRelocateFromDelta(IloSolver,
                                           IloSolution, IloSchedulerSolution);
};

IloSolution RelocateRC::createDelta(IloEnv env, IloSchedulerSolution current) {
  IloSchedulerSolution relocateDelta(env);

  // old state: _relocated is not on resource _to
  // new state: _relocated is on resource _to after _beforeInsertion

  // set the selected resource of _relocated
  relocateDelta.add(_relocated, current.getRestorable(_relocated));
  relocateDelta.setSelected(_relocated, _to);
  if (current.hasNextRC(_relocated))
    relocateDelta.unsetNext(_relocated, current.getNextRC(_relocated));

  // set the next of _relocated
  IloResourceConstraint afterInsertion;
  if (_beforeInsertion.getImpl() == 0)
    // insert _relocated as the first rc on _to
    afterInsertion = current.getSetupRC(_to);
  else
    afterInsertion = current.getNextRC(_beforeInsertion);

  if (afterInsertion.getImpl() != 0)
    relocateDelta.setNext(_relocated, afterInsertion);

  // set the next of _beforeInsertion
  if (_beforeInsertion.getImpl() != 0) {
    relocateDelta.add(_beforeInsertion,
                      current.getRestorable(_beforeInsertion));
```

```
                  relocateDelta.getSolution().copy(_beforeInsertion, current);
    if (relocateDelta.hasNextRC(_beforeInsertion))
      relocateDelta.unsetNext(_beforeInsertion,
                              current.getNextRC(_beforeInsertion));
    relocateDelta.setNext(_beforeInsertion, _relocated);
  }

  // change the rc that was before _relocated in the previous solution
  IloResourceConstraint oldBefore =
    current.getPrevRC(_relocated);
  if (oldBefore.getImpl() != 0) {
    relocateDelta.add(oldBefore,
                      current.getRestorable(oldBefore));
                          relocateDelta.getSolution().copy(oldBefore, current);
    relocateDelta.unsetNext(oldBefore, _relocated);
    if (current.hasNextRC(_relocated))
      relocateDelta.setNext(oldBefore, current.getNextRC(_relocated));
  }

  return relocateDelta;
}

RelocateRC* RelocateRC::findRelocateFromDelta(IloSolver solver,
                                IloSolution delta,
                                IloSchedulerSolution current) {
  // Try to generate a RelocateRC move from the information in delta.
  // This may fail as delta does not necessarily represent a SwapRC
  // move
  IloSchedulerSolution solution(delta);
  IloResourceConstraint resChange;
  for(IloSchedulerSolution::ResourceConstraintIterator iter2(solution);
      iter2.ok() && (resChange.getImpl() == 0); ++iter2) {
    resChange = *iter2;
    if (solution.getSelected(resChange).getImpl() ==
        current.getSelected(resChange).getImpl())
      resChange = 0;
  }

  if (0 == resChange.getImpl())
    return 0;

  // find the element that is not next of any element in the delta
  IloResourceConstraint first;
  for(IloSchedulerSolution::ResourceConstraintIterator iter(solution);
      iter.ok() && (0 == first.getImpl()); ++iter) {
    first = *iter;
    for(IloSchedulerSolution::ResourceConstraintIterator iter3(solution);
        iter3.ok() && (0 != first.getImpl()); ++iter3) {
      IloResourceConstraint rc = *iter3;
      if (rc.getImpl() != first.getImpl()) {
        if (solution.hasNextRC(rc) &&
            (solution.getNextRC(rc).getImpl() == first.getImpl()))
          first = 0;
      }
    }
  }

  assert(0 != first.getImpl());
```

```
  if (first.getImpl() == resChange.getImpl())
    return new (solver.getEnv()) RelocateRC(first,
                                            solution.getSelected(first),
                                            0);
  else
    return new (solver.getEnv()) RelocateRC(resChange,
                                            solution.getSelected(resChange),
                                            first);
}


////////////////////////////////////////////////////////////////////////////
//
// UTILITIES
//
// We use some utilities classes to help carry out the features common
// to the neighborhoods we use
//
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
//
// Critical Path Object: This object computes and returns a set of
// resource constraints that constitute a randomly selected critical
// path in the current solution
//
////////////////////////////////////////////////////////////////////////////

class RCSortElement {
private:
  IloResourceConstraint _rc;
  IloInt _startValue;
  IloInt _tieBreaker;

public:
  RCSortElement() : _rc(0), _startValue(0), _tieBreaker(0) {}
  RCSortElement(IloResourceConstraint rc, IloInt startValue, IloInt tieBreaker)
    : _rc(rc), _startValue(startValue), _tieBreaker(tieBreaker) {}

  IloResourceConstraint getRC() { return _rc; }
  IloInt getStartValue() { return _startValue; }
  IloInt getTieBreaker() { return _tieBreaker; }

  void setRC(IloResourceConstraint rc) { _rc = rc; }
  void setStartValue(IloInt st) { _startValue = st; }
  void setTieBreaker(IloInt tie) { _tieBreaker = tie; }
};

static
int RCSortElementAscendingSTCompare(const void* first,
                                    const void* second)
{
  RCSortElement *a = (RCSortElement *) first;
  RCSortElement *b = (RCSortElement *) second;

  if (a->getStartValue() > b->getStartValue())
    return 1;
```

```
  else if (a->getStartValue() < b->getStartValue())
    return -1;
  else if (a->getTieBreaker() > b->getTieBreaker())
    return 1;
  else if (a->getTieBreaker() < b->getTieBreaker())
    return -1;

  return 0;
}

class CriticalPath {
private:
  IloBool _needsRecomputing;

  IloInt _cpSize;
  IloArray<IloResourceConstraint> _cpArray;

  IloRandom _randGen;

  RCSortElement *getCriticalRCs(IloSchedulerSolution, IloInt&);
  void pickRandomCP(RCSortElement *, IloSchedulerSolution,IloInt);
  void addCPElement(IloResourceConstraint);

public:
  CriticalPath(IloEnv env) :
    _needsRecomputing(IloTrue), _cpSize(0),
    _cpArray(env), _randGen(env, 98998) {}
  ~CriticalPath() {}

  void needsRecomputing() { _needsRecomputing = IloTrue; }

  IloArray<IloResourceConstraint>& computeCP(IloSchedulerSolution, IloInt&);
};

RCSortElement *CriticalPath::getCriticalRCs(IloSchedulerSolution solution,
                                            IloInt& nbCriticalRCs) {
  // count the number of RCs with assigned start times
  for(IloSchedulerSolution::ResourceConstraintIterator iter(solution);
      iter.ok(); ++iter) {
    IloResourceConstraint rc = *iter;
    IloActivity act = rc.getActivity();
    if (solution.getStartMin(act) == solution.getStartMax(act))
      ++nbCriticalRCs;
  }

  // populate array
  RCSortElement *sortArray = new RCSortElement[nbCriticalRCs];
  IloInt index = 0;
  for(IloSchedulerSolution::ResourceConstraintIterator iter2(solution);
      iter2.ok(); ++iter2) {

    IloResourceConstraint rc = *iter2;
    IloActivity act = rc.getActivity();

    if (solution.getStartMin(act) == solution.getStartMax(act)) {
      sortArray[index].setRC(rc);
      sortArray[index].setStartValue( (IloInt)solution.getStartMin(act) );
      sortArray[index].setTieBreaker(index);
```

```
      ++index;
    }
  }

  // order in ascending start time
  qsort(sortArray, nbCriticalRCs,
        sizeof(RCSortElement), RCSortElementAscendingSTCompare);

  return sortArray;
}

void CriticalPath::addCPElement(IloResourceConstraint rc) {
  if (_cpSize < _cpArray.getSize())
    // reuse
    _cpArray[_cpSize] = rc;
  else
    // add
    _cpArray.add(rc);

  _cpSize++;
}

void CriticalPath::pickRandomCP(RCSortElement *sortArray,
                                IloSchedulerSolution solution,
                                IloInt nbCriticalRCs) {
  _cpSize = 0;
  IloInt endRC = 0;
  IloInt i = -1;
  while(i < nbCriticalRCs - 1) {
    // skip elements not on the same critical path as rc
    IloInt nextIndexStart = i + 1;
    while((nextIndexStart < nbCriticalRCs) &&
          (sortArray[nextIndexStart].getStartValue() < endRC))
      nextIndexStart++;

    // gather elements that are successors of rc on the critical
    // path. There may be more than one
    IloInt nextIndexEnd = nextIndexStart + 1;
    while((nextIndexEnd < nbCriticalRCs) &&
          (sortArray[nextIndexEnd].getStartValue() == endRC))
      nextIndexEnd++;

    if (nextIndexStart < nbCriticalRCs) {

      // At this point the elements between sortArray[nextIndexStart]
      // and sortArray[nextIndexEnd-1] inclusive are all successors to
      // rc on some critical path.
      // We randomly pick one of them.
      IloInt randIndex = nextIndexStart;
      IloInt indexDiff = nextIndexEnd - nextIndexStart;
      if (indexDiff > 1)
        randIndex += _randGen.getInt(indexDiff);

      IloResourceConstraint next = sortArray[randIndex].getRC();
      addCPElement(next);

      i = randIndex;
      endRC = (IloInt)solution.getEndMin(next.getActivity());
```

```
    }
    else
      i = nbCriticalRCs;
  }

}

IloArray<IloResourceConstraint>& CriticalPath::computeCP(
                                      IloSchedulerSolution solution,
                                      IloInt& cpSize) {
  if (_needsRecomputing) {
    IloInt nbCriticalRCs = 0;
    RCSortElement *sortArray = getCriticalRCs(solution, nbCriticalRCs);

    // pick one (randomly selected) critical path
    pickRandomCP(sortArray, solution, nbCriticalRCs);
    delete [] sortArray;
    _needsRecomputing = IloFalse;
  }

  cpSize = _cpSize;
  return _cpArray;
}


////////////////////////////////////////////////////////////////////////////
//
// NEIGHBORHOODS
//
////////////////////////////////////////////////////////////////////////////

// Abstract Neighborhood
class MyNHoodI : public IloNHoodI {
private:
  void resetCache()  { _cp->needsRecomputing(); }

protected:
  IloSchedulerSolution _solution;

  CriticalPath *_cp;

  IloArray<LSMove*> _moves;
  IloInt _nbMoves;

  void realAddMove(LSMove *m) {
    if (_nbMoves < _moves.getSize()) _moves[_nbMoves] = m;
    else _moves.add(m);
    _nbMoves++;
  }

public:
  MyNHoodI(IloEnv env, CriticalPath *cp,
           const char *name = 0 )
    : IloNHoodI(env, name),
      _solution(0), _cp(cp),
      _moves(env), _nbMoves(0) {}
  ~MyNHoodI() {}
```

```
  void start(IloSolver, IloSolution soln) {
    _solution = IloSchedulerSolution(soln);
    _nbMoves = 0;
  }

  IloSolution define(IloSolver solver, IloInt i) {
    return (_moves[i])->createDelta(solver.getEnv(), _solution);
  }

  IloInt getSize(IloSolver) const { return _nbMoves; }


  void notify(IloSolver, IloInt) { resetCache(); }
  void notifyOther (IloSolver, IloSolution) { resetCache(); }
  void reset() { _solution = 0; _nbMoves = 0; resetCache(); }
};

////////////////////////////////////////////////////////////////////////////
//
// NEIGHBORHOOD #1:  The N1 neighborhood found by swapping activities
//                      on the critical path
//
////////////////////////////////////////////////////////////////////////////

class N1NHoodI : public MyNHoodI {
private:
  void addMove(IloSolver solver,
               IloResourceConstraint rc, IloResourceConstraint after) {
    SwapRC *move = new (solver.getEnv()) SwapRC(rc, after);
    MyNHoodI::realAddMove(move);
  }

public:
  N1NHoodI(IloEnv env, CriticalPath *cp,
           const char *name = 0 ) :
    MyNHoodI (env, cp, name) {}
  ~N1NHoodI() {}

  void start(IloSolver solver, IloSolution soln);
};

void N1NHoodI::start(IloSolver solver, IloSolution soln) {
  MyNHoodI::start(solver, soln);

  IloInt cpSize = 0;
  IloArray<IloResourceConstraint> cpArray = _cp->computeCP(_solution, cpSize);

  // create the moves
  for(IloInt i = 0; i < cpSize - 1; ++i) {
    IloResourceConstraint rc = cpArray[i];
    IloResourceConstraint after = cpArray[i + 1];

    if ((_solution.getNextRC(rc).getImpl() != 0) &&
        (after.getImpl() == _solution.getNextRC(rc).getImpl())) {

      addMove(solver, rc, after);
    }
  }
```

```
}


IloNHood N1NHood(IloEnv env, CriticalPath *cp,
                                                    const char *name = 0) {
  return new (env) N1NHoodI(env, cp, name);
}

////////////////////////////////////////////////////////////////////////////
//
// NEIGHBORHOOD #2:  The neighborhood found by relocating activities
//                      on the critical path to another possible resource
//                     alternative
//
////////////////////////////////////////////////////////////////////////////


class RelocateNHoodI : public MyNHoodI {
private:
  void addMove(IloSolver solver,
               IloResourceConstraint relocate, IloResource to,
               IloResourceConstraint beforeInsertion) {
    RelocateRC *move = new (solver.getEnv())
      RelocateRC(relocate, to, beforeInsertion);
    MyNHoodI::realAddMove(move);
  }

public:
  RelocateNHoodI(IloEnv env, CriticalPath *cp,
                 const char *name = 0 ) :
    MyNHoodI(env, cp, name) {}
  ~RelocateNHoodI() {}

  void  start(IloSolver solver, IloSolution soln);
};

void RelocateNHoodI::start(IloSolver solver, IloSolution soln) {
  MyNHoodI::start(solver, soln);

  IloInt cpSize = 0;
  IloArray<IloResourceConstraint> cpArray = _cp->computeCP(_solution, cpSize);

  // create the moves
  for(IloInt i = 0; i < cpSize; ++i) {
    IloResourceConstraint rc = cpArray[i];
    IloResource selected = _solution.getSelected(rc);
    if (rc.hasAltResSet()) {
      for(IloAltResSet::Iterator iter(rc.getAltResSet());
          iter.ok(); ++iter) {
        IloResource res = *iter;
        if (selected.getImpl() != res.getImpl()) {

          // add move to insert rc first on res
          addMove(solver, rc, res, 0);
          IloResourceConstraint prev = _solution.getSetupRC(res);
          while(prev.getImpl() != 0) {
            addMove(solver, rc, res, prev);
            prev = _solution.getNextRC(prev);
```

```
          }
        }
      }
    }
  }
}


IloNHood RelocateNHood(IloEnv env, CriticalPath *cp,
                       const char *name = 0) {
  return new (env) RelocateNHoodI(env, cp, name);
}


////////////////////////////////////////////////////////////////////////////
//
// TABU LIST
//
////////////////////////////////////////////////////////////////////////////

class MyTabuI : public IloMetaHeuristicI {
private:
  IloSchedulerSolution _currentSolution;
  IloInt _maxTabuSize;
  IloArray<LSMove*> _tabuList;

  IloInt _nbTabuEles;
  IloInt _nextTabuEle;

  IloNumVar _costVar;
  IloNum _bestCost;

public:
  MyTabuI(IloEnv env, IloNumVar costVar, const char *name = 0)
    : IloMetaHeuristicI(env, name)
    , _currentSolution(0)
    , _maxTabuSize(10)
    , _tabuList(env, _maxTabuSize)
    , _nbTabuEles(0)
    , _nextTabuEle(0)
    , _costVar(costVar)
    , _bestCost(IloInfinity)
  { }

  virtual IloBool start(IloSolver, IloSolution);
  virtual IloBool test(IloSolver, IloSolution);
  virtual void notify(IloSolver, IloSolution);
  virtual IloBool complete();

  virtual void reset() {
    _bestCost = IloInfinity; _nbTabuEles = 0; _nextTabuEle = 0;
  }

};

IloBool MyTabuI::start(IloSolver, IloSolution solution) {
  _currentSolution = IloSchedulerSolution(solution);
```

```
  if (solution.isObjectiveSet()) {
    IloNumVar obj = solution.getObjectiveVar();
    if (obj.getImpl() && obj.getImpl() == _costVar.getImpl()) {
      if (solution.getObjectiveValue() < _bestCost)
        _bestCost = solution.getObjectiveValue();
    }
    else
      throw "MyTabuI::start - Solution objective must be a simple variable.";
  }
  else
    throw "MyTabuI::start - Solution has no objective.";

  return IloTrue;
}


IloBool MyTabuI::test(IloSolver solver, IloSolution delta) {
  if (solver.getMin(_costVar) < _bestCost)
    return IloTrue;

  for(IloInt i = 0; i < _nbTabuEles; ++i) {
    if ((_tabuList[i] != 0) && _tabuList[i]->isReverseMove(delta))
      return IloFalse;
  }

  return IloTrue;
}

void MyTabuI::notify(IloSolver solver, IloSolution delta) {
  LSMove *move = 0;
  RelocateRC *relocateMove =
    RelocateRC::findRelocateFromDelta(solver, delta, _currentSolution);
  if (0 != relocateMove)
    move = relocateMove;
  else {
    SwapRC *swapMove = SwapRC::findSwapFromDelta(solver,
                                                 delta, _currentSolution);
    if (0 != swapMove)
      move = swapMove;
  }

  assert(move != 0);

  _tabuList[_nextTabuEle] = move;

  _nextTabuEle++;
  _nextTabuEle %= _maxTabuSize;

  if (_nbTabuEles < _maxTabuSize)
    _nbTabuEles++;
}

IloBool MyTabuI::complete() {
  // iterate through the tabu list (from oldest to youngest) removing
  // elements up to and including the oldest element that is part of
  // the neighborbood. Insert the youngest element in place of all
  // elements that are removed.
```

```
  IloInt youngest = _nextTabuEle - 1;
  if (youngest < 0)
    youngest = _nbTabuEles - 1;

  if ((youngest < 0) ||
      (_tabuList[youngest] == _tabuList[_nextTabuEle]))
    return IloTrue;

  _tabuList[_nextTabuEle] = _tabuList[youngest];

  _nextTabuEle++;
  _nextTabuEle %= _maxTabuSize;

  return IloFalse;
}

IloMetaHeuristic MyTabu(IloEnv env, IloNumVar costVar,
                        const char *name = 0)
{
  return (IloMetaHeuristicI *) new (env) MyTabuI(env, costVar, name);
}

////////////////////////////////////////////////////////////////////
//
// FIND A FIRST SOLUTION
//
////////////////////////////////////////////////////////////////////
void FindInitialSolution(IloModel model,
                         IloSchedulerSolution globalSolution,
                         IloSchedulerSolution lsSolution,
                         IloNumVar costVar) {
  IloEnv env = model.getEnv();
  IloSolver solver(model);
  IlcScheduler scheduler(solver);

  IloGoal g = IloAssignAlternative(env) &&
    IloRankForward(env) &&
    IloInstantiate(env, costVar);
  solver.startNewSearch(g);

  if(solver.next()) {
    IloNum best = solver.getValue(costVar);
    solver.out() << "Solution for Cost " <<  best << endl;
    solver.printInformation();
    globalSolution.store(scheduler);
    lsSolution.store(scheduler);
  }

  solver.endSearch();
  solver.end();
}

////////////////////////////////////////////////////////////////////
//
// SOLVE THE MODEL USING LOCAL SEARCH
//
////////////////////////////////////////////////////////////////////
IloSchedulerSolution CreateLSSolution(IloEnv env,
```

```
                                        IloSchedulerSolution globalSolution) {

  /* CREATE LOCAL SEARCH SOLUTION */
  IloSchedulerSolution lsSolution = globalSolution.makeClone(env);

  for (IloIterator <IloResourceConstraint> iter(env); iter.ok(); ++iter)
    lsSolution.setRestorable(*iter,
                             IloRestoreRCNext | IloRestoreRCSelected);

  for (IloIterator <IloResource> resIter(env); resIter.ok(); ++resIter)
    lsSolution.add(*resIter);

  return lsSolution;
}

void SolveModel(IloModel model,
                IloNumVar makespan,
                IloSchedulerSolution& globalSolution) {

  IloEnv env = model.getEnv();

  /* CREATE LOCAL SEARCH SOLUTION */
  IloSchedulerSolution lsSolution = CreateLSSolution(env, globalSolution);
  IloObjective obj = IloMinimize(env, makespan);
  lsSolution.getSolution().add(obj);

  globalSolution.getSolution().add(makespan);

  /* GENERATE AN INITIAL SOLUTION. */
  FindInitialSolution(model, globalSolution, lsSolution, makespan);
  IloNum best = globalSolution.getMax(makespan);
  env.out() << "Initial solution:" << endl;
  PrintSolution(env, globalSolution, makespan);

  /* SET PARAMETERS FOR LOCAL SEARCH */
  IloSchedulerEnv schedEnv(env);
  schedEnv.setPrecedenceEnforcement(IloHigh);
  schedEnv.getResourceParam().ignoreCapacityConstraints();

  IloSolver lsSolver(model);
  IlcScheduler lsScheduler(lsSolver);

  /* HILL CLIMB */
  CriticalPath cp(env);
  IloNHood nhood = N1NHood(env, &cp) + RelocateNHood(env, &cp);
  IloGoal greedyMove = IloSingleMove(env,
                                     lsSolution,
                                     nhood,
                                     IloImprove(env),
                                     IloFirstSolution(env),
                                     IloInstantiate(env, makespan));

  IloInt maxIter = 100;
  IloInt movesDone = 0;
  while((movesDone < maxIter) && lsSolver.solve(greedyMove)) {
    IloNum cost = lsSolution.getSolution().getObjectiveValue();
    lsSolver.out() << "Move: " << movesDone << ":\t";
```

```
    ++movesDone;
    lsSolver.out() << "solution at cost: " << cost << " ** HC" << endl;
    best = cost;
    globalSolution.store(lsScheduler);
  }

  /* TABU */
  IloMetaHeuristic mh = MyTabu(env, makespan);
  IloGoal move = IloSingleMove(env,
                               lsSolution,
                               nhood,
                               mh,
                               IloMinimizeVar(env, makespan),
                               IloInstantiate(env, makespan));

  for(IloInt i = movesDone; i < maxIter; ++i) {
    lsSolver.out() << "Move: " << i << ":\t";
    if (!lsSolver.solve(move)) {
      lsSolver.out() << "no solution" << endl;
      if ((nhood.getSize(lsSolver) == 0) || mh.complete())
        break;
    }
    else {
      IloNum cost = lsSolution.getSolution().getObjectiveValue();
      lsSolver.out() << "solution at cost: " << cost;
      if (cost < best) {
        globalSolution.store(lsScheduler);
        best = cost;
        lsSolver.out() << " **";
      }
      lsSolver.out() << endl;
    }
  }

  env.out() << "Final solution Cost: " << best << endl;

  PrintSolution(env, globalSolution, makespan);
  lsSolver.end();
}

////////////////////////////////////////////////////////////////////
//
// DEFINE THE MODEL WITH ALTERNATIVE RESOURCES
//
////////////////////////////////////////////////////////////////////
IloModel
DefineModel(IloEnv& env,
            IloInt numberOfJobs,
            IloInt numberOfResources,
            IloInt* resourceNumbers,
            IloNum* durations,
            IloRandom randomGenerator,
            IloSchedulerSolution solution,
            IloNumVar& makespan)
{
  IloModel model(env);

  /* CREATE THE MAKESPAN VARIABLE. */
```

```
IloInt numberOfActivities = numberOfJobs * numberOfResources;
IloNum horizon = 0;
IloInt k;

for (k = 0; k < numberOfActivities; k++)
  horizon += durations[k];

makespan = IloNumVar(env, 0, IloIntMax/2, ILOINT);

/* CREATE THE RESOURCES. */
IloSchedulerEnv schedEnv(env);
IloResourceParam resParam = schedEnv.getResourceParam();
resParam.setPrecedenceEnforcement(IloMediumHigh);

char buffer[128];
IloInt j;
IloUnaryResource *resources =
  new (env) IloUnaryResource[numberOfResources];
for (j = 0; j < numberOfResources; j++) {
  sprintf(buffer, "R%ld", j);
  resources[j] = IloUnaryResource(env, buffer);
}

/* CREATE THE ALTERNATIVE RESOURCE SETS */
env.out() << "Creating resource sets" << endl;
IloInt *altResourceNumbers = new (env) IloInt[numberOfResources];
IloAltResSet *altResSets =
  new (env) IloAltResSet[numberOfResources];
for (j = 0; j < numberOfResources; j++) {
  altResSets[j] = IloAltResSet(env);
  altResSets[j].add(resources[j]);

  // RANDOMLY PICK ANOTHER RESOURCE TO BE IN THE SET
  assert(numberOfResources > 1);
  IloInt index = randomGenerator.getInt(numberOfResources);
  while(index == j)
    index = randomGenerator.getInt(numberOfResources);

  altResSets[j].add(resources[index]);
  altResourceNumbers[j] = index;
  env.out() << "Set #" << j << ":\t" << resources[j].getName()
            << " " << resources[index].getName() << endl;
}

/* CREATE THE ALTERNATIVE DURATIONS */
IloNum *altDurations = new (env) IloNum[numberOfActivities];
for(k = 0; k < numberOfActivities; k++) {
  IloNum multiplier = 1.0 + (randomGenerator.getFloat() / 2.0);
  altDurations[k] = IloCeil(multiplier * durations[k]);
}

/* CREATE THE ACTIVITIES. */
env.out() << "Setting alternative processing times" << endl;
k = 0;
IloInt i;
for (i = 0; i < numberOfJobs; i++) {
  IloActivity previousActivity;
  for (j = 0; j < numberOfResources; j++) {
```

```
      IloNum ptMin = IloMin(durations[k], altDurations[k]);
      IloNum ptMax = IloMax(durations[k], altDurations[k]);
      IloNumVar ptVar(env, ptMin, ptMax, ILOINT);

      IloActivity activity(env, ptVar);

      sprintf(buffer, "J%ldS%ldR%ld", i, j, resourceNumbers[k]);
      activity.setName(buffer);
      solution.add(activity, IloRestoreNothing);

      IloResourceConstraint rc =
        activity.requires(altResSets[resourceNumbers[k]]);

      // SET THE DIFFERENT DURATIONS DEPENDING ON
      // RESOURCE SELECTION
      rc.setProcessingTimeMax(resources[resourceNumbers[k]],
                              durations[k]);
      rc.setProcessingTimeMin(resources[resourceNumbers[k]],
                              durations[k]);

      rc.setProcessingTimeMax(
             resources[altResourceNumbers[resourceNumbers[k]]],
             altDurations[k]);
      rc.setProcessingTimeMin(
             resources[altResourceNumbers[resourceNumbers[k]]],
             altDurations[k]);

      model.add(rc);

      solution.add(rc, IloRestoreNothing);
      env.out() << activity.getName()
                << ":\tProcessing Time("
                << resources[resourceNumbers[k]].getName()
                << "): " << durations[k]
                << "\n\tProcessing Time("
                << resources[altResourceNumbers[
                                    resourceNumbers[k]]].getName()
                << "): " << altDurations[k]
                << endl;


      if (j != 0)
        model.add(activity.startsAfterEnd(previousActivity));
      previousActivity = activity;
      k++;
    }
    model.add(previousActivity.endsBefore(makespan));
  }

  /* RETURN THE MODEL. */
  return model;
}


////////////////////////////////////////////////////////////
//
// INITIALIZE THE PROGRAM ARGUMENTS
```

```
//
////////////////////////////////////////////////////////////////
void
InitParameters(int argc,
               char** argv,
               IloInt& numberOfJobs,
               IloInt& numberOfResources,
               IloInt*& resourceNumbers,
               IloNum*& durations)
{
  numberOfJobs = 6;
  numberOfResources = 6;
  resourceNumbers = ResourceNumbers06;
  durations = Durations06;

  if (argc > 1) {
    IloInt number = atol(argv[1]);
    if (number == 10) {
      numberOfJobs = 10;
      numberOfResources = 10;
      resourceNumbers = ResourceNumbers10;
      durations = Durations10;
    }
    else if (number == 20) {
      numberOfJobs = 20;
      numberOfResources = 5;
      resourceNumbers = ResourceNumbers20;
      durations = Durations20;
    }
  }
}

////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////

int main(int argc, char** argv)
{
  IloInt numberOfJobs;
  IloInt numberOfResources;
  IloInt* resourceNumbers;
  IloNum* durations;

  InitParameters(argc,
                 argv,
                 numberOfJobs,
                 numberOfResources,
                 resourceNumbers,
                 durations);

  try {

    IloEnv env;
    IloNumVar makespan;
    IloRandom randGen(env, 8975324);
    IloSchedulerSolution solution(env);
```

```
    IloModel model = DefineModel(env,
                                 numberOfJobs,
                                 numberOfResources,
                                 resourceNumbers,
                                 durations,
                                 randGen,
                                 solution,
                                 makespan);

    SolveModel(model, makespan, solution);
    env.end();
  }
  catch (IloSchedulerException& exc) {
    cout << exc << endl;
  }
  catch (IloException& exc) {
    cout << exc << endl;
  }

  return 0;
}
```

C H A P T E R

29

Large Neighborhood Search for the Jobshop Problem with Alternatives

In Chapter 28, *Tabu Search for the Jobshop Problem with Alternatives*, we described a local search algorithm to solve a problem where two types of decisions must be made: the resource assigned to each activity, and the sequence of activities on each resource. In this section, we consider another approach based on large neighborhood search. Three different large neighborhoods are applied. Two are pre-defined neighborhoods, and the other one is a dedicated neighborhood for the jobshop scheduling problem.

Problem Description

The problem used in this chapter is once again the standard job shop scheduling problem, extended so that each activity requires one of a set of two resources rather than having a predefined resource requirement.

Define the Problem

Compared to the definition of the problem used with tabu search, the main difference here is the extraction of the underlying structure of the problem: the function `DefineModel` returns the set of jobs that are created while building the scheduling model.

```
IloModel
DefineModel(IloEnv& env,
            IloInt numberOfJobs,
            IloInt numberOfResources,
            IloInt* resourceNumbers,
            IloNum* durations,
            IloRandom randomGenerator,
            IloSchedulerSolution solution,
            IloArray<IloActivityArray>& jobs,
            IloNumVar& makespan)
```

Solving the Problem

To solve the problem, the following large neighborhoods are applied:

- `IloRelocateActivityNHood`
- `TimeWindowNHood`
- `RelocateJobNHood`

When applying these neighborhoods, only a subset of the decisions that are currently in the schedule solution is restored. These decisions are the start times, the end times, the durations and the processing times of the activities, and for the resource constraints, the previous, the setup, the teardown, the successors, the predecessors, the resource assignment and the capacity.

To turn this partial schedule to a complete solution, one must apply a search goal that is focused on a subpart of the problem. Notice that precedence information such as next, previous, setup, teardown, successors and predecessors are stored in the scheduler solution due to the use of the precedence graph.

The first two neighborhoods, `IloRelocateActivityNHood` and `TimeWindowNHood`, are predefined. `RelocateJobNHood` is a new neighborhood that is similar to the neighborhood `IloRelocateActivityNHood`, but instead of reconsidering the decisions related to one activity at a time, it reconsiders all the decisions related to a job.

Neighborhood IloRelocateActivityNHood

The size of this neighborhood is the number of activities of the problem.

For a given index, the set of selected scheduling objects is made of an single activity and all its associated resource constraints. The default behavior of this neighborhood is to restore the successors, the predecessors, the resource assignment and the capacity required of all resource constraints except the ones related to the activity selected: in other words, the setup, the teardown, the next and previous are not restored. Regarding the resource constraints of the selected activity, none of the decisions that are stored in the scheduler solution is restored. Lastly, the start times, end times, durations and the processing times of the activities are not restored.

The following line creates an instance of `IloRelocateActivityNHood` with the default behavior described above.

```
IloSchedulerLargeNHood relocateActivities = IloRelocateActivityNHood(env);
```

Neighborhood IloTimeWindowNHood

The size of this neighborhood depends on two parameters: the `windowSize` and the `windowStep`. The parameter `windowSize` specifies the number of consecutive activities in a given time window. The parameter `windowStep` is the number of activities to skip to move to the next time window. The size of this neighborhood can be computed using the following formula:

size = [nbActivities - (windowSize - windowStep)] / windowStep

For instance, on the 10x10 instance, with a `windowSize` of 20 and a `windowStep` of 10, the number of time windows is 9.

For a given index, the set of selected scheduling objects is made of all the activities belonging to the corresponding time window together with their associated resource constraints. The default behavior of this neighborhood is to restore all precedence information of all resource constraints except the ones related to the selected activities (that is, the next, the prev, the setup, the teardown, the successors, the predecessors, the resource assignment and the capacity required). Regarding the resource constraints of the selected activities, none of the decisions that are stored in the scheduler solution is restored. Lastly, the start times, the end times, the durations and the processing times of the activities are not restored.

The following line creates an instance of `IloTimeWindowNHood` with the default behavior described above:

```
IloSchedulerLargeNHood timeWindowNHood =
  IloTimeWindowNHood(env, 20, 10);
```

Defining a new scheduler large neighborhood: RelocateJobNHood

We define a special neighborhood that can be seen as an extension of the neighborhood `IloRelocateActivityNHood` but instead of selecting one activity we select all the activities belonging to a job: the size of this neighborhood is the number of jobs in the problem. For this new neighborhood, we will use the same default behavior as `IloRelocateActivityNHood`.

New class RelocateJobNHoodI

To define a scheduler large neighborhood, the first step is to define a sub-class of the abstract class `IloSchedulerLargeNHooI` and to define the three virtual member functions: `start`, `getSize`, `defineSelected`:

```
class RelocateJobNHoodI : public IloSchedulerLargeNHoodI {
protected:
  IloArray<IloActivityArray> _jobs;

public:
  RelocateJobNHoodI(IloEnv env,
                    IloArray<IloActivityArray> jobs,
                    const char* name);
  ~RelocateJobNHoodI();
  // virtuals
  virtual IloInt getSize(IloSolver solver) const;
  virtual IloSolution defineSelected(IloSolver solver, IloInt index);
  virtual void start(IloSolver solver, IloSolution solution);
};
```

Constructor of class RelocateJobNHoodI

The constructor of this neighborhood takes as a parameter an array of jobs where a job is simply a set of activities. The default behavior of this neighborhood is to restore the successors, the predecessors the resource assignment and the required capacity of all resource constraints but the one selected. This default behavior is specified by using predicates.

The predicates are shown in the following code.

```
ILOPREDICATE0(IloRCFalsePredicate,
              IloResourceConstraint, rc) {
  return IloFalse;
}

ILOCTXPREDICATE0(IloRCTrueIfNotSelectedPredicate,
                 IloResourceConstraint, rc,
                 IloSchedulerLargeNHood, nhood) {
  if (nhood.isSelected(rc))
    return IloFalse;
  else
    return IloTrue;
}
```

```
ILOPREDICATE0(IloActivityFalsePredicate,
              IloActivity, activity) {
  return IloFalse;
}

RelocateJobNHoodI::RelocateJobNHoodI(IloEnv env,
                                     IloArray<IloActivityArray> jobs,
                                     const char* name)
  : IloSchedulerLargeNHoodI(env, name),
    _jobs(env)
{
  // set default restore policy
  IloPredicate<IloResourceConstraint> rcFalsePredicate =
    IloRCFalsePredicate(env);
  setRestoreRCNextPredicate(rcFalsePredicate);
  setRestoreRCPrevPredicate(rcFalsePredicate);
  setRestoreRCSetupPredicate(rcFalsePredicate);
  setRestoreRCTeardownPredicate(rcFalsePredicate);
  IloPredicate<IloResourceConstraint> rcTrueIfNotSelectedPredicate =
    IloRCTrueIfNotSelectedPredicate(env);
  setRestoreRCDirectSuccessorPredicate(rcTrueIfNotSelectedPredicate);
  setRestoreRCDirectPredecessorPredicate(rcTrueIfNotSelectedPredicate);
  setRestoreRCCapacityPredicate(rcTrueIfNotSelectedPredicate);
  setRestoreRCSelectedPredicate(rcTrueIfNotSelectedPredicate);

  IloPredicate<IloActivity> activityFalsePredicate =
    IloActivityFalsePredicate(env);
  setRestoreActivityStartPredicate(activityFalsePredicate);
  setRestoreActivityEndPredicate(activityFalsePredicate);
  setRestoreActivityDurationPredicate(activityFalsePredicate);
  setRestoreActivityProcessingTimePredicate(activityFalsePredicate);
  setRestoreActivityExternalPredicate(activityFalsePredicate);

  for (IloInt j=0; j<jobs.getSize(); ++j) {
    IloInt nbActivities = jobs[j].getSize();
    IloActivityArray job(env);
    for (IloInt a=0; a<nbActivities; ++a) {
      IloActivity activity = jobs[j][a];
      job.add(activity);
    }
    _jobs.add(job);
  }
}
```

Overriding virtual method start

The method start simply calls the corresponding method of its parent class: this allows registration the scheduler solution `solution` as the current solution.

```
void
RelocateJobNHoodI::start(IloSolver solver, IloSolution solution)
{
  IloSchedulerLargeNHoodI::start(solver, solution);
}
```

Overriding virtual method getSize

The size of this neighborhood is the simply the number of jobs in the jobshop problem.

```
IloInt
RelocateJobNHoodI::getSize(IloSolver solver) const
{
  IloInt size = _jobs.getSize();

  return size;
}
```

Overriding virtual method defineSelected

The role of this method is to define a scheduler solution that represents the set of scheduler objects that are selected for the given index. Here the index of the neighborhood is simply used as the index in the array of jobs: all activities and all resource constraints corresponding to that job are added to the set.

```
IloSolution
RelocateJobNHoodI::defineSelected(IloSolver solver, IloInt index)
{
  IloEnv env = solver.getEnv();

  IloActivityArray job;
  job = _jobs[index];

  IloSchedulerSolution selected(env);

  IloSchedulerSolution currentSolution = getCurrentSolution();

  // add in selected all activities of selected job and their
  // resource constraints
  IloInt nbActivities = job.getSize();
  for (IloInt a=0; a<nbActivities; a++) {
    IloActivity activity = job[a];
    selected.add(activity);
    IloSchedulerSolution::ResourceConstraintIterator
      rcIter(currentSolution, activity);
    for (; rcIter.ok(); ++rcIter) {
      IloResourceConstraint rc = *rcIter;
      selected.add(rc);
    }
  }

  return selected;
}
```

Defining a Wrapper

The last step consists in writing a function that wraps an instance of `RelocateJobNHoodI` in a `IloSchedulerLargeNHood` handle.

```
IloSchedulerLargeNHood
RelocateJobNHood(IloEnv env,
                 IloArray<IloActivityArray> jobs,
                 const char* name = 0)
{
  return new (env) RelocateJobNHoodI(env,
                                     jobs,
                                     name);
}
```

The following line creates an instance of `RelocateJobNHood` with the default behavior described above.

```
IloSchedulerLargeNHood relocateJobNHood = RelocateJobNHood(env, jobs);
```

Solving the Subproblem

As in a large neighborhood search approach, only a subpart of the decisions stored in the current solution is restored, we need to a apply a search goal to transform the partial schedule into a complete solution. Notice that the more decisions are restored, the lower the size of the search tree is and also the lower the number of solutions that can be reached. Clearly there is trade-off between the complexity of the subproblem and the number of solutions that can be reached. Notice also, that although the size of the search tree is generally much smaller, the complexity of the subproblem can still be very high: again there is a trade-off between exploring a large part of the search tree and few neighbors and exploring a small part of the search tree and many neighbors.

To solve the subproblem, we choose to apply a combination of the predefined goals `IloAssignAlternatives` and `IloRankForward`.

```
IloGoal subGoal =
  IloAssignAlternative(env) &&
  IloRankForward(env) &&
  IloInstantiate(env, makespan);
```

We also choose to apply an incomplete search by limiting the number of failures while exploring the search tree of the subproblem. In case a solution is found before this limit, we

choose to let the search continue to find an even better solution until this limit is reached, as follows.

```
IloInt failLimit = 2000;

if (failLimit < IloIntMax) {
  subGoal = IloLimitSearch(env, subGoal, IloFailLimit(env, failLimit));
}
```

Complete Program

You can see the entire program `altlns.cpp` here or view it online in the standard distribution.

```
#include <ilsched/ilolnsgoals.h>
#include <ilsolver/iimls.h>

ILOSTLBEGIN

IloInt ResourceNumbers06 [] = {2, 0, 1, 3, 5, 4,
                               1, 2, 4, 5, 0, 3,
                               2, 3, 5, 0, 1, 4,
                               1, 0, 2, 3, 4, 5,
                               2, 1, 4, 5, 0, 3,
                               1, 3, 5, 0, 4, 2};

IloNum Durations06 [] = { 1,  3,  6,  7,  3,  6,
                          8,  5, 10, 10, 10,  4,
                          5,  4,  8,  9,  1,  7,
                          5,  5,  5,  3,  8,  9,
                          9,  3,  5,  4,  3,  1,
                          3,  3,  9, 10,  4,  1};

IloInt ResourceNumbers10 [] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
                               0, 2, 4, 9, 3, 1, 6, 5, 7, 8,
                               1, 0, 3, 2, 8, 5, 7, 6, 9, 4,
                               1, 2, 0, 4, 6, 8, 7, 3, 9, 5,
                               2, 0, 1, 5, 3, 4, 8, 7, 9, 6,
                               2, 1, 5, 3, 8, 9, 0, 6, 4, 7,
                               1, 0, 3, 2, 6, 5, 9, 8, 7, 4,
                               2, 0, 1, 5, 4, 6, 8, 9, 7, 3,
                               0, 1, 3, 5, 2, 9, 6, 7, 4, 8,
                               1, 0, 2, 6, 8, 9, 5, 3, 4, 7};

IloNum Durations10 [] = {29, 78,  9, 36, 49, 11, 62, 56, 44, 21,
                         43, 90, 75, 11, 69, 28, 46, 46, 72, 30,
                         91, 85, 39, 74, 90, 10, 12, 89, 45, 33,
                         81, 95, 71, 99,  9, 52, 85, 98, 22, 43,
                         14,  6, 22, 61, 26, 69, 21, 49, 72, 53,
                         84,  2, 52, 95, 48, 72, 47, 65,  6, 25,
                         46, 37, 61, 13, 32, 21, 32, 89, 30, 55,
                         31, 86, 46, 74, 32, 88, 19, 48, 36, 79,
                         76, 69, 76, 51, 85, 11, 40, 89, 26, 74,
```

```
                          85, 13, 61,  7, 64, 76, 47, 52, 90, 45};

IloInt ResourceNumbers20 [] = {0, 1, 2, 3, 4,
                               0, 1, 3, 2, 4,
                               1, 0, 2, 4, 3,
                               1, 0, 4, 2, 3,
                               2, 1, 0, 3, 4,
                               2, 1, 4, 0, 3,
                               1, 0, 2, 3, 4,
                               2, 1, 0, 3, 4,
                               0, 3, 2, 1, 4,
                               1, 2, 0, 3, 4,
                               1, 3, 0, 4, 2,
                               2, 0, 1, 3, 4,
                               0, 2, 1, 3, 4,
                               2, 0, 1, 3, 4,
                               0, 1, 4, 2, 3,
                               1, 0, 3, 4, 2,
                               0, 2, 1, 3, 4,
                               0, 1, 4, 2, 3,
                               1, 2, 0, 3, 4,
                               0, 1, 2, 3, 4};

IloNum Durations20 [] = {29,  9, 49, 62, 44,
                         43, 75, 69, 46, 72,
                         91, 39, 90, 12, 45,
                         81, 71,  9, 85, 22,
                         14, 22, 26, 21, 72,
                         84, 52, 48, 47,  6,
                         46, 61, 32, 32, 30,
                         31, 46, 32, 19, 36,
                         76, 76, 85, 40, 26,
                         85, 61, 64, 47, 90,
                         78, 36, 11, 56, 21,
                         90, 11, 28, 46, 30,
                         85, 74, 10, 89, 33,
                         95, 99, 52, 98, 43,
                          6, 61, 69, 49, 53,
                          2, 95, 72, 65, 25,
                         37, 13, 21, 89, 55,
                         86, 74, 88, 48, 79,
                         69, 51, 11, 89, 74,
                         13,  7, 76, 52, 45};


////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////

void PrintRange(IloEnv& env, IloNum min, IloNum max) {
  if (min == max)
    env.out() << (IlcInt)min;
  else
    env.out() << (IlcInt)min << ".." << (IlcInt)max;
}
```

```
void PrintSolution(IloEnv& env,
                   const IloSchedulerSolution solution,
                   const IloNumVar makespan)
{
  if (solution.contains(makespan)) {
    env.out() << "Solution with makespan ["
      << solution.getMin(makespan) << ".."
      << solution.getMax(makespan) << "]" << endl;
  }

  for (IloSchedulerSolution::ResourceConstraintIterator
                                          iter(solution);
       iter.ok();
       ++iter)
  {
    IloResourceConstraint rc = *iter;
    if (!solution.isResourceSelected(rc))
      IloSchedulerException("No resource assigned!");

    IloActivity activity = rc.getActivity();
    env.out() << activity.getName() << "[";
    PrintRange(env,
               solution.getStartMin(activity),
               solution.getStartMax(activity));
    env.out() << " -- ";
    PrintRange(env,
               solution.getDurationMin(activity),
               solution.getDurationMax(activity));
    env.out() << " --> ";
    PrintRange(env,
               solution.getEndMin(activity),
               solution.getEndMax(activity));
    env.out() << "]: " << solution.getSelected(rc).getName()
              << endl;
  }
}

////////////////////////////////////////////////////////////////////
//
// FIND A FIRST SOLUTION
//
////////////////////////////////////////////////////////////////////

void FindInitialSolution(IloModel model,
                         IloSchedulerSolution globalSolution,
                         IloSchedulerSolution lsSolution,
                         IloNumVar costVar) {
  IloEnv env = model.getEnv();
  IloSolver solver(model);
  IlcScheduler scheduler(solver);

  IloGoal g = IloAssignAlternative(env) &&
    IloRankForward(env) &&
    IloInstantiate(env, costVar);

  solver.startNewSearch(g);
```

```
  if(solver.next()) {
    IloNum best = solver.getValue(costVar);
    solver.out() << "Solution for Cost " <<  best << endl;
    solver.printInformation();
    globalSolution.store(scheduler);
    lsSolution.store(scheduler);
  }

  solver.endSearch();
  solver.end();
}

////////////////////////////////////////////////////////////////////
//
// DEFINE A NEW NEIGHBORHOOD : RELOCATE JOB
//
////////////////////////////////////////////////////////////////////

class RelocateJobNHoodI : public IloSchedulerLargeNHoodI {
protected:
  IloArray<IloActivityArray> _jobs;

public:
  RelocateJobNHoodI(IloEnv env,
                    IloArray<IloActivityArray> jobs,
                    const char* name);
  ~RelocateJobNHoodI();
  // virtuals
  virtual IloInt getSize(IloSolver solver) const;
  virtual IloSolution defineSelected(IloSolver solver, IloInt index);
  virtual void start(IloSolver solver, IloSolution solution);
};

ILOPREDICATE0(IloRCFalsePredicate,
              IloResourceConstraint, rc) {
  return IloFalse;
}

ILOCTXPREDICATE0(IloRCTrueIfNotSelectedPredicate,
                 IloResourceConstraint, rc,
                 IloSchedulerLargeNHood, nhood) {
  if (nhood.isSelected(rc))
    return IloFalse;
  else
    return IloTrue;
}

ILOPREDICATE0(IloActivityFalsePredicate,
               IloActivity, activity) {
  return IloFalse;
}

RelocateJobNHoodI::RelocateJobNHoodI(IloEnv env,
                                     IloArray<IloActivityArray> jobs,
                                     const char* name)
  : IloSchedulerLargeNHoodI(env, name),
    _jobs(env)
{
```

```
  // set default restore policy
  IloPredicate<IloResourceConstraint> rcFalsePredicate =
    IloRCFalsePredicate(env);
  setRestoreRCNextPredicate(rcFalsePredicate);
  setRestoreRCPrevPredicate(rcFalsePredicate);
  setRestoreRCSetupPredicate(rcFalsePredicate);
  setRestoreRCTeardownPredicate(rcFalsePredicate);
  IloPredicate<IloResourceConstraint> rcTrueIfNotSelectedPredicate =
    IloRCTrueIfNotSelectedPredicate(env);
  setRestoreRCDirectSuccessorPredicate(rcTrueIfNotSelectedPredicate);
  setRestoreRCDirectPredecessorPredicate(rcTrueIfNotSelectedPredicate);
  setRestoreRCCapacityPredicate(rcTrueIfNotSelectedPredicate);
  setRestoreRCSelectedPredicate(rcTrueIfNotSelectedPredicate);

  IloPredicate<IloActivity> activityFalsePredicate =
    IloActivityFalsePredicate(env);
  setRestoreActivityStartPredicate(activityFalsePredicate);
  setRestoreActivityEndPredicate(activityFalsePredicate);
  setRestoreActivityDurationPredicate(activityFalsePredicate);
  setRestoreActivityProcessingTimePredicate(activityFalsePredicate);
  setRestoreActivityExternalPredicate(activityFalsePredicate);

  for (IloInt j=0; j<jobs.getSize(); ++j) {
    IloInt nbActivities = jobs[j].getSize();
    IloActivityArray job(env);
    for (IloInt a=0; a<nbActivities; ++a) {
      IloActivity activity = jobs[j][a];
      job.add(activity);
    }
    _jobs.add(job);
  }
}

RelocateJobNHoodI::~RelocateJobNHoodI()
{
  _jobs.end();
}

void
RelocateJobNHoodI::start(IloSolver solver, IloSolution solution)
{
  IloSchedulerLargeNHoodI::start(solver, solution);
}

IloInt
RelocateJobNHoodI::getSize(IloSolver solver) const
{
  IloInt size = _jobs.getSize();

  return size;
}

IloSolution
RelocateJobNHoodI::defineSelected(IloSolver solver, IloInt index)
{
  IloEnv env = solver.getEnv();

  IloActivityArray job;
```

```
  job = _jobs[index];

  IloSchedulerSolution selected(env);

  IloSchedulerSolution currentSolution = getCurrentSolution();

  // add in selected all activities of selected job and their
  // resource constraints
  IloInt nbActivities = job.getSize();
  for (IloInt a=0; a<nbActivities; a++) {
    IloActivity activity = job[a];
    selected.add(activity);
    IloSchedulerSolution::ResourceConstraintIterator
      rcIter(currentSolution, activity);
    for (; rcIter.ok(); ++rcIter) {
      IloResourceConstraint rc = *rcIter;
      selected.add(rc);
    }
  }

  return selected;
}

IloSchedulerLargeNHood
RelocateJobNHood(IloEnv env,
                 IloArray<IloActivityArray> jobs,
                 const char* name = 0)
{
  return new (env) RelocateJobNHoodI(env,
                                     jobs,
                                     name);
}


IloSchedulerSolution CreateLSSolution(IloEnv env,
                                      IloSchedulerSolution globalSolution) {


  /* CREATE LOCAL SEARCH SOLUTION */
  IloSchedulerSolution lsSolution = globalSolution.makeClone(env);

  for (IloIterator <IloResourceConstraint> iter(env); iter.ok(); ++iter)
    lsSolution.setRestorable(*iter,
                             IloRestoreRCNext | IloRestoreRCSelected);

  for (IloIterator <IloResource> resIter(env); resIter.ok(); ++resIter)
    lsSolution.add(*resIter);

  return lsSolution;
}

////////////////////////////////////////////////////////////////////
//
// SOLVE THE MODEL USING A GREEDY DESCENT SEARCH
//
////////////////////////////////////////////////////////////////////

void SolveModel(IloModel model,
```

```
                IloNumVar makespan,
                IloArray<IloActivityArray> jobs,
                IloSchedulerSolution& globalSolution) {

IloEnv env = model.getEnv();

/* CREATE LOCAL SEARCH SOLUTION */
IloSchedulerSolution lsSolution = CreateLSSolution(env, globalSolution);
IloObjective obj = IloMinimize(env, makespan);
lsSolution.getSolution().add(obj);

globalSolution.getSolution().add(makespan);

/* GENERATE AN INITIAL SOLUTION. */
FindInitialSolution(model, globalSolution, lsSolution, makespan);
IloNum best = globalSolution.getMax(makespan);
env.out() << "Initial solution:" << endl;
PrintSolution(env, globalSolution, makespan);

/* SET PARAMETERS FOR LOCAL SEARCH */
IloSolver lsSolver(model);
IlcScheduler lsScheduler(lsSolver);

/* GLIDING TIME WINDOW SEARCH */

IloGoal subGoal =
  IloAssignAlternative(env) &&
  IloRankForward(env) &&
  IloInstantiate(env, makespan);

IloInt failLimit = 2000;

if (failLimit < IloIntMax) {
  subGoal = IloLimitSearch(env, subGoal, IloFailLimit(env, failLimit));
}

subGoal = IloSelectSearch(env, subGoal, IloMinimizeVar(env, makespan, 1.0));

IloSchedulerLargeNHood relocateActivities = IloRelocateActivityNHood(env);

IloSchedulerLargeNHood timeWindowNHood =
  IloTimeWindowNHood(env, 20, 10);

IloSchedulerLargeNHood relocateJobNHood = RelocateJobNHood(env, jobs);

IloNHood nhood =
  IloContinue(env, relocateActivities)
  +
  IloContinue(env, relocateJobNHood)
  +
  IloContinue(env, timeWindowNHood)
  ;


IloGoal greedyMove = IloSingleMove(env,
                                   lsSolution,
                                   nhood,
                                   IloImprove(env),
```

```
                                    IloFirstSolution(env),
                                    subGoal);
  IloInt movesDone = 0;
  while(lsSolver.solve(greedyMove)) {
    IloNum cost = lsSolution.getSolution().getObjectiveValue();
    lsSolver.out() << "Move: " << movesDone << ":\t";
    ++movesDone;
    lsSolver.out() << "solution at cost: " << cost << " ** HC" << endl;
    best = cost;
    globalSolution.store(lsScheduler);
  }

  env.out() << "Final solution Cost: " << best << endl;

  PrintSolution(env, globalSolution, makespan);

  lsSolver.end();
}

////////////////////////////////////////////////////////////////////
//
// DEFINE THE MODEL WITH ALTERNATIVE RESOURCES
//
////////////////////////////////////////////////////////////////////

IloModel
DefineModel(IloEnv& env,
            IloInt numberOfJobs,
            IloInt numberOfResources,
            IloInt* resourceNumbers,
            IloNum* durations,
            IloRandom randomGenerator,
            IloSchedulerSolution solution,
            IloArray<IloActivityArray>& jobs,
            IloNumVar& makespan)
{
  IloModel model(env);

  /* CREATE THE MAKESPAN VARIABLE. */
  IloInt numberOfActivities = numberOfJobs * numberOfResources;
  IloNum horizon = 0;
  IloInt k;

  for (k = 0; k < numberOfActivities; k++)
    horizon += durations[k];

  makespan = IloNumVar(env, 0, IloIntMax/2, ILOINT);

  /* CREATE THE RESOURCES. */
  IloSchedulerEnv schedEnv(env);
  IloResourceParam resParam = schedEnv.getResourceParam();
  resParam.setCapacityEnforcement(IloMediumLow);

  char buffer[128];
  IloInt j;
  IloUnaryResource *resources =
    new (env) IloUnaryResource[numberOfResources];
  for (j = 0; j < numberOfResources; j++) {
```

```
  sprintf(buffer, "R%ld", j);
  resources[j] = IloUnaryResource(env, buffer);
}

/* CREATE THE ALTERNATIVE RESOURCE SETS */
env.out() << "Creating resource sets" << endl;
IloInt *altResourceNumbers = new (env) IloInt[numberOfResources];
IloAltResSet *altResSets =
  new (env) IloAltResSet[numberOfResources];
for (j = 0; j < numberOfResources; j++) {
  altResSets[j] = IloAltResSet(env);
  altResSets[j].add(resources[j]);

  // RANDOMLY PICK ANOTHER RESOURCE TO BE IN THE SET
  assert(numberOfResources > 1);
  IloInt index = randomGenerator.getInt(numberOfResources);
  while(index == j)
    index = randomGenerator.getInt(numberOfResources);

  altResSets[j].add(resources[index]);
  altResourceNumbers[j] = index;
  env.out() << "Set #" << j << ":\t" << resources[j].getName()
            << " " << resources[index].getName() << endl;
}

/* CREATE THE ALTERNATIVE DURATIONS */
IloNum *altDurations = new (env) IloNum[numberOfActivities];
for(k = 0; k < numberOfActivities; k++) {
  IloNum multiplier = 1.0 + (randomGenerator.getFloat() / 2.0);
  altDurations[k] = IloCeil(multiplier * durations[k]);
}

/* CREATE THE ACTIVITIES. */
env.out() << "Setting alternative processing times" << endl;
k = 0;
IloInt i;
jobs = IloArray<IloActivityArray>(env);
for (i = 0; i < numberOfJobs; i++) {
  IloActivityArray job(env);
  jobs.add(job);
  IloActivity previousActivity;
  for (j = 0; j < numberOfResources; j++) {

    IloNum ptMin = IloMin(durations[k], altDurations[k]);
    IloNum ptMax = IloMax(durations[k], altDurations[k]);
    IloNumVar ptVar(env, ptMin, ptMax, ILOINT);

    IloActivity activity(env, ptVar);
    job.add(activity);

    sprintf(buffer, "J%ldS%ldR%ld", i, j, resourceNumbers[k]);
    activity.setName(buffer);
    solution.add(activity, IloRestoreNothing);

    IloResourceConstraint rc =
      activity.requires(altResSets[resourceNumbers[k]]);

    // SET THE DIFFERENT DURATIONS DEPENDING ON
```

```
      // RESOURCE SELECTION
      rc.setProcessingTimeMax(resources[resourceNumbers[k]],
                              durations[k]);
      rc.setProcessingTimeMin(resources[resourceNumbers[k]],
                              durations[k]);

      rc.setProcessingTimeMax(
             resources[altResourceNumbers[resourceNumbers[k]]],
             altDurations[k]);
      rc.setProcessingTimeMin(
             resources[altResourceNumbers[resourceNumbers[k]]],
             altDurations[k]);

      model.add(rc);

      solution.add(rc, IloRestoreNothing);
      env.out() << activity.getName()
                << ":\tProcessing Time("
                << resources[resourceNumbers[k]].getName()
                << "): " << durations[k]
                << "\n\tProcessing Time("
                << resources[altResourceNumbers[
                                    resourceNumbers[k]]].getName()
                << "): " << altDurations[k]
                << endl;


      if (j != 0)
        model.add(activity.startsAfterEnd(previousActivity));
      previousActivity = activity;
      k++;
    }
    model.add(previousActivity.endsBefore(makespan));
  }

  /* RETURN THE MODEL. */
  return model;
}


////////////////////////////////////////////////////////////////
//
// INITIALIZE THE PROGRAM ARGUMENTS
//
////////////////////////////////////////////////////////////////
void
InitParameters(int argc,
               char** argv,
               IloInt& numberOfJobs,
               IloInt& numberOfResources,
               IloInt*& resourceNumbers,
               IloNum*& durations)
{
  numberOfJobs = 6;
  numberOfResources = 6;
  resourceNumbers = ResourceNumbers06;
  durations = Durations06;
```

```
  if (argc > 1) {
    IloInt number = atol(argv[1]);
    if (number == 10) {
      numberOfJobs = 10;
      numberOfResources = 10;
      resourceNumbers = ResourceNumbers10;
      durations = Durations10;
    }
    else if (number == 20) {
      numberOfJobs = 20;
      numberOfResources = 5;
      resourceNumbers = ResourceNumbers20;
      durations = Durations20;
    }
  }
}

////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////

int main(int argc, char** argv)
{
  IloInt numberOfJobs;
  IloInt numberOfResources;
  IloInt* resourceNumbers;
  IloNum* durations;

  InitParameters(argc,
                 argv,
                 numberOfJobs,
                 numberOfResources,
                 resourceNumbers,
                 durations);

  try {

    IloEnv env;
    IloNumVar makespan;
    IloArray<IloActivityArray> jobs;
    IloRandom randGen(env, 8975324);
    IloSchedulerSolution solution(env);
    IloModel model = DefineModel(env,
                                 numberOfJobs,
                                 numberOfResources,
                                 resourceNumbers,
                                 durations,
                                 randGen,
                                 solution,
                                 jobs,
                                 makespan);

    SolveModel(model, makespan, jobs, solution);
    env.end();
  }
  catch (IloSchedulerException& exc) {
```

```
    cout << exc << endl;
  }
  catch (IloException& exc) {
    cout << exc << endl;
  }

  return 0;
}
```

Part IV

Integrated Applications

This part presents examples that are generally more complex and delve into specific applications. Principles, applications, and programming techniques of interest to a narrower audience are discussed.

C H A P T E R

30

A Randomizing Algorithm: the Job-Shop Problem

When we introduced *temporal constraints* in *Part I, Getting Started with Scheduler,* we emphasized that under certain conditions, the algorithm that Scheduler uses to search for a solution is *complete* in the sense that it either reports that the constraints are incompatible so no solution can be found, or it finds a solution that satisfies the temporal constraints. The algorithms that we looked at for the job-shop problems in Chapter 15 and Chapter 16 were complete in this sense: they either found an optimal solution or they failed and reported reliably that no solution could be found.

One disadvantage of such complete optimization algorithms is that their completeness requires a systematic exploration of the entire search space. A consequence of that thoroughness is that if a "mistake" is made at one of the early decisions, then the algorithm explores all of the possible consequences of this "mistake." Of course, as we've emphasized, constraint propagation reduces the amount of exploration that is performed. However, even the best constraint propagation techniques do not prove sufficient to guarantee a perfect resolution of some hard job-shop scheduling problems in a reasonable amount of CPU time.

This observation suggests that perhaps we should abandon the idea of guaranteeing the optimality of a solution. Instead, we could settle for a solution that is "good enough" in some sense relevant to our problem. We can exercise this option—abandoning the quest for optimality and settling for a good enough solution—simply by stopping an algorithm after a given amount of computational time.

Another—complementary—alternative consists of performing a large number of iterations of an algorithm stopped after a very small number of backtracks. By limiting the amount of backtracking, each iteration may consequently fail either because there is no solution with a given makespan, or because the algorithm could not find such a solution in the given number of backtracks. To allow several iterations with the same makespan to take place, we "randomize" the process of constructing a solution.

As a result, several iterations with the same maximal makespan follow different paths in the search space; the *nth* of these iterations may succeed when the (*n - 1*) previous iterations failed.

Developing the Problem-Solving Algorithm

As we develop the randomized algorithm, we need to consider which activity to schedule first and how to minimize the makespan. The following sections address those issues.

Choosing the Activity to Schedule First

Using a randomized strategy does not mean that everything ought to be done randomly. Indeed, the goal that is pursued at each iteration is still to compute a solution to the scheduling problem with a makespan less than the best makespan found so far. Constraint propagation helps in this process as it identifies activities that cannot be first to execute on the corresponding resource. The basic structure of an algorithm aimed at finding a new solution is consequently very similar to the basic structure of the preceding algorithms.

What is going to differ most is the heuristic we use to select which activity to schedule first. In the context of the preceding algorithms, it was very important to avoid bad decisions about that point. This pressure lead us to schedule the most critical resource first. However, avoiding bad decisions is not that important here, so two other heuristics appear useful:

- First, proceed chronologically so as to build the solution in a consistent fashion. In particular, it is useless to schedule an activity whose earliest start time is later than the earliest end time of another activity.
- Second, select an activity with the resource for which the number of candidates (the number of "possible firsts") is minimal.

The following function, `SelectResourceConstraint`, integrates these two heuristics.

```
IlcInt
GetEndMin(IlcSchedule& schedule)
{
  /* COMPUTE THE SMALLEST MINIMAL END TIME OF UNRANKED ACTIVITIES. */
  IlcInt endMin = schedule.getTimeMax() + 1;
  for(IlcUnaryResourceIterator resIte(schedule);
      resIte.ok();
      ++resIte) {
```

```
    for (IlcResourceConstraintIterator ite(*resIte,
                                           IlcFromStartToEnd);
         ite.ok(); ++ite) {
      IlcResourceConstraint constraint = *ite;
      if ((constraint.isPossibleFirst())
          && (constraint.getActivity().getEndMin() < endMin))
        endMin = constraint.getActivity().getEndMin();
    }
  }
  return endMin;
}

IlcBool
SelectResourceConstraint(IlcResourceConstraint& constraint,
                         IlcSchedule& schedule,
                         IloRandom rand)
{
  IlcInt endMin = GetEndMin(schedule);
  if (schedule.getTimeMax() < endMin)
    return IlcFalse;
  IlcInt chosenNumberOfCandidates = schedule.getNumberOfActivities() +
    1;
  for(IlcUnaryResourceIterator resIte(schedule);
      resIte.ok();
      ++resIte) {
    IlcResourceConstraint bestConstraint;
    IloNum bestValue = -1.0;
    IlcInt numberOfCandidates = 0;
    for (IlcResourceConstraintIterator ite(*resIte,
                                           IlcFromStartToEnd);
         ite.ok(); ++ite) {
      IlcResourceConstraint ct = *ite;
      if (ct.isPossibleFirst()
          && (ct.getActivity().getStartMin() < endMin)) {
        numberOfCandidates++;
        IloNum value = rand.getFloat();
        if (bestValue < value) {
          bestConstraint = ct;
          bestValue = value;
        }
      }
    }
    if ((0 < numberOfCandidates)
        && (numberOfCandidates < chosenNumberOfCandidates)) {
      /* resource IS THE RESOURCE WITH THE SMALLEST NUMBER OF
         CANDIDATES. THE CHOSEN CONSTRAINT BECOMES THE BEST CONSTRAINT
         OF resource. */
      chosenNumberOfCandidates = numberOfCandidates;
      constraint = bestConstraint;
    }
  }
  return IlcTrue;
}
```

The function `SelectResourceConstraint` selects the resource constraint that corresponds to the chosen activity. The function returns `IlcTrue` if it succeeded in selecting a resource constraint, and `IlcFalse` otherwise.

Optimizing the Makespan

Inside the function `MinimizeRandom`, the search goal `SolveProblem` is limited with the function `IloLimitSearch`. More precisely, the number of backtracks is limited to the value of `numberOfFailsPerIteration`, a parameter of the function `MinimizeRandom`.

```
ILCGOAL1(SolveProblemIlc, IloRandom, rand) {
  IloSolver s = getSolver();
  IlcScheduler scheduler = IlcScheduler(s);
  IlcResourceConstraint constraint;
  if (SelectResourceConstraint(constraint, scheduler, rand)) {
    return IlcAnd(IlcTryRankFirst(constraint),
                  this);
  }
  return 0;
}

ILOCPGOALWRAPPER1(SolveProblem, solver, IloRandom, rand) {
  return SolveProblemIlc(solver, rand);
}

void
MinimizeRandom(IloSolver& solver,
               IloNumVar& makespan,
               IloSchedulerSolution& solution,
               IloInt numberOfIterations,
               IloInt numberOfFailsPerIteration)
{
  IloEnv env = solver.getEnv();
  IlcScheduler sched(solver);
  IloRandom randGen(env);
  IloGoal goal;

  /* GENERATE AN INITIAL SOLUTION. */
  goal = IloGoalTrue(solver.getEnv());
  solver.solve(goal);
  IloNum min = solver.getMin(makespan);
  makespan.setLB(min);

  goal = SolveProblem(env, randGen);
  solver.solve(goal);
  IloNum max = solver.getMin(makespan);

  /* OPTIMIZE. */
  IloInt iteration = 0;

  goal = IloLimitSearch(env,
                        SolveProblem(env, randGen),
                        IloFailLimit(env, numberOfFailsPerIteration));

  while ((min != max) && (iteration < numberOfIterations)) {
    iteration++;
    IloNum value = IloTrunc((min + iteration * max) / (iteration + 1));
    makespan.setUB(value);

    solver.out() << “Trying makespan: “ << value << endl;
```

```
    if (solver.solve(goal)) {
      max = solver.getMin(makespan);
      solver.out() << "Solution with makespan " << max << endl;
      solution.store(sched);
    }

    else /* ... */
  }
  /* ... */
}
```

Remembering our experience with the dichotomizing binary search (in Chapter 16), at each iteration, we use a weighted sum of the minimal value `min` and maximal value `max` of the `makespan` variable as a new tentative bound for this constrained variable. Here, it would not be a good idea to systematically take the average of these two values as a new tentative makespan. In fact, it could be that no solution exists with a makespan of `(min+max)/2`, so taking the same tentative makespan at each iteration could lead to ignoring many good solutions to the scheduling problem.

Each iteration may fail either because there is no solution with a given makespan, or because the algorithm could not find such a solution in the given number of backtracks. By testing the number of failures (the value of `solver.getNumberOfFails()`) at the end of the iteration, we know whether the `numberOfFailsPerIteration` backtracks have been consumed or not. If not, the algorithm has proven that no solution exists with a makespan of at most the given `value`. Hence, it is correct to write the following code.

```
    if (solver.solve(goal)) {
      max = solver.getMin(makespan);
      solver.out() << "Solution with makespan " << max << endl;
      solution.store(sched);
    }

    else if (solver.getNumberOfFails() < numberOfFailsPerIteration) {
      /* ACTUAL FAILURE NOT DUE TO THE LIMITED NUMBER OF FAILS. */
      solver.out() << "Failure with makespan " << value << endl;
      min = value + 1;
    }
    else
      solver.out() << "Failure from fail limit" << endl;
  }
```

Notice that the best solution is stored in an instance of the class `IloSchedulerSolution`.

Complete Program and Output

You can see the entire program `jobshopr.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>
```

```
ILOSTLBEGIN

IloInt ResourceNumbers06 [] = {2, 0, 1, 3, 5, 4,
                               1, 2, 4, 5, 0, 3,
                               2, 3, 5, 0, 1, 4,
                               1, 0, 2, 3, 4, 5,
                               2, 1, 4, 5, 0, 3,
                               1, 3, 5, 0, 4, 2};

IloInt Durations06 [] = { 1,  3,  6,  7,  3,  6,
                          8,  5, 10, 10, 10,  4,
                          5,  4,  8,  9,  1,  7,
                          5,  5,  5,  3,  8,  9,
                          9,  3,  5,  4,  3,  1,
                          3,  3,  9, 10,  4,  1};

IloInt ResourceNumbers10 [] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
                               0, 2, 4, 9, 3, 1, 6, 5, 7, 8,
                               1, 0, 3, 2, 8, 5, 7, 6, 9, 4,
                               1, 2, 0, 4, 6, 8, 7, 3, 9, 5,
                               2, 0, 1, 5, 3, 4, 8, 7, 9, 6,
                               2, 1, 5, 3, 8, 9, 0, 6, 4, 7,
                               1, 0, 3, 2, 6, 5, 9, 8, 7, 4,
                               2, 0, 1, 5, 4, 6, 8, 9, 7, 3,
                               0, 1, 3, 5, 2, 9, 6, 7, 4, 8,
                               1, 0, 2, 6, 8, 9, 5, 3, 4, 7};

IloInt Durations10 [] = {29, 78,  9, 36, 49, 11, 62, 56, 44, 21,
                         43, 90, 75, 11, 69, 28, 46, 46, 72, 30,
                         91, 85, 39, 74, 90, 10, 12, 89, 45, 33,
                         81, 95, 71, 99,  9, 52, 85, 98, 22, 43,
                         14,  6, 22, 61, 26, 69, 21, 49, 72, 53,
                         84,  2, 52, 95, 48, 72, 47, 65,  6, 25,
                         46, 37, 61, 13, 32, 21, 32, 89, 30, 55,
                         31, 86, 46, 74, 32, 88, 19, 48, 36, 79,
                         76, 69, 76, 51, 85, 11, 40, 89, 26, 74,
                         85, 13, 61,  7, 64, 76, 47, 52, 90, 45};

IloInt ResourceNumbers20 [] = {0, 1, 2, 3, 4,
                               0, 1, 3, 2, 4,
                               1, 0, 2, 4, 3,
                               1, 0, 4, 2, 3,
                               2, 1, 0, 3, 4,
                               2, 1, 4, 0, 3,
                               1, 0, 2, 3, 4,
                               2, 1, 0, 3, 4,
                               0, 3, 2, 1, 4,
                               1, 2, 0, 3, 4,
                               1, 3, 0, 4, 2,
                               2, 0, 1, 3, 4,
                               0, 2, 1, 3, 4,
                               2, 0, 1, 3, 4,
                               0, 1, 4, 2, 3,
                               1, 0, 3, 4, 2,
                               0, 2, 1, 3, 4,
                               0, 1, 4, 2, 3,
                               1, 2, 0, 3, 4,
                               0, 1, 2, 3, 4};
```

```
IloInt Durations20 [] = {29,  9, 49, 62, 44,
                         43, 75, 69, 46, 72,
                         91, 39, 90, 12, 45,
                         81, 71,  9, 85, 22,
                         14, 22, 26, 21, 72,
                         84, 52, 48, 47,  6,
                         46, 61, 32, 32, 30,
                         31, 46, 32, 19, 36,
                         76, 76, 85, 40, 26,
                         85, 61, 64, 47, 90,
                         78, 36, 11, 56, 21,
                         90, 11, 28, 46, 30,
                         85, 74, 10, 89, 33,
                         95, 99, 52, 98, 43,
                          6, 61, 69, 49, 53,
                          2, 95, 72, 65, 25,
                         37, 13, 21, 89, 55,
                         86, 74, 88, 48, 79,
                         69, 51, 11, 89, 74,
                         13,  7, 76, 52, 45};

////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////

IloModel
DefineModel(IloEnv& env,
            IloInt numberOfJobs,
            IloInt numberOfResources,
            IloInt* resourceNumbers,
            IloInt* durations,
            IloNumVar& makespan,
            IloSchedulerSolution& solution)
{
  IloModel model(env);

  /* CREATE THE MAKESPAN VARIABLE. */
  IloInt numberOfActivities = numberOfJobs * numberOfResources;
  IloInt horizon = 0;
  IloInt k;

  for (k = 0; k < numberOfActivities; k++)
    horizon += durations[k];

  makespan = IloNumVar(env, 0, horizon, ILOINT);
  solution = IloSchedulerSolution(env);

  /* CREATE THE RESOURCES. */
  IloSchedulerEnv schedEnv(env);
  schedEnv.getResourceParam().setCapacityEnforcement(IloMediumHigh);

  IloInt j;
  IloUnaryResource *resources =
    new (env) IloUnaryResource[numberOfResources];
  for (j = 0; j < numberOfResources; j++)
```

```
    resources[j] = IloUnaryResource(env);

  /* CREATE THE ACTIVITIES. */
  char buffer[128];
  k = 0;
  IloInt i;
  for (i = 0; i < numberOfJobs; i++) {
    IloActivity previousActivity;
    for (j = 0; j < numberOfResources; j++) {
      IloActivity activity(env, durations[k]);
      sprintf(buffer, "J%ldS%ldR%ld", i, j, resourceNumbers[k]);
      activity.setName(buffer);
      model.add(activity.requires(resources[resourceNumbers[k]]));
      if (j != 0)
        model.add(activity.startsAfterEnd(previousActivity));
      solution.add(activity);
      previousActivity = activity;
      k++;
    }
    model.add(previousActivity.endsBefore(makespan));
  }

  /* RETURN THE MODEL. */
  return model;
}


////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////////////////

void
PrintSolution(const IloSchedulerSolution& solution)
{
  IloEnv env = solution.getEnv();
  for(IloSchedulerSolution::ActivityIterator ite(solution);
  ite.ok(); ++ite) {
    IloActivity act = *ite;
    env.out() << act.getName() << " [";
    env.out() << solution.getStartMin(act) << " -- ";
    env.out() << solution.getDurationMin(act) << " -- ";
    env.out() << solution.getEndMin(act) << "]" << endl;
  }
}

////////////////////////////////////////////////////////////////////////////
//
// PROBLEM SOLVING
//
////////////////////////////////////////////////////////////////////////////

IlcInt
GetEndMin(IlcSchedule& schedule)
{
  /* COMPUTE THE SMALLEST MINIMAL END TIME OF UNRANKED ACTIVITIES. */
  IlcInt endMin = schedule.getTimeMax() + 1;
```

```
  for(IlcUnaryResourceIterator resIte(schedule);
      resIte.ok();
      ++resIte) {
    for (IlcResourceConstraintIterator ite(*resIte,
                                           IlcFromStartToEnd);
         ite.ok(); ++ite) {
      IlcResourceConstraint constraint = *ite;
      if ((constraint.isPossibleFirst())
          && (constraint.getActivity().getEndMin() < endMin))
        endMin = constraint.getActivity().getEndMin();
    }
  }
  return endMin;
}

IlcBool
SelectResourceConstraint(IlcResourceConstraint& constraint,
                         IlcSchedule& schedule,
                         IloRandom rand)
{
  IlcInt endMin = GetEndMin(schedule);
  if (schedule.getTimeMax() < endMin)
    return IlcFalse;
  IlcInt chosenNumberOfCandidates = schedule.getNumberOfActivities() +
    1;
  for(IlcUnaryResourceIterator resIte(schedule);
      resIte.ok();
      ++resIte) {
    IlcResourceConstraint bestConstraint;
    IloNum bestValue = -1.0;
    IlcInt numberOfCandidates = 0;
    for (IlcResourceConstraintIterator ite(*resIte,
                                           IlcFromStartToEnd);
         ite.ok(); ++ite) {
      IlcResourceConstraint ct = *ite;
      if (ct.isPossibleFirst()
          && (ct.getActivity().getStartMin() < endMin)) {
        numberOfCandidates++;
        IloNum value = rand.getFloat();
        if (bestValue < value) {
          bestConstraint = ct;
          bestValue = value;
        }
      }
    }
    if ((0 < numberOfCandidates)
        && (numberOfCandidates < chosenNumberOfCandidates)) {
      /* resource IS THE RESOURCE WITH THE SMALLEST NUMBER OF
         CANDIDATES. THE CHOSEN CONSTRAINT BECOMES THE BEST CONSTRAINT
         OF resource. */
      chosenNumberOfCandidates = numberOfCandidates;
      constraint = bestConstraint;
    }
  }
  return IlcTrue;
}
ILCGOAL1(SolveProblemIlc, IloRandom, rand) {
  IloSolver s = getSolver();
```

```
  IlcScheduler scheduler = IlcScheduler(s);
  IlcResourceConstraint constraint;
  if (SelectResourceConstraint(constraint, scheduler, rand)) {
    return IlcAnd(IlcTryRankFirst(constraint),
                  this);
  }
  return 0;
}

ILOCPGOALWRAPPER1(SolveProblem, solver, IloRandom, rand) {
  return SolveProblemIlc(solver, rand);
}

void
MinimizeRandom(IloSolver& solver,
               IloNumVar& makespan,
               IloSchedulerSolution& solution,
               IloInt numberOfIterations,
               IloInt numberOfFailsPerIteration)
{
  IloEnv env = solver.getEnv();
  IlcScheduler sched(solver);
  IloRandom randGen(env);
  IloGoal goal;

  /* GENERATE AN INITIAL SOLUTION. */
  goal = IloGoalTrue(solver.getEnv());
  solver.solve(goal);
  IloNum min = solver.getMin(makespan);
  makespan.setLB(min);

  goal = SolveProblem(env, randGen);
  solver.solve(goal);
  IloNum max = solver.getMin(makespan);

  /* OPTIMIZE. */
  IloInt iteration = 0;

  goal = IloLimitSearch(env,
                        SolveProblem(env, randGen),
                        IloFailLimit(env, numberOfFailsPerIteration));

  while ((min != max) && (iteration < numberOfIterations)) {
    iteration++;
    IloNum value = IloTrunc((min + iteration * max) / (iteration + 1));
    makespan.setUB(value);

    solver.out() << "Trying makespan: " << value << endl;
    if (solver.solve(goal)) {
      max = solver.getMin(makespan);
      solver.out() << "Solution with makespan " << max << endl;
      solution.store(sched);
    }

    else if (solver.getNumberOfFails() < numberOfFailsPerIteration) {
      /* ACTUAL FAILURE NOT DUE TO THE LIMITED NUMBER OF FAILS. */
      solver.out() << "Failure with makespan " << value << endl;
      min = value + 1;
```

```
    }
    else
      solver.out() << "Failure from fail limit" << endl;
  }
}

////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////

void
InitParameters(int argc,
               char** argv,
               IloInt& numberOfJobs,
               IloInt& numberOfResources,
               IloInt*& resourceNumbers,
               IloInt*& durations,
               IloInt& numberOfIterations,
               IloInt& numberOfFailsPerIteration)
{
  if (argc > 1) {
    IloInt number = atol(argv[1]);
    if (number == 10) {
      numberOfJobs = 10;
      numberOfResources = 10;
      resourceNumbers = ResourceNumbers10;
      durations = Durations10;
    }
    else if (number == 20) {
      numberOfJobs = 20;
      numberOfResources = 5;
      resourceNumbers = ResourceNumbers20;
      durations = Durations20;
    }
  }
  if (argc > 2)
    numberOfIterations = atol(argv[2]);
  if (argc > 3)
    numberOfFailsPerIteration = atol(argv[3]);
}

int main(int argc, char** argv)
{
  try {
    IloEnv env;

    IloInt numberOfJobs = 6;
    IloInt numberOfResources = 6;
    IloInt* resourceNumbers = ResourceNumbers06;
    IloInt* durations = Durations06;
    IloInt numberOfIterations = 100;
    IloInt numberOfFailsPerIteration = 10;
    InitParameters(argc,
                   argv,
                   numberOfJobs,
                   numberOfResources,
```

```
                    resourceNumbers,
                    durations,
                    numberOfIterations,
                    numberOfFailsPerIteration);
    IloNumVar makespan;
    IloSchedulerSolution solution;
    IloModel model = DefineModel(env,
                                 numberOfJobs,
                                 numberOfResources,
                                 resourceNumbers,
                                 durations,
                                 makespan,
                                 solution);
    IloSolver solver(model);
    MinimizeRandom(solver,
                    makespan,
                    solution,
                    numberOfIterations,
                    numberOfFailsPerIteration);
    PrintSolution(solution);
    solver.printInformation();
    env.end();
  }
  catch (IloException& exc) {
    cout << exc << endl;
  }

  return 0;
}

////////////////////////////////////////////////////////////////////////////
//
// RESULTS
//
////////////////////////////////////////////////////////////////////////////
/*
jobshopr 6
Trying makespan: 61
Solution with makespan 61
Trying makespan: 56
Solution with makespan 56
Trying makespan: 53
Failure with makespan 53
Trying makespan: 55
Solution with makespan 55
Trying makespan: 54
Failure with makespan 54
J0S0R2 [5 -- 1 -- 6]
J0S1R0 [6 -- 3 -- 9]
J0S2R1 [16 -- 6 -- 22]
J0S3R3 [30 -- 7 -- 37]
J0S4R5 [38 -- 3 -- 41]
J0S5R4 [49 -- 6 -- 55]
J1S0R1 [0 -- 8 -- 8]
J1S1R2 [8 -- 5 -- 13]
J1S2R4 [13 -- 10 -- 23]
J1S3R5 [28 -- 10 -- 38]
J1S4R0 [38 -- 10 -- 48]
```

```
J1S5R3 [48 -- 4 -- 52]
J2S0R2 [0 -- 5 -- 5]
J2S1R3 [5 -- 4 -- 9]
J2S2R5 [9 -- 8 -- 17]
J2S3R0 [18 -- 9 -- 27]
J2S4R1 [27 -- 1 -- 28]
J2S5R4 [30 -- 7 -- 37]
J3S0R1 [8 -- 5 -- 13]
J3S1R0 [13 -- 5 -- 18]
J3S2R2 [22 -- 5 -- 27]
J3S3R3 [27 -- 3 -- 30]
J3S4R4 [37 -- 8 -- 45]
J3S5R5 [45 -- 9 -- 54]
J4S0R2 [13 -- 9 -- 22]
J4S1R1 [22 -- 3 -- 25]
J4S2R4 [25 -- 5 -- 30]
J4S3R5 [41 -- 4 -- 45]
J4S4R0 [48 -- 3 -- 51]
J4S5R3 [52 -- 1 -- 53]
J5S0R1 [13 -- 3 -- 16]
J5S1R3 [16 -- 3 -- 19]
J5S2R5 [19 -- 9 -- 28]
J5S3R0 [28 -- 10 -- 38]
J5S4R4 [45 -- 4 -- 49]
J5S5R2 [49 -- 1 -- 50]
*/
```

Computational Remarks: Using a Combination of Algorithms

Using the randomizing algorithm, we can solve the job-shop scheduling problem MT20 in less than 100 iterations, with at most 10 backtracks per iteration. In contrast, MT10 is not satisfactorily solved.

The results of this program strongly confirm what we've already suggested: that there are many possible algorithms to solve a given type of problem, and that a combination of algorithms may get better results, on average, than each algorithm separately.

C H A P T E R

31

Handling an Overconstrained Problem: Adding Due Dates to the Bridge Problem

In this chapter we revisit the *overconstrained* bridge problem from Chapter 25 and use Solver functions to change the model. Critical deadlines are iteratively removed and the problem is resolved at each iteration. This may yield a "better" solution, but for large problems, can also be more time-consuming than the approach presented in Chapter 25.

Defining the Problem

The array `DueDates` gives a targeted due date for all activities in the problem.

```
IloInt DueDates [] ={  8, 5, 12, 12, 18, 24,   // A1 -> A6
                       35, 17,                 // P1 -> P2
                       18, 10, 38, 23, 26, 37, // S1 -> S6
                       19, 11, 43, 20, 28, 38, // B1 -> B6
                       20, 13, 43, 26, 28, 39, // AB1 -> AB6
                       33, 22, 60, 53, 44, 80, // M1 -> M6
                       49, 105, 77, 65, 93,    // T1 -> T5
                       60, 102,                // V1 -> V2
                       32,                     // L
                       12,                     // UE
                       105,                    // UA
                       110};                   // PE

const IloInt NumberOfActivities = 43;
```

This array is used to express that we want activity A1 to be finished eight days after the beginning of the project, activity A2 five days after the beginning of the project, and so on. These due date constraints are added with the Concert Technology member function IloNumVar::setUb. The function DefineModel includes parameters for the activities and due dates.

The array activities contains all the activities of the problem such that, for each element activities[i] of this array, its associated targeted due date is equal to dueDates[i].

Note that the Concert Technology member function setObject is used to associate each activity with its due date constraint. By doing that, we can retrieve easily, outside the function DefineModel, the due date constraint of each activity.

```
IloModel
DefineModel(IloEnv env,
            IloInt* dueDates,
            IloNumVar& makespan,
            IloInt numberOfActivities) {
  /* ... */

  /* DUE DATES */
  IloInt i;
  for (i = 0 ; i < numberOfActivities ; i++) {
    activities[i].setEndMax(dueDates[i]);
    activities[i].setObject(&(dueDates[i]));
  }
  /* ... */
}
```

Solving the Problem

The solution search is done with a call to the member function IloSolver::solve.

```
IloEnv env;
IloNumVar makespan;
IloModel model = DefineModel(env,
                             DueDates,
                             makespan,
                             NumberOfActivities);

IloSolver solver(model);
IloGoal goal = IloRankForward(env, makespan);
IlcScheduler scheduler(solver);

/* FIRST TRY: TARGETED DUE DATES */
solver.out() << endl << "----------------------------------------" << endl;
solver.out() << "First try: targeted due dates" << endl;

if (solver.solve(goal)) {
  solver.out() << "current value of makespan: "
               << solver.getMin(makespan) << endl;
```

```
  PrintSolution(solver);
}
else
  solver.out() << "No solution! " << endl;
```

However, no solution to the problem is found because some targeted due dates are too strong to be satisfied. We can relax the problem by removing selected constraints and this should lead to a solvable problem. For instance, we could apply the following procedure:

1. Remove the first targeted due date and search for a solution.
2. If there is no solution, remove the second targeted due date and search again for a solution.
3. Continue the process with the third targeted due date, then the fourth, etc., until a solution is found.

This way of proceeding is too simple and appears to be inefficient because we have no guarantee that we remove the strongest due date constraints.

Using Minimal Due Dates

We choose a slightly more sophisticated way to remove targeted due dates. First, we search for the *minimal due dates* of each activity if we had no capacity constraints. By definition, a minimal due date is the minimal end time of an activity. If a targeted due date is less than its minimal due date, we know for certain that the targeted due date cannot be satisfied.

It seems intuitive that a targeted due date should be easier to satisfy if it is far from its minimal due date. Conversely, it also seems intuitive that if a targeted due date is near its minimal due date, it should be more difficult to satisfy.

We temporarily remove the capacity constraints to find the minimal due dates, search for a solution, and then store each minimal due date with the Concert Technology function `setLb`.

```
/* SECOND TRY: RELAXED CAPACITIES */

IloSchedulerEnv schedEnv(env);
IloResourceParam resParam = schedEnv.getResourceParam();
resParam.ignoreCapacityConstraints(IloTrue);

solver.out() << endl << "----------------------------------------" << endl;
solver.out() << "Second try: relaxed capacities" << endl;

if (solver.solve(goal)) {
  solver.out() << "current value of makespan: " << solver.getMin(makespan)
               << endl;
  solver.out() << "Overlapping activities: " << endl;
  PrintOverlappingActivities(solver);
  for(IloIterator<IloActivity> iter(env); iter.ok(); ++iter) {
    IloActivity act = *iter;
    act.setEndMin(scheduler.getActivity(act).getEndMin());
  }
```

```
    }
    else
      solver.out() << “No solution! “ << endl;
```

In addition, the function `PrintOverlappingActivities` is called to display the activities which overlap.

```
void
PrintOverlappingActivities(const IloSolver& solver)
{
  IlcScheduler scheduler(solver);
  for (IlcResourceIterator resIter(scheduler); resIter.ok() ; ++resIter) {
    for (IlcResourceConstraintIterator RCiter(*resIter);
         RCiter.ok();
         ++RCiter) {
      IlcActivity act = RCiter.getActivity();
      for (IlcResourceConstraintIterator RCiter2(*resIter);
           RCiter2.ok();
           ++RCiter2) {
        IlcActivity act2 = RCiter2.getActivity();
        if (act == act2)
          break;
        IlcInt endMin1 = act.getEndMin();
        IlcInt startMax1 = act.getStartMax();
        if (endMin1 > startMax1) {
          IlcInt endMin2 = act2.getEndMin();
          IlcInt startMax2 = act2.getStartMax();
          if (endMin2 > startMax2)
            if ((startMax1 < endMin2) && (startMax2 < endMin1))
              solver.out() << act << “ and “ << act2 << “ overlap “ << endl;
        }
      }
    }
  }
}
```

Displaying the overlapping activities is done here only for information but one could define a repair policy from the set of these overlapping activities.

Removing Targeted Due Dates

Once we know the minimal due dates, we can re-add the capacity constraints and remove the due date constraints incrementally. We want to remove the fewest number of due date constraints possible and start by removing what we believe are the strongest targeted due dates to satisfy.

To do that, we remove constraints incrementally:

1. We remove the targeted due dates which are less than or equal to their minimal due dates and search for a solution.
2. If there is no solution, we remove the targeted due dates which are less than or equal to their minimal due dates plus *one* day.

3. If there is still no solution, we remove the targeted due dates which are less than or equal to their minimal due dates plus *two* days.

4. We repeat this procedure until a solution is found.

The procedure is defined in the following code.

```
/* THIRD TRY: SOME DUE DATES ARE INCREMENTALLY REMOVED  */

resParam.ignoreCapacityConstraints(IloFalse);

solver.out() << endl << "----------------------------------------" << endl;
solver.out() << "Third try: some due dates are incrementally removed "
             << endl;
IloInt j;
for (j=0 ; j < 99 ; j++) {

  solver.out() << endl << "Pass number " << j+1 << endl;
  for(IloIterator<IloActivity> iter(env); iter.ok(); ++iter) {
    IloActivity act = *iter;
    IloInt ub = (IloInt)act.getEndMax();
    IloInt lb = (IloInt)act.getEndMin();
    if (ub - lb <= j) {
      solver.out() << "\tRemove [" << act.getName() << " ends before "
                   << ub << "]" << endl;
                                                act.setEndMax(IloIntMax);
    }
  }

  if (solver.solve(goal)) {
    solver.out() << "\tcurrent value of makespan: "
                 << solver.getMin(makespan) << endl;
    solver.out() << "\tSolution: " << endl;
    PrintSolution(solver);
    break;
  }
  else
    solver.out() << "\tNo solution! " << endl;
}
```

The execution of the program shows that in the first pass, 4 due date constraints have been removed (the targeted due dates of the activities `S2`, `B2`, `B4`, and `L`), but no solution was found. In the second pass, the due dates of activities `AB2`, `M1`, and `V1` have been removed but there is still no solution. Finally, in the sixth pass, a solution is found. Due dates which are violated are displayed with the function `PrintSolution`.

```
void
PrintSolution(const IloSolver& solver)
{
  IlcScheduler scheduler(solver);
  IloEnv env = solver.getEnv();
  for(IloIterator<IloActivity> ite(env); ite.ok(); ++ite) {
    IloActivity act = *ite;
    solver.out() << scheduler.getActivity(act);
    IloInt dueDate = *((IloInt *)act.getObject());
    if (scheduler.getActivity(act).getEndMin() > dueDate) {
      solver.out() << "  <- violated due date: " << dueDate;
```

```
    }
    solver.out() << endl;
  }
}
```

A due date of an activity is violated if its earliest possible end time is greater than its associated due date. Note that the Concert Technology member function `getObject` is used to get the due date constraint associated with an activity.

Complete Program and Output

You can see the entire program `bridgeov.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////

IloInt DueDates [] ={  8, 5, 12, 12, 18, 24,   // A1 -> A6
                       35, 17,                 // P1 -> P2
                       18, 10, 38, 23, 26, 37, // S1 -> S6
                       19, 11, 43, 20, 28, 38, // B1 -> B6
                       20, 13, 43, 26, 28, 39, // AB1 -> AB6
                       33, 22, 60, 53, 44, 80, // M1 -> M6
                       49, 105, 77, 65, 93,    // T1 -> T5
                       60, 102,                // V1 -> V2
                       32,                     // L
                       12,                     // UE
                       105,                    // UA
                       110};                   // PE

const IloInt NumberOfActivities = 43;
IloActivity
MakeActivity(IloModel model,
             const char* name,
             IloInt duration)
{
  IloEnv env = model.getEnv();
  IloActivity activity(env, duration, name);
  return activity;
}


IloUnaryResource
MakeResource(IloModel model,
             const char* name,
             IloInt numberOfActivities,
             IloActivityArray activities)
```

```
{
  IloEnv env = model.getEnv();
  IloUnaryResource resource(env, name);
  for (IloInt i = 0; i < numberOfActivities; i++) {
    model.add(activities[i].requires(resource));
  }

  return resource;
}
IloModel
DefineModel(IloEnv env,
            IloInt* dueDates,
            IloNumVar& makespan,
            IloInt numberOfActivities) {
  /* CREATE THE SCHEDULE. */
  IloModel model(env);
  IloSchedulerEnv schedEnv(env);
  schedEnv.getResourceParam().setCapacityEnforcement(IloMediumHigh);

  IloInt horizon = 365;
  schedEnv.setHorizon(horizon);

  /* CREATE ACTIVITIES AND RESOURCES: EXCAVATIONS. */
  IloActivity A1 = MakeActivity(model, “A1”, 4);
  IloActivity A2 = MakeActivity(model, “A2”, 2);
  IloActivity A3 = MakeActivity(model, “A3”, 2);
  IloActivity A4 = MakeActivity(model, “A4”, 2);
  IloActivity A5 = MakeActivity(model, “A5”, 2);
  IloActivity A6 = MakeActivity(model, “A6”, 5);
  IloActivityArray A(env,6);
  A[0]=A1; A[1]=A2; A[2]=A3; A[3]=A4; A[4]=A5; A[5]=A6;
  MakeResource(model, “EXCAVATOR”, 6, A);

  /* CREATE ACTIVITIES AND RESOURCES: FOUNDATIONS. */
  IloActivity P1 = MakeActivity(model, “P1”, 20);
  IloActivity P2 = MakeActivity(model, “P2”, 13);
  IloActivityArray P(env,2);
  P[0]=P1; P[1]=P2;
  MakeResource(model, “PILE DRIVER”, 2, P);

  /* CREATE ACTIVITIES AND RESOURCES: FORMWORKS. */
  IloActivity S1 = MakeActivity(model, “S1”, 8);
  IloActivity S2 = MakeActivity(model, “S2”, 4);
  IloActivity S3 = MakeActivity(model, “S3”, 4);
  IloActivity S4 = MakeActivity(model, “S4”, 4);
  IloActivity S5 = MakeActivity(model, “S5”, 4);
  IloActivity S6 = MakeActivity(model, “S6”, 10);
  IloActivityArray S(env,6);
  S[0]=S1; S[1]=S2; S[2]=S3; S[3]=S4; S[4]=S5; S[5]=S6;
  MakeResource(model, “CARPENTRY”, 6, S);

  /* CREATE ACTIVITIES AND RESOURCES: CONCRETE FOUNDATIONS. */
  IloActivity B1 = MakeActivity(model, “B1”, 1);
  IloActivity B2 = MakeActivity(model, “B2”, 1);
  IloActivity B3 = MakeActivity(model, “B3”, 1);
  IloActivity B4 = MakeActivity(model, “B4”, 1);
  IloActivity B5 = MakeActivity(model, “B5”, 1);
  IloActivity B6 = MakeActivity(model, “B6”, 1);
```

```
IloActivityArray B(env,6);
B[0]=B1; B[1]=B2; B[2]=B3; B[3]=B4; B[4]=B5; B[5]=B6;
MakeResource(model, “CONCRETE MIXER”, 6, B);

/* CREATE ACTIVITIES AND RESOURCES: CONCRETE SETTING TIMES. */
IloActivity AB1 = MakeActivity(model, “AB1”, 1);
IloActivity AB2 = MakeActivity(model, “AB2”, 1);
IloActivity AB3 = MakeActivity(model, “AB3”, 1);
IloActivity AB4 = MakeActivity(model, “AB4”, 1);
IloActivity AB5 = MakeActivity(model, “AB5”, 1);
IloActivity AB6 = MakeActivity(model, “AB6”, 1);
IloActivityArray AB(env,6);
AB[0]=AB1; AB[1]=AB2; AB[2]=AB3; AB[3]=AB4; AB[4]=AB5; AB[5]=AB6;

/* CREATE ACTIVITIES AND RESOURCES: MASONRY. */
IloActivity M1 = MakeActivity(model, “M1”, 16);
IloActivity M2 = MakeActivity(model, “M2”, 8);
IloActivity M3 = MakeActivity(model, “M3”, 8);
IloActivity M4 = MakeActivity(model, “M4”, 8);
IloActivity M5 = MakeActivity(model, “M5”, 8);
IloActivity M6 = MakeActivity(model, “M6”, 20);
IloActivityArray M(env,6);
M[0]=M1; M[1]=M2; M[2]=M3; M[3]=M4; M[4]=M5; M[5]=M6;
MakeResource(model, “BRICKLAYING”, 6, M);

/* CREATE ACTIVITIES: POSITIONING. */
IloActivity T1 = MakeActivity(model, “T1”, 12);
IloActivity T2 = MakeActivity(model, “T2”, 12);
IloActivity T3 = MakeActivity(model, “T3”, 12);
IloActivity T4 = MakeActivity(model, “T4”, 12);
IloActivity T5 = MakeActivity(model, “T5”, 12);
IloActivityArray T(env,5);
T[0]=T1; T[1]=T2; T[2]=T3; T[3]=T4; T[4]=T5;
MakeResource(model, “CRANE”, 5, T);

/* CREATE ACTIVITIES: FILLING. */
IloActivity V1 = MakeActivity(model, “V1”, 15);
IloActivity V2 = MakeActivity(model, “V2”, 10);
IloActivityArray V(env,2);
V[0]=V1; V[1]=V2;
MakeResource(model, “CATERPILLAR”, 2, V);
IloActivity PE = MakeActivity(model, “PE”, 0);
makespan = IloNumVar(env, 0, IloInfinity, ILOINT);
model.add(PE.endsAt(makespan));

/* CREATE ACTIVITIES: DELIVERY OF THE PREFORMED BEARERS. */
IloActivity L = MakeActivity(model, “L”,  2);

/* CREATE ACTIVITIES: REMOVAL OF THE TEMPORARY HOUSINGS. */
IloActivity UE = MakeActivity(model, “UE”, 10);
IloActivity UA = MakeActivity(model, “UA”, 10);

/* CREATE ACTIVITIES: PROJECT END. THE makespan POINTER IS SET TO
   THE END-TIME VARIABLE OF THE PROJECT END. */

/* DELIVERY OF THE PREFORMED BEARERS OCCURS EXACTLY 30 DAYS AFTER
   THE BEGINNING OF THE PROJECT. */
L.setStartMin(30);
```

```
IloInt k;
for(k = 0; k < 5; k++) {
  /* POSITIONING STARTS AFTER THE DELIVERY OF THE PREFORMED BEARERS. */
  model.add(T[k].startsAfterEnd(L));
  /* MASONRY WORKS M[k] AND M[k + 1] PRECEDE POSITIONING T[k]. */
  model.add(T[k].startsAfterEnd(M[k]));
  model.add(T[k].startsAfterEnd(M[k + 1]));
}

for(k = 0; k < 6; k++) {
  /* FORMWORKS Sk PRECEDE CONCRETE FOUNDATIONS Bk. */
  model.add(B[k].startsAfterEnd(S[k]));
  /* CONCRETE FOUNDATIONS Bk PRECEDE CONCRETE SETTING TIMES ABk. */
  model.add(AB[k].startsAfterEnd(B[k]));
  /* CONCRETE SETTING TIMES ABk PRECEDE MASONRIES Mk. */
  model.add(M[k].startsAfterEnd(AB[k]));
  /* THE TIME BETWEEN THE COMPLETION OF A FORMWORK Sk AND THE
     COMPLETION OF ITS CORRESPONDING CONCRETE FOUNDATION Bk IS AT
     MOST 4 DAYS. */
  model.add(S[k].endsAfterEnd(B[k], -4));
  /* FORMWORKS Sk MUST BEGIN AT LEAST SIX DAYS AFTER THE BEGINNING
     OF ERECTION OF TEMPORARY HOUSING UE. */
  model.add(S[k].startsAfterStart(UE, 6));
  /* THE REMOVAL OF THE TEMPORARY HOUSING UA CAN START TWO DAYS
     BEFORE THE END OF THE LAST MASONRY WORK. */
  model.add(UA.startsAfterEnd(M[k], -2));
}

/* EXCAVATIONS PRECEDE FOUNDATIONS. */
model.add(P1.startsAfterEnd(A3));
model.add(P2.startsAfterEnd(A4));

/* EXCAVATIONS AND FOUNDATIONS PRECEDE FORMWORKS. */
model.add(S1.startsAfterEnd(A1));
model.add(S2.startsAfterEnd(A2));
model.add(S3.startsAfterEnd(P1));
model.add(S4.startsAfterEnd(P2));
model.add(S5.startsAfterEnd(A5));
model.add(S6.startsAfterEnd(A6));

/* POSITIONING OF BEARERS PRECEDE FILLING. */
model.add(V1.startsAfterEnd(T1));
model.add(V2.startsAfterEnd(T5));

/* THERE ARE AT MOST THREE DAYS BEFORE THE END OF A PARTICULAR
   EXCAVATION (OR FOUNDATION PILES) AND THE BEGINNING OF THE
   CORRESPONDING FORMWORK. */
model.add(A1.endsAfterStart(S1, -3));
model.add(A2.endsAfterStart(S2, -3));
model.add(P1.endsAfterStart(S3, -3));
model.add(P2.endsAfterStart(S4, -3));
model.add(A5.endsAfterStart(S5, -3));
model.add(A6.endsAfterStart(S6, -3));
/* PROJECT END. */
model.add(PE.startsAfterEnd(V1));
model.add(PE.startsAfterEnd(T2));
model.add(PE.startsAfterEnd(T3));
```

```
  model.add(PE.startsAfterEnd(T4));
  model.add(PE.startsAfterEnd(V2));
  model.add(PE.startsAfterEnd(UA));

  IloActivityArray activities(env,numberOfActivities);
  activities[0]  = A1;  activities[1]   = A2;  activities[2] = A3;
  activities[3]  = A4;  activities[4]   = A5;  activities[5] = A6;
  activities[6]  = P1;  activities[7]   = P2;
  activities[8]  = S1;  activities[9]   = S2;  activities[10] = S3;
  activities[11] = S4;  activities[12]  = S5;  activities[13] = S6;
  activities[14] = B1;  activities[15]  = B2;  activities[16] = B3;
  activities[17] = B4;  activities[18]  = B5;  activities[19] = B6;
  activities[20] = AB1; activities[21]  = AB2; activities[22] = AB3;
  activities[23] = AB4; activities[24]  = AB5; activities[25] = AB6;
  activities[26] = M1;  activities[27]  = M2;  activities[28] = M3;
  activities[29] = M4;  activities[30]  = M5;  activities[31] = M6;
  activities[32] = T1;  activities[33]  = T2;  activities[34] = T3;
  activities[35] = T4;  activities[36]  = T5;
  activities[37] = V1;  activities[38]  = V2;  activities[39] = L;
  activities[40] = UE;  activities[41]  = UA;  activities[42] = PE;
  /* DUE DATES */
  IloInt i;
  for (i = 0 ; i < numberOfActivities ; i++) {
    activities[i].setEndMax(dueDates[i]);
    activities[i].setObject(&(dueDates[i]));
  }
  /* RETURN THE CREATED SCHEDULE. */
  return model;
}

////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////////////////
void
PrintSolution(const IloSolver& solver)
{
  IlcScheduler scheduler(solver);
  IloEnv env = solver.getEnv();
  for(IloIterator<IloActivity> ite(env); ite.ok(); ++ite) {
    IloActivity act = *ite;
    solver.out() << scheduler.getActivity(act);
    IloInt dueDate = *((IloInt *)act.getObject());
    if (scheduler.getActivity(act).getEndMin() > dueDate) {
      solver.out() << "  <- violated due date: " << dueDate;
    }
    solver.out() << endl;
  }
}
void
PrintOverlappingActivities(const IloSolver& solver)
{
  IlcScheduler scheduler(solver);
  for (IlcResourceIterator resIter(scheduler); resIter.ok() ; ++resIter) {
    for (IlcResourceConstraintIterator RCiter(*resIter);
         RCiter.ok();
         ++RCiter) {
```

```
      IlcActivity act = RCiter.getActivity();
      for (IlcResourceConstraintIterator RCiter2(*resIter);
           RCiter2.ok();
           ++RCiter2) {
        IlcActivity act2 = RCiter2.getActivity();
        if (act == act2)
          break;
        IlcInt endMin1 = act.getEndMin();
        IlcInt startMax1 = act.getStartMax();
        if (endMin1 > startMax1) {
          IlcInt endMin2 = act2.getEndMin();
          IlcInt startMax2 = act2.getStartMax();
          if (endMin2 > startMax2)
            if ((startMax1 < endMin2) && (startMax2 < endMin1))
              solver.out() << act << " and " << act2 << " overlap " << endl;
        }
      }
    }
  }
}


////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////

int main()
{
  try {
    IloEnv env;
    IloNumVar makespan;
    IloModel model = DefineModel(env,
                                 DueDates,
                                 makespan,
                                 NumberOfActivities);

    IloSolver solver(model);
    IloGoal goal = IloRankForward(env, makespan);
    IlcScheduler scheduler(solver);

    /* FIRST TRY: TARGETED DUE DATES */
    solver.out() << endl << "----------------------------------------" << endl;
    solver.out() << "First try: targeted due dates" << endl;

    if (solver.solve(goal)) {
      solver.out() << "current value of makespan: "
                   << solver.getMin(makespan) << endl;
      PrintSolution(solver);
    }
    else
      solver.out() << "No solution! " << endl;
    /* SECOND TRY: RELAXED CAPACITIES */

    IloSchedulerEnv schedEnv(env);
    IloResourceParam resParam = schedEnv.getResourceParam();
    resParam.ignoreCapacityConstraints(IloTrue);
```

```
    solver.out() << endl << "----------------------------------------" << endl;
    solver.out() << "Second try: relaxed capacities" << endl;

    if (solver.solve(goal)) {
      solver.out() << "current value of makespan: " << solver.getMin(makespan)
                   << endl;
      solver.out() << "Overlapping activities: " << endl;
      PrintOverlappingActivities(solver);
      for(IloIterator<IloActivity> iter(env); iter.ok(); ++iter) {
        IloActivity act = *iter;
        act.setEndMin(scheduler.getActivity(act).getEndMin());
      }
    }
    else
      solver.out() << "No solution! " << endl;
    /* THIRD TRY: SOME DUE DATES ARE INCREMENTALLY REMOVED  */

    resParam.ignoreCapacityConstraints(IloFalse);

    solver.out() << endl << "----------------------------------------" << endl;
    solver.out() << "Third try: some due dates are incrementally removed "
                 << endl;
    IloInt j;
    for (j=0 ; j < 99 ; j++) {

      solver.out() << endl << "Pass number " << j+1 << endl;
      for(IloIterator<IloActivity> iter(env); iter.ok(); ++iter) {
        IloActivity act = *iter;
        IloInt ub = (IloInt)act.getEndMax();
        IloInt lb = (IloInt)act.getEndMin();
        if (ub - lb <= j) {
          solver.out() << "\tRemove [" << act.getName() << " ends before "
                       << ub << "]" << endl;
                                                  act.setEndMax(IloIntMax);
        }
      }

      if (solver.solve(goal)) {
        solver.out() << "\tcurrent value of makespan: "
                     << solver.getMin(makespan) << endl;
        solver.out() << "\tSolution: " << endl;
        PrintSolution(solver);
        break;
      }
      else
        solver.out() << "\tNo solution! " << endl;
    }
    solver.out() << endl << "----------------------------------------" << endl;
    solver.out() << "Statistics: " << endl;
    solver.printInformation();
    env.end();
  } catch (IloException& exc) {
    cout << exc << endl;
  }

  return 0;
}
```

```
////////////////////////////////////////////////////////////////////////////
//
// RESULTS
//
////////////////////////////////////////////////////////////////////////////
/*

  ----------------------------------------
  First try: targeted due dates
  No solution!

  ----------------------------------------
  Second try: relaxed capacities
  current value of makespan: 60
  Overlapping activities:
  V1 [44..45 -- 15 --> 59..60] and V2 [50 -- 10 --> 60] overlap
  T1 [32..33 -- 12 --> 44..45] and T5 [38 -- 12 --> 50] overlap
  M2 [12..14 -- 8 --> 20..22] and M6 [18 -- 20 --> 38] overlap
  M1 [16..17 -- 16 --> 32..33] and M6 [18 -- 20 --> 38] overlap
  M1 [16..17 -- 16 --> 32..33] and M2 [12..14 -- 8 --> 20..22] overlap
  S4 [15 -- 4 --> 19] and S6 [6 -- 10 --> 16] overlap
  S2 [6 -- 4 --> 10] and S6 [6 -- 10 --> 16] overlap
  S1 [6..7 -- 8 --> 14..15] and S6 [6 -- 10 --> 16] overlap
  S1 [6..7 -- 8 --> 14..15] and S2 [6 -- 4 --> 10] overlap
  P1 [2..14 -- 20 --> 22..34] and P2 [2 -- 13 --> 15] overlap
  A4 [0 -- 2 --> 2] and A6 [0..1 -- 5 --> 5..6] overlap
  A1 [0..3 -- 4 --> 4..7] and A6 [0..1 -- 5 --> 5..6] overlap

  ----------------------------------------
  Third try: some due dates are incrementally removed

  Pass number 1
        Remove [S2 ends before 10]
        Remove [B2 ends before 11]
        Remove [B4 ends before 20]
        Remove [L ends before 32]
        No solution!

  Pass number 2
        Remove [AB2 ends before 13]
        Remove [M1 ends before 33]
        Remove [V1 ends before 60]
        No solution!

  Pass number 3
        Remove [A2 ends before 5]
        Remove [P2 ends before 17]
        Remove [M2 ends before 22]
        Remove [UE ends before 12]
        No solution!

  Pass number 4
        No solution!

  Pass number 5
        Remove [A1 ends before 8]
        Remove [S1 ends before 18]
```

```
      Remove [S4 ends before 23]
      Remove [B1 ends before 19]
      Remove [AB1 ends before 20]
      No solution!

  Pass number 6
      Remove [AB4 ends before 26]
      Remove [T1 ends before 49]
      current value of makespan: 104
      Solution:
  A1 [4..6 -- 4 --> 8..10]
  A2 [2..4 -- 2 --> 4..6]
  A3 [0..1 -- 2 --> 2..3]
  A4 [8..10 -- 2 --> 10..12]
  A5 [13..16 -- 2 --> 15..18]
  A6 [18..19 -- 5 --> 23..24]
  P1 [2..3 -- 20 --> 22..23]
  P2 [22..25 -- 13 --> 35..38]  <- violated due date: 17
  S1 [10 -- 8 --> 18]
  S2 [6 -- 4 --> 10]
  S3 [22..23 -- 4 --> 26..27]
  S4 [36..38 -- 4 --> 40..42]  <- violated due date: 23
  S5 [18..19 -- 4 --> 22..23]
  S6 [26..27 -- 10 --> 36..37]
  B1 [18 -- 1 --> 19]
  B2 [10 -- 1 --> 11]
  B3 [26..30 -- 1 --> 27..31]
  B4 [40..42 -- 1 --> 41..43]  <- violated due date: 20
  B5 [22..26 -- 1 --> 23..27]
  B6 [36..37 -- 1 --> 37..38]
  AB1 [19 -- 1 --> 20]
  AB2 [11 -- 1 --> 12]
  AB3 [27..42 -- 1 --> 28..43]
  AB4 [41..43 -- 1 --> 42..44]  <- violated due date: 26
  AB5 [23..27 -- 1 --> 24..28]
  AB6 [37..38 -- 1 --> 38..39]
  M1 [20 -- 16 --> 36]  <- violated due date: 33
  M2 [12 -- 8 --> 20]
  M3 [52 -- 8 --> 60]
  M4 [44 -- 8 --> 52]
  M5 [36 -- 8 --> 44]
  M6 [60 -- 20 --> 80]
  T1 [36..41 -- 12 --> 48..53]
  T2 [92 -- 12 --> 104]
  T3 [64..65 -- 12 --> 76..77]
  T4 [52..53 -- 12 --> 64..65]
  T5 [80 -- 12 --> 92]
  V1 [48..77 -- 15 --> 63..92]  <- violated due date: 60
  V2 [92 -- 10 --> 102]
  PE [104 -- 0 --> 104]
  L [30..39 -- 2 --> 32..41]
  UE [0 -- 10 --> 10]
  UA [78..94 -- 10 --> 88..104]
*/
```

C H A P T E R

32

Extending Transition Cost Usage

The example in Chapter 21, *Using Transition Times and Costs*, defined transition cost as the energy required by the maintenance operation (that is, the product between the transition time and the transition capacity). In this example, the maintenance operations require a discrete capacity resource. The transition cost is defined as the product of the transition time between activities and the transition capacity of the discrete resource.

Describing the Problem

The problem is another extension of the job-shop problem. In this problem, a team of workers is needed to perform a maintenance operation between each consecutive pair of activities on a machine. These operations depend only on the activity pair. We assume that all machines are identical. The objective is to minimize the makespan.

Defining the Jobs and Activities

The activities of the jobs are of known processing time and require a predefined machine. The jobs also define the temporal constraints between the activities. The temporal constraints are of type `IloActivity::startsAfterEnd`.

```
const IloInt Horizon = 10000;
const IloInt NumberOfJobs = 6;
const IloInt NumberOfResources = 6;
```

```
const IloInt NumberOfWorkers = 5;
const IloInt NumberOfTypes = 4;

IloInt ResourceNumbers [] = {2, 0, 1, 3, 5, 4,
                             1, 2, 4, 5, 0, 3,
                             2, 3, 5, 0, 1, 4,
                             1, 0, 2, 3, 4, 5,
                             2, 1, 4, 5, 0, 3,
                             1, 3, 5, 0, 4, 2};

IloInt ProcessingTimes [] = { 10,  30,  60,  70,  30,  60,
                              80,  50, 100, 100, 100,  40,
                              50,  40,  80,  90,  10,  70,
                              50,  50,  50,  30,  80,  90,
                              90,  30,  50,  40,  30,  10,
                              30,  30,  90, 100,  40,  10};
```

Each of these activities has a transition type.

```
IloInt Types [] = { 0, 1, 2, 2, 0, 3,
                    1, 3, 2, 3, 3, 0,
                    2, 3, 1, 1, 0, 1,
                    2, 1, 0, 0, 1, 1,
                    3, 3, 2, 0, 1, 1,
                    3, 1, 2, 0, 3, 3};
```

Creating Transition Tables

The team of workers is modeled as a discrete resource. After each activity and before the first activity on a machine, a maintenance operation (called the setup operation when it is performed before the first activity on a machine) that requires the workers is executed according to the sequence on the machines.

Several tables are needed in the model. The first one defines the duration of the maintenance operations. This table is used to build both the maintenance operations as a transition cost and the transition time between a pair of activities on a machine. An array of integers is given for the duration of the setup operation on each machine.

```
IloInt TableDurations [] = { 0, 16, 14, 19,
                             12, 0, 8, 15,
                             15, 10, 0, 14,
                             16, 15, 17, 0};

IloInt SetupDurations [] = { 7, 0, 6, 7};
```

A transition cost table and a setup cost array are also defined for the number of workers that are required for a maintenance or setup operation. We call this the transition capacity.

```
IloInt TableCapacities [] = { 0, 2, 3, 1,
                              2, 0, 4, 2,
                              3, 3, 0, 2,
                              1, 2, 3, 0};

IloInt SetupCapacities [] = { 2, 1, 1, 1};
```

As a heuristic tie breaker for selecting the successor of an activity on a machine, we use a cost corresponding to the energy required by the maintenance activity: the product between the transition time and the transition capacity (see Solving the Problem). The transition parameters are initialized as follows:

```
  enerTParam = IloTransitionParam(env, numberOfTypes);
  IloTransitionParam durTParam(env, numberOfTypes);
  IloTransitionParam capTParam(env, numberOfTypes);
  for (i = 0; i < numberOfTypes; ++i) {
    durTParam.setSetup(i, setupDurations[i]);
    capTParam.setSetup(i, setupCapacities[i]);
    enerTParam.setSetup(i, setupDurations[i] * setupCapacities[i]);
    IloInt index = i * numberOfTypes;
    for(j = 0 ; j < numberOfTypes; ++j) {
      durTParam.setValue(i, j, tableDurations[index]);
      capTParam.setValue(i, j, tableCapacities[index]);
      enerTParam.setValue(i, j, tableDurations[index] *
                          tableCapacities[index]);
      ++index;
    }
  }
```

Setting Temporal Constraints

A maintenance operation begins after the end time of its corresponding job activity and ends before the start time of the following activity on its machine. We calculate the start time of the following activity using a variable transition cost on the machine. This variable transition cost is defined in Concert Technology using the macro `ILOTRANSITIONCOSTOBJECT`. This macro allows the definition and extraction of a variable transaction cost in Concert. First of all, `StartTCostObjectIlcI`, a derived class of `IlcTransitionCostObjectI`, is created for this purpose. An instance of this class is used during the search to calculate the transition costs. The macro `ILOTRANSITIONCOSTOBJECT` defines a Concert Technology variable transition cost class, `StartTCostObject`, that will be extracted into an instance of `StartTCostObjectIlcI`. Notice that the chosen teardown cost ensures that the last maintenance operation ends before the horizon of the schedule.

```
class StartTCostObjectIlcI : public IlcTransitionCostObjectI {
public:
  StartTCostObjectIlcI();
  ~StartTCostObjectIlcI();
  virtual IlcInt getTransitionCostMax(const IlcResourceConstraint srct1,
```

```
                                        const IlcResourceConstraint srct2)
    const;
  virtual IlcInt getTransitionCostMin(const IlcResourceConstraint srct1,
                                        const IlcResourceConstraint srct2)
    const;
  virtual IlcInt getSetupCostMin(const IlcResourceConstraint srct1) const;
  virtual IlcInt getSetupCostMax(const IlcResourceConstraint srct1) const;
  virtual IlcInt getTeardownCostMin(const IlcResourceConstraint srct1)
    const;
  virtual IlcInt getTeardownCostMax(const IlcResourceConstraint srct1)
    const;
};

StartTCostObjectIlcI::StartTCostObjectIlcI()
  :IlcTransitionCostObjectI(IlcTrue)
{}

StartTCostObjectIlcI::~StartTCostObjectIlcI() {}

IlcInt
StartTCostObjectIlcI::getTransitionCostMax(const IlcResourceConstraint,
                                           const IlcResourceConstraint srct2)
  const {
  return srct2.getActivity().getStartMax();
}

IlcInt
StartTCostObjectIlcI::getTransitionCostMin(const IlcResourceConstraint,
                                           const IlcResourceConstraint srct2)
  const {
  return srct2.getActivity().getStartMin();
}

IlcInt
StartTCostObjectIlcI::getSetupCostMin(const IlcResourceConstraint srct1)
  const {
  return srct1.getActivity().getStartMin();
}

IlcInt
StartTCostObjectIlcI::getSetupCostMax(const IlcResourceConstraint srct1)
  const {
  return srct1.getActivity().getStartMax();
}

IlcInt
StartTCostObjectIlcI::getTeardownCostMin(const IlcResourceConstraint srct1)
  const {
  return srct1.getActivity().getEndMin();
}

IlcInt
StartTCostObjectIlcI::getTeardownCostMax(const IlcResourceConstraint srct1)
  const {
  return srct1.getActivity().getEndMax();
}

ILOTRANSITIONCOSTOBJECT0(StartTCostObject, solver) {
```

```
  return new (solver.getHeap()) StartTCostObjectIlcI();
}
  IloTransitionCostObject startTCostObj =
    StartTCostObject(env);
```

Defining Machines

A machine is created as a unary resource with a capacity enforcement level of `IloHigh`.

```
IloUnaryResource
MakeMachine(IloModel model, const char* name) {
  IloUnaryResource machine(model.getEnv(), name);
  machine.setCapacityEnforcement(IloHigh);
  return machine;
}
```

Creating the Maintenance Operations

Maintenance operations require the discrete resource representing the workers. The durations, required capacities, and precedence constraints are defined from the transition cost variable on the machines.

Two kinds of activities are created. One for the maintenance operation after an activity on a machine; the other for the setup operation on a machine.

```
IloActivity
MakeMaintenanceTask(IloModel model,
                    IloResourceConstraint rct,
                    IloDiscreteResource workers,
                    IloTransitionCost durTCost,
                    IloTransitionCost capTCost,
                    const char* name) {
  IloEnv env = model.getEnv();
  IloNumVar ptVar(env, 0, IloInfinity, ILOINT);
  IloActivity act(env, ptVar);
  model.add(ptVar == durTCost.getNextCostExpr(rct));
  act.setName(name);
  model.add(act.startsAfterEnd(rct.getActivity()));
  IloNumVar capVar(env, 0, IloInfinity, ILOINT);
  model.add(act.requires(workers, capVar));
  model.add(capVar == capTCost.getNextCostExpr(rct));
  // Mark maintenance activity
  act.setObject((IloAny)1);
  return act;
}

IloActivity
MakeSetupTask(IloModel model,
              IloDiscreteResource workers,
              IloTransitionCost durTCost,
              IloTransitionCost capTCost,
              const char* name) {
  IloEnv env = model.getEnv();
```

```
  IloNumVar ptVar(env, 0, IloInfinity, ILOINT);
  IloActivity act(env, ptVar);
  model.add(ptVar == durTCost.getSetupCostExpr());
  act.setName(name);
  // Mark maintenance activity
  act.setObject((IloAny)1);
  IloNumVar capVar(env, 0, IloInfinity, ILOINT);
  model.add(act.requires(workers, capVar));
  model.add(capVar == capTCost.getSetupCostExpr());
  return act;
}
```

Solving the Problem

To search for a solution, we must sequence each machine. The goal `IlcSequence` is provided by Scheduler Engine for this purpose. We can guess that having a minimal makespan is closely related to having the smallest possible transition costs and times. That's why the selector of the next activity on a partial sequence uses the minimal energy of the maintenance team workers' resource as tie breaker. Precisely, we select the successor with minimal earliest start time, then minimal transition energy, then minimal latest end time.

The order of execution of the activities on a unary resource suffices to make a solution explicit. This is not true on a discrete resource for which the start time of each activity must be assigned. For that purpose, we use the `IlcSetTimes` goal with a selector on the activities requiring the discrete resource, that is, the maintenance activities.

```
/* CREATE A SELECTOR FOR THE ACTIVITIES ON THE WORKERS RESOURCE. */

ILOPREDICATE0(MaintenanceTaskPredicate, IlcActivity, act) {
  return (act.getObject() != 0);
}

/* CREATE SEARCH GOAL. */
ILCGOAL2(SolveIlc,
         IlcIntVar, makespan,
         IlcTransitionCostObject, enerTCostObjIlc) {
  IloSolver solver = getSolver();
  IlcScheduler scheduler (solver);

  IloPredicate<IlcActivity> predA = !IlcActivityStartVarBoundPredicate(solver)
    && !IlcActivityPostponedPredicate(solver)
    && MaintenanceTaskPredicate(solver);
  IloComparator<IlcActivity> compA =

IloComposeLexical(IlcActivityStartMinEvaluator(solver).makeLessThanComparator()
,
                  IlcActivityEndMaxEvaluator(solver).makeLessThanComparator());

  IloSelector<IlcActivity,IlcSchedule> maintenanceTaskSelector =
    IloBestSelector<IlcActivity,IlcSchedule>(predA, compA);

  IloEvaluator<IlcResourceConstraint> eval =
```

```
    IlcResourceConstraintNextTransitionCostEvaluator(solver,enerTCostObjIlc);
  IloTranslator<IlcActivity, IlcResourceConstraint> ac =
    IlcActivityResourceConstraintTranslator(solver);

  IloSelector<IlcResourceConstraint,IlcResourceConstraint> selector =
    IloBestSelector<IlcResourceConstraint,IlcResourceConstraint>(
       IloComposeLexical((IlcActivityStartMinEvaluator(solver) <<
ac).makeLessThanComparator(),
                                                    eval.makeLessThanComparator(),
         (IlcActivityEndMaxEvaluator(solver) << ac).makeLessThanComparator()));

  return IlcAnd(IlcSequence(scheduler, selector),
                IlcSetTimes(scheduler, makespan,
                            maintenanceTaskSelector));
}

ILOCPGOALWRAPPER2(Solve, solver,
                  IloNumVar, makespan,
                  IloTransitionParam, enerTParam) {
  IlcScheduler scheduler(solver);
  return SolveIlc(solver,
                  solver.getIntVar(makespan),
                  scheduler.getTransitionCostObject(enerTParam));
}
```

Complete Program and Output

You can see the entire program `maintain.cpp` here or view it in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

#if defined(ILO_SDXLOUTPUT)
#include “sdxloutput.h”
#endif

const IloInt Horizon = 10000;
const IloInt NumberOfJobs = 6;
const IloInt NumberOfResources = 6;
const IloInt NumberOfWorkers = 5;
const IloInt NumberOfTypes = 4;

IloInt ResourceNumbers [] = {2, 0, 1, 3, 5, 4,
                             1, 2, 4, 5, 0, 3,
                             2, 3, 5, 0, 1, 4,
                             1, 0, 2, 3, 4, 5,
                             2, 1, 4, 5, 0, 3,
                             1, 3, 5, 0, 4, 2};

IloInt ProcessingTimes [] = { 10,  30,  60,  70,  30,  60,
                              80,  50, 100, 100, 100,  40,
                              50,  40,  80,  90,  10,  70,
                              50,  50,  50,  30,  80,  90,
```

```
                              90,  30,  50,  40,  30,  10,
                              30,  30,  90, 100,  40,  10};

IloInt Types [] = { 0, 1, 2, 2, 0, 3,
                    1, 3, 2, 3, 3, 0,
                    2, 3, 1, 1, 0, 1,
                    2, 1, 0, 0, 1, 1,
                    3, 3, 2, 0, 1, 1,
                    3, 1, 2, 0, 3, 3};

IloInt TableDurations [] = { 0, 16, 14, 19,
                             12, 0, 8, 15,
                             15, 10, 0, 14,
                             16, 15, 17, 0};

IloInt SetupDurations [] = { 7, 0, 6, 7};

IloInt TableCapacities [] = { 0, 2, 3, 1,
                              2, 0, 4, 2,
                              3, 3, 0, 2,
                              1, 2, 3, 0};

IloInt SetupCapacities [] = { 2, 1, 1, 1};

////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////

class StartTCostObjectIlcI : public IlcTransitionCostObjectI {
public:
  StartTCostObjectIlcI();
  ~StartTCostObjectIlcI();
  virtual IlcInt getTransitionCostMax(const IlcResourceConstraint srct1,
                                      const IlcResourceConstraint srct2)
    const;
  virtual IlcInt getTransitionCostMin(const IlcResourceConstraint srct1,
                                      const IlcResourceConstraint srct2)
    const;
  virtual IlcInt getSetupCostMin(const IlcResourceConstraint srct1) const;
  virtual IlcInt getSetupCostMax(const IlcResourceConstraint srct1) const;
  virtual IlcInt getTeardownCostMin(const IlcResourceConstraint srct1)
    const;
  virtual IlcInt getTeardownCostMax(const IlcResourceConstraint srct1)
    const;
};

StartTCostObjectIlcI::StartTCostObjectIlcI()
  :IlcTransitionCostObjectI(IlcTrue)
{}

StartTCostObjectIlcI::~StartTCostObjectIlcI() {}

IlcInt
StartTCostObjectIlcI::getTransitionCostMax(const IlcResourceConstraint,
                                           const IlcResourceConstraint srct2)
  const {
```

```
  return srct2.getActivity().getStartMax();
}

IlcInt
StartTCostObjectIlcI::getTransitionCostMin(const IlcResourceConstraint,
                                           const IlcResourceConstraint srct2)
  const {
  return srct2.getActivity().getStartMin();
}

IlcInt
StartTCostObjectIlcI::getSetupCostMin(const IlcResourceConstraint srct1)
  const {
  return srct1.getActivity().getStartMin();
}

IlcInt
StartTCostObjectIlcI::getSetupCostMax(const IlcResourceConstraint srct1)
  const {
  return srct1.getActivity().getStartMax();
}

IlcInt
StartTCostObjectIlcI::getTeardownCostMin(const IlcResourceConstraint srct1)
  const {
  return srct1.getActivity().getEndMin();
}

IlcInt
StartTCostObjectIlcI::getTeardownCostMax(const IlcResourceConstraint srct1)
  const {
  return srct1.getActivity().getEndMax();
}

ILOTRANSITIONCOSTOBJECT0(StartTCostObject, solver) {
  return new (solver.getHeap()) StartTCostObjectIlcI();
}

/* CREATION OF A MACHINE */
IloUnaryResource
MakeMachine(IloModel model, const char* name) {
  IloUnaryResource machine(model.getEnv(), name);
  machine.setCapacityEnforcement(IloHigh);
  return machine;
}

/* CREATION OF A TASK */
IloActivity
MakeTask(IloModel model,
         IloUnaryResource machine,
         IloInt type,
         IloInt procTime,
         IloInt setupTime,
         const char* name,
         IloResourceConstraint& rct) {
  IloActivity act(model.getEnv(), procTime, type, name);
  act.setStartMin(setupTime);
  rct = act.requires(machine);
```

```
  model.add(rct);
  return act;
}

IloActivity
MakeMaintenanceTask(IloModel model,
                    IloResourceConstraint rct,
                    IloDiscreteResource workers,
                    IloTransitionCost durTCost,
                    IloTransitionCost capTCost,
                    const char* name) {
  IloEnv env = model.getEnv();
  IloNumVar ptVar(env, 0, IloInfinity, ILOINT);
  IloActivity act(env, ptVar);
  model.add(ptVar == durTCost.getNextCostExpr(rct));
  act.setName(name);
  model.add(act.startsAfterEnd(rct.getActivity()));
  IloNumVar capVar(env, 0, IloInfinity, ILOINT);
  model.add(act.requires(workers, capVar));
  model.add(capVar == capTCost.getNextCostExpr(rct));
  // Mark maintenance activity
  act.setObject((IloAny)1);
  return act;
}

IloActivity
MakeSetupTask(IloModel model,
              IloDiscreteResource workers,
              IloTransitionCost durTCost,
              IloTransitionCost capTCost,
              const char* name) {
  IloEnv env = model.getEnv();
  IloNumVar ptVar(env, 0, IloInfinity, ILOINT);
  IloActivity act(env, ptVar);
  model.add(ptVar == durTCost.getSetupCostExpr());
  act.setName(name);
  // Mark maintenance activity
  act.setObject((IloAny)1);
  IloNumVar capVar(env, 0, IloInfinity, ILOINT);
  model.add(act.requires(workers, capVar));
  model.add(capVar == capTCost.getSetupCostExpr());
  return act;
}

void DefineProblem(IloModel model,
                   IloInt horizon,
                   IloInt numberOfJobs,
                   IloInt numberOfResources,
                   IloInt numberOfWorkers,
                   IloInt numberOfTypes,
                   IloInt* resourceNumbers,
                   IloInt* processingTimes,
                   IloInt* types,
                   IloInt* tableDurations,
                   IloInt* tableCapacities,
                   IloInt* setupDurations,
                   IloInt* setupCapacities,
                   IloNumVar& makespan,
```

```
                IloTransitionParam& enerTParam,
                IloDiscreteResource& workers) {
/* CREATE THE MAKESPAN VARIABLE. */
IloEnv env = model.getEnv();
makespan = IloNumVar(env, 0, horizon, IloNumVar::Int);

IloUnaryResource* machines =
  new (env) IloUnaryResource[NumberOfResources];

char buffer[128];
IloInt i, j, k;

/* CREATE THE TRANSITION FUNCTIONS. */
enerTParam = IloTransitionParam(env, numberOfTypes);
IloTransitionParam durTParam(env, numberOfTypes);
IloTransitionParam capTParam(env, numberOfTypes);
IloTransitionCostObject startTCostObj =
  StartTCostObject(env);

for (i = 0; i < numberOfTypes; ++i) {
  durTParam.setSetup(i, setupDurations[i]);
  capTParam.setSetup(i, setupCapacities[i]);
  enerTParam.setSetup(i, setupDurations[i] * setupCapacities[i]);
  IloInt index = i * numberOfTypes;
  for(j = 0 ; j < numberOfTypes; ++j) {
    durTParam.setValue(i, j, tableDurations[index]);
    capTParam.setValue(i, j, tableCapacities[index]);
    enerTParam.setValue(i, j, tableDurations[index] *
                        tableCapacities[index]);
    ++index;
  }
}

/* CREATE THE WORKERS RESOURCE. */
workers = IloDiscreteResource(env, numberOfWorkers, "Workers");

/* CREATE THE RESOURCES AND SETUP TASKS. */
IloTransitionCost* costs =
  new (env) IloTransitionCost[3 * NumberOfResources];
for (j = 0; j < numberOfResources; j++) {
  sprintf(buffer, "Machine%ld", j);
  machines[j] = MakeMachine(model, buffer);
  IloTransitionTime durTTime(machines[j], durTParam);
  costs[j] =
    IloTransitionCost(machines[j], durTParam);
  costs[j+NumberOfResources] =
    IloTransitionCost(machines[j], capTParam);
  costs[j+2*NumberOfResources] =
    IloTransitionCost(machines[j], startTCostObj);

  sprintf(buffer, "Setup%ld", j);
  IloActivity stask = MakeSetupTask(model,
                                    workers,
                                    costs[j],
                                    costs[j+NumberOfResources],
                                    buffer);
  IloTransitionCost cost = costs[j+2*numberOfResources];
  IloNumVar delayVar(env, 0, IloInfinity, ILOINT);
```

```
    model.add(stask.endsBefore(delayVar));
    model.add(delayVar == cost.getSetupCostExpr());
  }

  /* CREATE THE ACTIVITIES, THE TEMPORAL CONSTRAINTS
     AND MAINTENANCE TASKS. */
  k = 0;
  for (i = 0; i < numberOfJobs; i++) {
    IloActivity previousTask;
    for (j = 0; j < numberOfResources; j++) {
      IloInt number = resourceNumbers[k];
      sprintf(buffer, "J%ldS%ldR%ldT%ld", i, j, number, types[k]);
      IloResourceConstraint rct;
      IloActivity task = MakeTask(model,
                                  machines[number],
                                  types[k],
                                  processingTimes[k],
                                  setupDurations[types[k]],
                                  buffer,
                                  rct);
      if (0 != j)
        model.add(task.startsAfterEnd(previousTask));
      previousTask = task;

      IloActivity mtask = MakeMaintenanceTask(model,
                                              rct,
                                              workers,
                                              costs[number],
                                              costs[number+numberOfResources],
                                              buffer);

      IloTransitionCost cost = costs[number+2*numberOfResources];
      IloNumVar delayVar(env, 0, IloInfinity, ILOINT);
      model.add(mtask.endsBefore(delayVar));
      model.add(delayVar == cost.getNextCostExpr(rct));
      k++;
    }
    model.add(previousTask.endsBefore(makespan));;
  }

  model.add(IloMinimize(env, makespan));
}

////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTION
//
////////////////////////////////////////////////////////////////////////////

void PrintSolution(IloSolver& solver,
                   IloDiscreteResource workers) {
  IlcScheduler scheduler(solver);
  for(IlcUnaryResourceIterator iter(scheduler); iter.ok(); ++iter) {
    IlcUnaryResource machine = *iter;
    solver.out() << endl << "Machine : " << machine << endl;
    IlcInt index = machine.getSetupVar().getValue();
    IlcInt sink = machine.getTeardownIndex();
    while(index != sink) {
```

```
      IlcResourceConstraint rct = machine.getSequenceRC(index);
      solver.out() << "\t" << rct.getActivity() << endl;
      index = rct.getNextVar().getValue();
    }
  }
  solver.out() << endl;
  solver.out() << workers << endl;
  for(IlcResourceConstraintIterator iter2(scheduler.getResource(workers));
      iter2.ok(); ++iter2) {
    solver.out() <<  "\t" << (*iter2).getActivity() << endl;
  }
}

////////////////////////////////////////////////////////////////////////////
//
// SOLVING PROBLEM
//
////////////////////////////////////////////////////////////////////////////

/* CREATE A SELECTOR FOR THE ACTIVITIES ON THE WORKERS RESOURCE. */

ILOPREDICATE0(MaintenanceTaskPredicate, IlcActivity, act) {
  return (act.getObject() != 0);
}

/* CREATE SEARCH GOAL. */
ILCGOAL2(SolveIlc,
         IlcIntVar, makespan,
         IlcTransitionCostObject, enerTCostObjIlc) {
  IloSolver solver = getSolver();
  IlcScheduler scheduler (solver);

  IloPredicate<IlcActivity> predA = !IlcActivityStartVarBoundPredicate(solver)
    && !IlcActivityPostponedPredicate(solver)
    && MaintenanceTaskPredicate(solver);
  IloComparator<IlcActivity> compA =

IloComposeLexical(IlcActivityStartMinEvaluator(solver).makeLessThanComparator()
,
                  IlcActivityEndMaxEvaluator(solver).makeLessThanComparator());

  IloSelector<IlcActivity,IlcSchedule> maintenanceTaskSelector =
    IloBestSelector<IlcActivity,IlcSchedule>(predA, compA);

  IloEvaluator<IlcResourceConstraint> eval =
    IlcResourceConstraintNextTransitionCostEvaluator(solver,enerTCostObjIlc);
  IloTranslator<IlcActivity, IlcResourceConstraint> ac =
    IlcActivityResourceConstraintTranslator(solver);

  IloSelector<IlcResourceConstraint,IlcResourceConstraint> selector =
    IloBestSelector<IlcResourceConstraint,IlcResourceConstraint>(
       IloComposeLexical((IlcActivityStartMinEvaluator(solver) <<
ac).makeLessThanComparator(),
                                                  eval.makeLessThanComparator(),
         (IlcActivityEndMaxEvaluator(solver) << ac).makeLessThanComparator()));

  return IlcAnd(IlcSequence(scheduler, selector),
                IlcSetTimes(scheduler, makespan,
```

```
                              maintenanceTaskSelector));
}

ILOCPGOALWRAPPER2(Solve, solver,
                  IloNumVar, makespan,
                  IloTransitionParam, enerTParam) {
  IlcScheduler scheduler(solver);
  return SolveIlc(solver,
                  solver.getIntVar(makespan),
                  scheduler.getTransitionCostObject(enerTParam));
}

int main() {
  try {
    IloEnv env;
    IloModel model(env);
    IloNumVar makespan;
    IloTransitionParam enerTParam;
    IloDiscreteResource workers;
    DefineProblem(model,
                  Horizon,
                  NumberOfJobs,
                  NumberOfResources,
                  NumberOfWorkers,
                  NumberOfTypes,
                  ResourceNumbers,
                  ProcessingTimes,
                  Types,
                  TableDurations,
                  TableCapacities,
                  SetupDurations,
                  SetupCapacities,
                  makespan,
                  enerTParam,
                  workers);

    IloSolver solver(model);
    if (solver.solve(Solve(env, makespan, enerTParam))) {

      env.out() << " Solution with makespan : "
                << solver.getIntVar(makespan) << endl;
      PrintSolution(solver, workers);

#if defined(ILO_SDXLOUTPUT)
      IloSDXLOutput output(env);
      ofstream outFile("maintain.xml");
      output.write(IlcScheduler(solver), outFile, solver.getIntVar(makespan));
      outFile.close();
#endif
    } else
      env.out() << "No solution " << endl;
    env.end();
  } catch (IloException& exc) {
    cout << exc << endl;
  }
  return 0;
}
```

C H A P T E R

33

Scheduling with State Resources: the Trolley Problem, with Transition Times

The problem we want to solve here is the same as the one introduced in Chapter 23, *Scheduling with State Resources: the Trolley Problem*, except that we now take into account the time taken by the trolley to go from one area to another. This example demonstrates the use of transition times on state resources.

Describing the Problem

The time taken by the trolley to move from one area to another is shown in Figure 33.1.

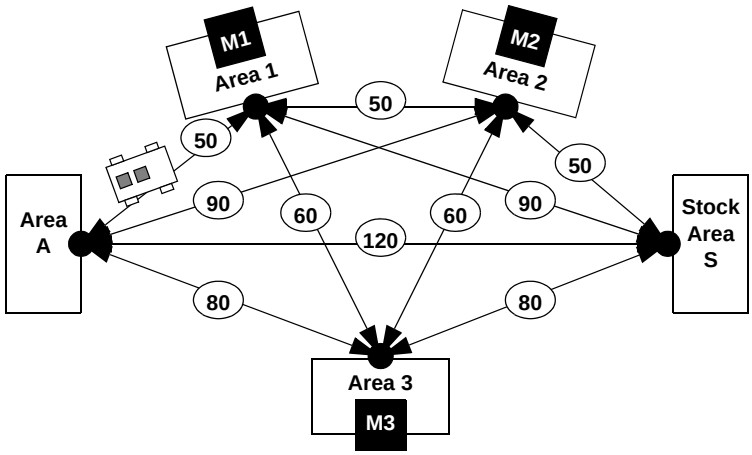


Figure 33.1 *Machines, Areas, and Durations for the Trolley Problem with Transition Times*

We define an array, `TransitDurations`, to represent the trolley transition times.

```
                                  // M1   M2   M3   A     S
IloNum TransitDurations[] = {      0,  50,  60,  50,   90,    // M1
                                  50,   0,  60,  90,   50,    // M2
                                  60,  60,  0,   80,   80,    // M3
                                  50,  90,  80,   0,  120,    // A
                                  90,  50,  80, 120,    0   };// S
```

A function, `getMaxTransitDuration`, is defined on this array. It returns the longest transit duration between two areas of the shop.

```
IloNum getMaxTransitDuration(IloInt nbArea, IloNum* transitDurations){
  IloNum max = 0;
  IloInt size = nbArea*nbArea;
  for (IloInt i = 0; i < size; i++)
    if (transitDurations[i] > max) max = transitDurations[i];
  return max;
}
```

Now, the transit duration array is added to the function `DefineModel` as an extra argument.

```
IloEnv env;
IloAnyArray jobs;
IloStateResource trolley;
IloArray<IloUnaryResource> machines;
IloNumVar makespan;
IloModel model = DefineModel(env,
    JobNames,
    TransitDurations,
    Machines1, Durations1,
    Machines2, Durations2,
    jobs,
    trolley,
    machines,
    makespan);
```

In the next section we show how the `DefineModel` function is modified to take transition times into account.

Defining the Problem, Designing a Model

In this section we discuss setting the time horizon for the schedule and the creation of a state resource with transition times.

Schedule

The *time horizon* for the schedule is computed using the assumptions that the jobs are executed one after another and that the trolley always needs the maximum duration to move from one place to another.

```
IloModel DefineModel(IloEnv env,
                     const char** jobNames,
                     IloNum*      transitDurations,
                     IloInt*      machines1,
                     IloNum*      durations1,
                     IloInt*      machines2,
                     IloNum*      durations2,
                     IloAnyArray&  jobs,
                     IloStateResource& trolley,
                     IloArray<IloUnaryResource>& machines,
                     IloNumVar&   makespan)
{
  IloInt i, j;
  IloModel model(env);

  /* CREATE THE MAKESPAN VARIABLE. */
  IloNum horizon =
    numberOfJobs * ((6 * loadDuration) +
                    (4 * getMaxTransitDuration(5, transitDurations)));
```

```
  for (i = 0; i < numberOfJobs; i++)
    horizon += durations1[i] + durations2[i];
  makespan = IloNumVar(env,0,horizon,ILOINT);
 /* ... */
 /* RETURN THE CREATED MODEL */
  return model;
}
```

Resources

Scheduler provides a way to create a *state resource* with *transition times*, just as with unary resources (Chapter 20). The following code creates a symmetrical transition time table and associates it with the state resource `trolley`. The enforcement level `IloBasic` means that the scheduler will spend a basic amount of effort (the default) at enforcing the applicable constraints; `IloMediumHigh` specifies that more effort be expended.

```
  /* CREATE THE TROLLEY POSITION RESOURCE */
  IloAnyArray arrayOfPossibleAreas(env,
                                   5,
                                   (IloAny)AREAA,
                                   (IloAny)MACH1,
                                   (IloAny)MACH2,
                                   (IloAny)MACH3,
                                   (IloAny)AREAS);
  IloAnySet   setOfPossibleAreas(env,arrayOfPossibleAreas);
  trolley = IloStateResource(env,setOfPossibleAreas, “trolley”);

  /* CREATE THE (SYMMETRIC) TRANSITION TIME TABLE */
  IloTransitionParam transitionParam(env, 5, IloTrue);
  for (i = 0; i < 5; i++)
    for (j = i; j < 5; j++)
      transitionParam.setValue(i,j,transitDurations[j + (5 * i)]);

  /* ASSOCIATE A TRANSITION TIME TO THE RESOURCE */
  IloTransitionTime transitionTime(trolley,transitionParam);

  /* SELECT APPROPRIATE ENFORCEMENT LEVEL */
  trolley.setTransitionTimeEnforcement(IloBasic);
  trolley.setCapacityEnforcement(IloMediumHigh);
```

The `Job` class remains the same as for the first version of the trolley example, except that we must define, for each activity that requires the state resource (load/unload activities), the transition type of the activity. The *transition type* of an activity is the index of the transition table that will be used to compute the transition time between two activities. In our example, the transition type of a load/unload activity depends on the area where the activity must take place.

The transition time between two activities is introduced to some of the `startsAfterEnd` constraints to increase propagation and improve the solution.

```
Job::Job(IloEnv env,
         const char*  name,
```

```
        IloNum loadDuration,
        IloUnaryResource machine1, IloNum duration1, IloInt area1,
        IloUnaryResource machine2, IloNum duration2, IloInt area2)
  : _name(name),
    _loadA(   env,loadDuration),
    _unload1( env,loadDuration),
    _process1(env,duration1),
    _load1(   env,loadDuration),
    _unload2( env,loadDuration),
    _process2(env,duration2),
    _load2(   env,loadDuration),
    _unloadS( env,loadDuration),
    _area1(area1),
    _area2(area2),
    _machine1(machine1),
    _machine2(machine2)
{
  char buffer[256];

  sprintf(buffer, "arrival_load_%s", name);
  _loadA.setName(buffer);

  sprintf(buffer, "area%ld_unload_%s", area1, name);
  _unload1.setName(buffer);

  sprintf(buffer, "machine%ld_%s", area1, name);
  _process1.setName(buffer);

  sprintf(buffer, "area%ld_load_%s", area1, name);
  _load1.setName(buffer);

  sprintf(buffer, "area%ld_unload_%s", area2, name);
  _unload2.setName(buffer);

  sprintf(buffer, "machine%ld_%s", area2, name);
  _process2.setName(buffer);

  sprintf(buffer, "area%ld_load_%s", area2, name);
  _load2.setName(buffer);

  sprintf(buffer, "stock_load_%s", name);
  _unloadS.setName(buffer);

  _loadA.setTransitionType(AREAA - 1);
  _unload1.setTransitionType(_area1 - 1);
  _load1.setTransitionType(_area1 - 1);
  _unload2.setTransitionType(_area2 - 1);
  _load2.setTransitionType(_area2 - 1);
  _unloadS.setTransitionType(AREAS - 1);
}

Job::~Job()
{}

void Job::addToModel(IloModel model,
                     IloStateResource trolley,
                     IloNumVar makespan,
                     IloTransitionParam transitionParam)
```

```
{
  /* ADD MACHINE REQUIREMENT CONSTRAINTS */
  model.add(_process1.requires(_machine1));
  model.add(_process2.requires(_machine2));

  /* ADD TEMPORAL CONSTRAINTS BETWEEN ACTIVITIES */
  IloNum delay = transitionParam.getValue((IloInt)_loadA.getTransitionType(),

(IloInt)_unload1.getTransitionType());
  model.add(_unload1.startsAfterEnd(_loadA, delay));
  model.add(_process1.startsAfterEnd(_unload1));
  model.add(_load1.startsAfterEnd(_process1));
  delay = transitionParam.getValue((IloInt)_load1.getTransitionType(),
                                   (IloInt)_unload2.getTransitionType());
  model.add(_unload2.startsAfterEnd(_load1, delay));
  model.add(_process2.startsAfterEnd(_unload2));
  model.add(_load2.startsAfterEnd(_process2));
  delay = transitionParam.getValue((IloInt)_load2.getTransitionType(),
                                   (IloInt)_unloadS.getTransitionType());
  model.add(_unloadS.startsAfterEnd(_load2, delay));

  /* ADD TROLLEY POSITION REQUIREMENTS  */
  model.add(_loadA.requires(trolley, (IloAny)AREAA));
  model.add(_unload1.requires(trolley, (IloAny)_area1));
  model.add(_load1.requires(trolley, (IloAny)_area1));
  model.add(_unload2.requires(trolley, (IloAny)_area2));
  model.add(_load2.requires(trolley, (IloAny)_area2));
  model.add(_unloadS.requires(trolley, (IloAny)AREAS));

  /* ADD MAKESPAN CONSTRAINT */
  model.add(_unloadS.endsBefore(makespan));
}
```

Solving the Problem

For this second version of the trolley example, we are interested in finding a “good solution” without a lot of computational effort.

This trade off between quality of the solution and computation time is achieved by limiting the number of fails at each step of the optimization process. When the limit is exceeded, the search is stopped and the last found schedule is returned. The maximal number of fails per optimization step is passed to the program as a parameter.

```
  IloInt failLimit       = 1000;
  if (argc > 1)
    failLimit = atol(argv[1]);
 /* ... */

    IloBool solved = IloFalse;
    IloNum  minMakespan = 0.;

    IloSolver solver(model);
```

```
IloGoal unlimitedGoal = IloSetTimesForward(env, makespan);
IloGoal limitedGoal = IloLimitSearch(env, unlimitedGoal,
                                     IloFailLimit(env,failLimit) );
solver.startNewSearch(limitedGoal);

while (solver.next()==IlcTrue) {
         solved = IloTrue;
  minMakespan = solver.getMin(makespan);
  solver.out() << "SOLUTION WITH MAKESPAN "
                     << minMakespan << endl;
}
solver.endSearch();

if (solved){
  makespan.setBounds(minMakespan,minMakespan);
  solver.solve(unlimitedGoal);
  PrintSolution(solver,jobs,trolley,machines,makespan);
} else {
  solver.out() << "NO SOLUTION FOUND" << endl;
}
```

Complete Program and Output

You can see the entire program `trolley2.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

const IloInt MACH1 = 1; // Machine 1 area
const IloInt MACH2 = 2; // Machine 2 area
const IloInt MACH3 = 3; // Machine 3 area
const IloInt AREAA = 4; // Arrival area
const IloInt AREAS = 5; // Stock area

const IloNum loadDuration     = 20;
const IloInt numberOfJobs     = 6;

const char* JobNames  [] = { "J1",   "J2",   "J3",   "J4",   "J5",   "J6"};
IloInt Machines1      [] = { 1,      2,      2,      1,      3,      2   };
IloNum Durations1     [] = { 80,     120,    80,     160,    180,    140 };
IloInt Machines2      [] = { 2,      3,      1,      3,      2,      3   };
IloNum Durations2     [] = { 60,     80,     60,     100,    80,     60  };

                              // M1   M2   M3   A     S
IloNum TransitDurations[] = {    0,  50,  60,  50,   90,    // M1
                                50,   0,  60,  90,   50,    // M2
                                60,  60,  0,   80,   80,    // M3
                                50,  90,  80,   0,  120,    // A
                                90,  50,  80, 120,    0   };// S

IloNum getMaxTransitDuration(IloInt nbArea, IloNum* transitDurations){
  IloNum max = 0;
```

```
  IloInt size = nbArea*nbArea;
  for (IloInt i = 0; i < size; i++)
    if (transitDurations[i] > max) max = transitDurations[i];
  return max;
}

////////////////////////////////////////////////////////////////////////////
//
// DEFINITION OF THE JOB CLASS
//
////////////////////////////////////////////////////////////////////////////

class Job {
public:
  const char* _name;
  IloActivity _loadA;
  IloActivity _unload1;
  IloActivity _process1;
  IloActivity _load1;
  IloActivity _unload2;
  IloActivity _process2;
  IloActivity _load2;
  IloActivity _unloadS;
  IloInt      _area1;
  IloInt      _area2;
  IloUnaryResource _machine1;
  IloUnaryResource _machine2;
public:
  Job(IloEnv env,
      const char*,
      IloNum loadDuration,
      IloUnaryResource machine1, IloNum duration1, IloInt area1,
      IloUnaryResource machine2, IloNum duration2, IloInt area2);
  void addToModel(IloModel model,
                  IloStateResource trolley,
                  IloNumVar makespan,
                  IloTransitionParam transitionParam);
  ~Job();
  void printSolution(IlcScheduler scheduler, ILOSTD(ostream)& out);
  const char* getName() const { return  _name;}

  IloActivity getLoadA() const { return _loadA;}
  IloActivity getUnload1() const { return _unload1;}
  IloActivity getProcess1() const { return _process1;}
  IloActivity getLoad1() const { return _load1;}
  IloActivity getUnload2() const { return _unload2;}
  IloActivity getProcess2() const { return _process2;}
  IloActivity getLoad2() const { return _load2;}
  IloActivity getUnloadS() const { return _unloadS;}

};


Job::Job(IloEnv env,
         const char*  name,
         IloNum loadDuration,
         IloUnaryResource machine1, IloNum duration1, IloInt area1,
         IloUnaryResource machine2, IloNum duration2, IloInt area2)
```

```
  : _name(name),
    _loadA(   env,loadDuration),
    _unload1( env,loadDuration),
    _process1(env,duration1),
    _load1(   env,loadDuration),
    _unload2( env,loadDuration),
    _process2(env,duration2),
    _load2(   env,loadDuration),
    _unloadS( env,loadDuration),
    _area1(area1),
    _area2(area2),
    _machine1(machine1),
    _machine2(machine2)
{
  char buffer[256];

  sprintf(buffer, “arrival_load_%s”, name);
  _loadA.setName(buffer);

  sprintf(buffer, “area%ld_unload_%s”, area1, name);
  _unload1.setName(buffer);

  sprintf(buffer, “machine%ld_%s”, area1, name);
  _process1.setName(buffer);

  sprintf(buffer, “area%ld_load_%s”, area1, name);
  _load1.setName(buffer);

  sprintf(buffer, “area%ld_unload_%s”, area2, name);
  _unload2.setName(buffer);

  sprintf(buffer, “machine%ld_%s”, area2, name);
  _process2.setName(buffer);

  sprintf(buffer, “area%ld_load_%s”, area2, name);
  _load2.setName(buffer);

  sprintf(buffer, “stock_load_%s”, name);
  _unloadS.setName(buffer);

  _loadA.setTransitionType(AREAA - 1);
  _unload1.setTransitionType(_area1 - 1);
  _load1.setTransitionType(_area1 - 1);
  _unload2.setTransitionType(_area2 - 1);
  _load2.setTransitionType(_area2 - 1);
  _unloadS.setTransitionType(AREAS - 1);
}

Job::~Job()
{}

void Job::addToModel(IloModel model,
                     IloStateResource trolley,
                     IloNumVar makespan,
                     IloTransitionParam transitionParam)
{
  /* ADD MACHINE REQUIREMENT CONSTRAINTS */
  model.add(_process1.requires(_machine1));
```

```
  model.add(_process2.requires(_machine2));

  /* ADD TEMPORAL CONSTRAINTS BETWEEN ACTIVITIES */
  IloNum delay = transitionParam.getValue((IloInt)_loadA.getTransitionType(),

(IloInt)_unload1.getTransitionType());
  model.add(_unload1.startsAfterEnd(_loadA, delay));
  model.add(_process1.startsAfterEnd(_unload1));
  model.add(_load1.startsAfterEnd(_process1));
  delay = transitionParam.getValue((IloInt)_load1.getTransitionType(),
                                   (IloInt)_unload2.getTransitionType());
  model.add(_unload2.startsAfterEnd(_load1, delay));
  model.add(_process2.startsAfterEnd(_unload2));
  model.add(_load2.startsAfterEnd(_process2));
  delay = transitionParam.getValue((IloInt)_load2.getTransitionType(),
                                   (IloInt)_unloadS.getTransitionType());
  model.add(_unloadS.startsAfterEnd(_load2, delay));

  /* ADD TROLLEY POSITION REQUIREMENTS  */
  model.add(_loadA.requires(trolley, (IloAny)AREAA));
  model.add(_unload1.requires(trolley, (IloAny)_area1));
  model.add(_load1.requires(trolley, (IloAny)_area1));
  model.add(_unload2.requires(trolley, (IloAny)_area2));
  model.add(_load2.requires(trolley, (IloAny)_area2));
  model.add(_unloadS.requires(trolley, (IloAny)AREAS));

  /* ADD MAKESPAN CONSTRAINT */
  model.add(_unloadS.endsBefore(makespan));
}


void Job::printSolution(IlcScheduler scheduler, ILOSTD(ostream)& out)
{
  /* PRINT JOB */
  out
    << "JOB " << _name << endl
    << "\t Load at area A: "
    << scheduler.getActivity( _loadA ) << endl
    << "\t Unload at area " << (long)_area1 << ": "
    << scheduler.getActivity( _unload1 )   << endl
    << "\t Process on machine " << (long)_area1 << ": "
    << scheduler.getActivity( _process1 )  << endl
    << "\t Load at area "   << (long)_area1 << ": "
    << scheduler.getActivity( _load1 )     << endl
    << "\t Unload at area " << (long)_area2 << ": "
    << scheduler.getActivity( _unload2 )   << endl
    << "\t Process on machine " << (long)_area2 << ": "
    << scheduler.getActivity( _process2 )  << endl
    << "\t Load at area "   << (long)_area2 << " : "
    << scheduler.getActivity( _load2 )     << endl
    << "\t Unload at area S: "
    << scheduler.getActivity( _unloadS )   << endl;
}


////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
```

```
//
////////////////////////////////////////////////////////////////////////////

#if defined(ILO_SDXLOUTPUT)
#include “sdxltrolley.h”
#endif

IloModel DefineModel(IloEnv env,
                     const char** jobNames,
                     IloNum*      transitDurations,
                     IloInt*      machines1,
                     IloNum*      durations1,
                     IloInt*      machines2,
                     IloNum*      durations2,
                     IloAnyArray&  jobs,
                     IloStateResource& trolley,
                     IloArray<IloUnaryResource>& machines,
                     IloNumVar&   makespan)
{
  IloInt i, j;
  IloModel model(env);

  /* CREATE THE MAKESPAN VARIABLE. */
  IloNum horizon =
    numberOfJobs * ((6 * loadDuration) +
                    (4 * getMaxTransitDuration(5, transitDurations)));
  for (i = 0; i < numberOfJobs; i++)
    horizon += durations1[i] + durations2[i];
  makespan = IloNumVar(env,0,horizon,ILOINT);
  /* CREATE THE TROLLEY POSITION RESOURCE */
  IloAnyArray arrayOfPossibleAreas(env,
                                   5,
                                   (IloAny)AREAA,
                                   (IloAny)MACH1,
                                   (IloAny)MACH2,
                                   (IloAny)MACH3,
                                   (IloAny)AREAS);
  IloAnySet   setOfPossibleAreas(env,arrayOfPossibleAreas);
  trolley = IloStateResource(env,setOfPossibleAreas, “trolley”);
  /* CREATE THE (SYMMETRIC) TRANSITION TIME TABLE */
  IloTransitionParam transitionParam(env, 5, IloTrue);
  for (i = 0; i < 5; i++)
    for (j = i; j < 5; j++)
      transitionParam.setValue(i,j,transitDurations[j + (5 * i)]);
  /* ASSOCIATE A TRANSITION TIME TO THE RESOURCE */
  IloTransitionTime transitionTime(trolley,transitionParam);

  /* SELECT APPROPRIATE ENFORCEMENT LEVEL */
  trolley.setTransitionTimeEnforcement(IloBasic);
  trolley.setCapacityEnforcement(IloMediumHigh);
  /* CREATE THE MACHINES RESOURCES */
  char buffer[256];
  machines = IloArray<IloUnaryResource>(env, 3);
  for (i = 0; i < 3; i++) {
    sprintf(buffer, “machine%ld”, i + 1);
    machines[i] = IloUnaryResource(env, buffer);
  }
```

```
  /* CREATION OF JOB INSTANCES */
  jobs = IloAnyArray(env, numberOfJobs);
  for (i = 0; i < numberOfJobs; i++) {
    Job* job = new (env) Job(env,
                             jobNames[i],
                             loadDuration,
                             machines[machines1[i] - 1],
                             durations1[i], machines1[i],
                             machines[machines2[i] - 1],
                             durations2[i], machines2[i]);
    jobs[i] = job;
    job->addToModel(model, trolley, makespan, transitionParam);
  }

  /* WE LOOK FOR AN OPTIMAL SOLUTION */
  model.add(IloMinimize(env, makespan));

  /* RETURN THE CREATED MODEL */
  return model;
}


////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////////////////

void PrintSolution(IloSolver solver,
                   IloAnyArray jobs,
                   IloStateResource trolley,
                   IloArray<IloUnaryResource> machines,
                   IloNumVar makespan)
{
  IlcScheduler scheduler(solver);

  solver.out() << "makespan = " << solver.getMin(makespan) << endl;
  for (IloInt i = 0; i < numberOfJobs; i++)
    ((Job*) jobs[i])->printSolution(scheduler, solver.out());

  solver.out() << "TROLLEY POSITIONS: " << endl;

  IlcStateResourceIterator ite(scheduler);
  IlcAnyTimetable tt = scheduler.getStateResource(trolley).getTimetable();
  for (IlcAnyTimetableCursor cr(tt,0); cr.ok(); ++cr)
    if (cr.isBound())
      solver.out() << "\t [" << cr.getTimeMin()
                   << "," << cr.getTimeMax()
                   << "): Position " << (long)cr.getValue()
                   << endl;

  solver.printInformation();

#if defined(ILO_SDXLOUTPUT)
  IloSDXLTrolleyOutput output(solver.getEnv());
  ofstream outFile("trolley2.xml");
  output.writeTrolley(scheduler,
                      outFile,
```

```
                    jobs,
                    trolley,
                    (IloInt) 1000,
                    machines,
                    makespan);
  outFile.close();
#endif

}


////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////

int main(int argc, char** argv) {
  IloInt failLimit        = 1000;
  if (argc > 1)
    failLimit = atol(argv[1]);

  try {
    IloEnv env;
    IloAnyArray jobs;
    IloStateResource trolley;
    IloArray<IloUnaryResource> machines;
    IloNumVar makespan;
    IloModel model = DefineModel(env,
        JobNames,
        TransitDurations,
        Machines1, Durations1,
        Machines2, Durations2,
        jobs,
        trolley,
        machines,
        makespan);

    IloBool solved = IloFalse;
    IloNum  minMakespan = 0.;

    IloSolver solver(model);
    IloGoal unlimitedGoal = IloSetTimesForward(env, makespan);
    IloGoal limitedGoal = IloLimitSearch(env, unlimitedGoal,
                                         IloFailLimit(env,failLimit) );
    solver.startNewSearch(limitedGoal);

    while (solver.next()==IlcTrue) {
             solved = IloTrue;
      minMakespan = solver.getMin(makespan);
      solver.out() << "SOLUTION WITH MAKESPAN "
                        << minMakespan << endl;
    }
    solver.endSearch();

    if (solved){
      makespan.setBounds(minMakespan,minMakespan);
      solver.solve(unlimitedGoal);
```

```
      PrintSolution(solver,jobs,trolley,machines,makespan);
    } else {
      solver.out() << "NO SOLUTION FOUND" << endl;
    }
    env.end();

  } catch (IloException& exc) {
    cout << exc << endl;
  }

  return 0;
}


////////////////////////////////////////////////////////////////////////////
//
// RESULTS
//
////////////////////////////////////////////////////////////////////////////
/*
  SOLUTION WITH MAKESPAN 1220
  SOLUTION WITH MAKESPAN 1130
  SOLUTION WITH MAKESPAN 1110
  SOLUTION WITH MAKESPAN 1070
  SOLUTION WITH MAKESPAN 990
  SOLUTION WITH MAKESPAN 980
  makespan = 980
  JOB J1
         Load at area A: [0 -- 20 --> 20]
         Unload at area 1: [70 -- 20 --> 90]
         Process on machine 1: [250 -- 80 --> 330]
         Load at area 1: [330 -- 20 --> 350]
         Unload at area 2: [400 -- 20 --> 420]
         Process on machine 2: [500 -- 60 --> 560]
         Load at area 2 : [580 -- 20 --> 600]
         Unload at area S: [960 -- 20 --> 980]
  JOB J2
         Load at area A: [0 -- 20 --> 20]
         Unload at area 2: [140 -- 20 --> 160]
         Process on machine 2: [160 -- 120 --> 280]
         Load at area 2: [400 -- 20 --> 420]
         Unload at area 3: [500 -- 20 --> 520]
         Process on machine 3: [680 -- 80 --> 760]
         Load at area 3 : [810 -- 20 --> 830]
         Unload at area S: [960 -- 20 --> 980]
  JOB J3
         Load at area A: [0 -- 20 --> 20]
         Unload at area 2: [140 -- 20 --> 160]
         Process on machine 2: [420 -- 80 --> 500]
         Load at area 2: [580 -- 20 --> 600]
         Unload at area 1: [650 -- 20 --> 670]
         Process on machine 1: [670 -- 60 --> 730]
         Load at area 1 : [730 -- 20 --> 750]
         Unload at area S: [960 -- 20 --> 980]
  JOB J4
         Load at area A: [0 -- 20 --> 20]
         Unload at area 1: [70 -- 20 --> 90]
         Process on machine 1: [90 -- 160 --> 250]
```

```
        Load at area 1: [300 -- 20 --> 320]
        Unload at area 3: [500 -- 20 --> 520]
        Process on machine 3: [580 -- 100 --> 680]
        Load at area 3 : [810 -- 20 --> 830]
        Unload at area S: [960 -- 20 --> 980]
  JOB J5
        Load at area A: [0 -- 20 --> 20]
        Unload at area 3: [220 -- 20 --> 240]
        Process on machine 3: [240 -- 180 --> 420]
        Load at area 3: [500 -- 20 --> 520]
        Unload at area 2: [580 -- 20 --> 600]
        Process on machine 2: [600 -- 80 --> 680]
        Load at area 2 : [890 -- 20 --> 910]
        Unload at area S: [960 -- 20 --> 980]
  JOB J6
        Load at area A: [0 -- 20 --> 20]
        Unload at area 2: [140 -- 20 --> 160]
        Process on machine 2: [280 -- 140 --> 420]
        Load at area 2: [420 -- 20 --> 440]
        Unload at area 3: [500 -- 20 --> 520]
        Process on machine 3: [520 -- 60 --> 580]
        Load at area 3 : [810 -- 20 --> 830]
        Unload at area S: [960 -- 20 --> 980]
  TROLLEY POSITIONS:
        [0,20): Position 4
        [70,90): Position 1
        [140,160): Position 2
        [220,240): Position 3
        [300,320): Position 1
        [330,350): Position 1
        [400,440): Position 2
        [500,520): Position 3
        [580,600): Position 2
        [650,670): Position 1
        [730,750): Position 1
        [810,830): Position 3
        [890,910): Position 2
        [960,980): Position 5
*/
```

This solution corresponds to a maximal number of fails per optimization step of 1000. It is represented in Figure 33.2. (Positions 4 and 5 in the output results correspond to the areas **A** and **S** in the diagram.)

Notice the effect of transition times on the bottom line of the schedule, representing the trolley. In the first version of the example, the trolley moved from one area to another 19 times. When transition times are taken into account, moves become critical with respect to the makespan criterion: which explains why the trolley performs only 11 moves in the solution shown in Figure 33.2.

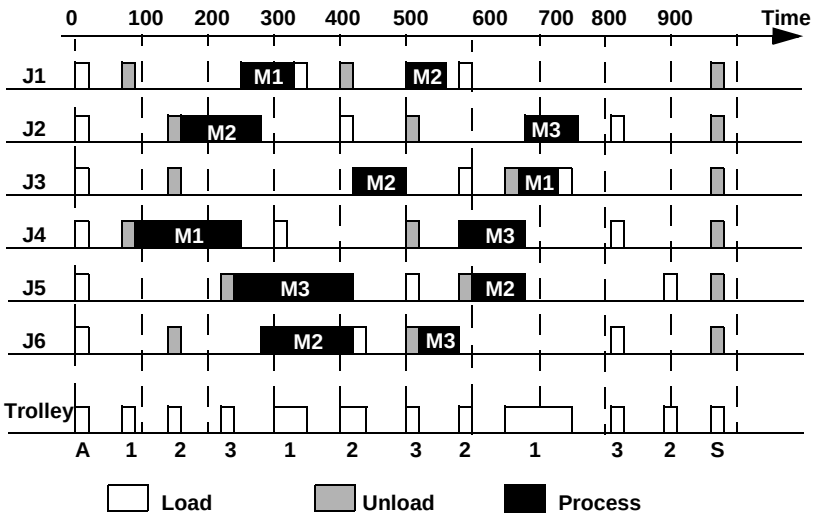


Figure 33.2 *A Solution for the Trolley Problem with Transition Times*

C H A P T E R

34

Scheduling with State Resources: the Trolley Problem, with Transition Times and Limited Capacity

In this chapter, we apply a capacity limitation on the trolley in our state resource example.

Describing the Problem

The problem is similar to the example in Chapter 33, *Scheduling with State Resources: the Trolley Problem, with Transition Times*, but now we want to represent a constraint on the maximal number of items the trolley can carry at the same time.

```
const IloNum trolleyMaxCapacity = 3;
const IloNum loadDuration     = 20;
const IloInt numberOfJobs     = 6;
```

Representing State Resource with Limited Capacity

The capacity constraint makes a change in how we model the trolley resource, and effects a change in the jobs and activities.

Resource Modeling

The maximal capacity constraint is represented by defining a *discrete resource* associated with the trolley. This resource is defined in the function `DefineModel` as follows.

```
/* CREATE THE TROLLEY CAPACITY RESOURCE */
IloDiscreteResource trolleyCapacity(env,trolleyMaxCapacity);
```

Let's see now how this discrete resource is used by the activities.

Jobs and Activities

An item occupies the trolley from the moment it is loaded on the trolley until the moment it is unloaded. As a consequence, three activities `_onTrolleyA1`, `_onTrolley12`, and `_onTrolley2S`, corresponding to the occupation of the trolley by the item, are added to the `Job` class.

```
class Job {
private:
  const char* _name;
  IloActivity _loadA;
  IloActivity _unload1;
  IloActivity _process1;
  IloActivity _load1;
  IloActivity _unload2;
  IloActivity _process2;
  IloActivity _load2;
  IloActivity _unloadS;
  IloInt      _area1;
  IloInt      _area2;
  IloUnaryResource _machine1;
  IloUnaryResource _machine2;
public:
  Job(IloEnv env,
      const char*,
      IloNum loadDuration,
      IloUnaryResource machine1, IloNum duration1, IloInt area1,
      IloUnaryResource machine2, IloNum duration2, IloInt area2);
  void addToModel(IloModel model,
                  IloStateResource trolley,
                  IloNumVar makespan,
                  IloDiscreteResource trolleyCapacity,
                  IloTransitionParam transitionParam);
  ~Job();
  void printSolution(IlcScheduler scheduler, ILOSTD(ostream)& out);
  const char* getName() const { return  _name;}

  IloActivity getLoadA() const { return _loadA;}
  IloActivity getUnload1() const { return _unload1;}
  IloActivity getProcess1() const { return _process1;}
  IloActivity getLoad1() const { return _load1;}
  IloActivity getUnload2() const { return _unload2;}
  IloActivity getProcess2() const { return _process2;}
  IloActivity getLoad2() const { return _load2;}
```

```
  IloActivity getUnloadS() const { return _unloadS;}

};
```

In the constructor of a `Job`, we specify that the trolley is occupied by the item for the interval between the start of the load activity and the end of the corresponding unload activity. The activities `_onTrolley` require one unit of trolley capacity each.

```
Job::Job(IloEnv env,
   const char* name,
   IloNum loadDuration,
   IloUnaryResource machine1, IloNum duration1, IloInt area1,
   IloUnaryResource machine2, IloNum duration2, IloInt area2)
  : _name(name),
    _loadA(      env,loadDuration),
    _unload1(    env,loadDuration),
    _process1(   env,duration1),
    _load1(      env,loadDuration),
    _unload2(    env,loadDuration),
    _process2(   env,duration2),
    _load2(      env,loadDuration),
    _unloadS(    env,loadDuration),
    _area1(area1),
    _area2(area2),
    _machine1(machine1),
    _machine2(machine2)
{
 char buffer[256];

  sprintf(buffer, “arrival_load_%s”, name);
  _loadA.setName(buffer);

  sprintf(buffer, “area%ld_unload_%s”, area1, name);
  _unload1.setName(buffer);

  sprintf(buffer, “machine%ld_%s”, area1, name);
  _process1.setName(buffer);

  sprintf(buffer, “area%ld_load_%s”, area1, name);
  _load1.setName(buffer);

  sprintf(buffer, “area%ld_unload_%s”, area2, name);
  _unload2.setName(buffer);

  sprintf(buffer, “machine%ld_%s”, area2, name);
  _process2.setName(buffer);

  sprintf(buffer, “area%ld_load_%s”, area2, name);
  _load2.setName(buffer);

  sprintf(buffer, “stock_load_%s”, name);
  _unloadS.setName(buffer);

  /* ADD TRANSITION TYPE  */
  _loadA.setTransitionType(AREAA - 1);
  _unload1.setTransitionType(area1 - 1);
  _load1.setTransitionType(area1 - 1);
  _unload2.setTransitionType(area2 - 1);
```

```
  _load2.setTransitionType(area2 - 1);
  _unloadS.setTransitionType(AREAS - 1);

}

void Job::addToModel(IloModel model,
                     IloStateResource trolley,
                     IloNumVar makespan,
                     IloDiscreteResource trolleyCapacity,
                     IloTransitionParam transitionParam)
{

  /* ADD MACHINE REQUIREMENT CONSTRAINTS */
  model.add(_process1.requires(_machine1));
  model.add(_process2.requires(_machine2));

  /* ADD TEMPORAL CONSTRAINTS BETWEEN ACTIVITIES */

  model.add(_unload1.startsAfterEnd(_loadA));
  model.add(_process1.startsAfterEnd(_unload1));
  model.add(_load1.startsAfterEnd(_process1));
  model.add(_unload2.startsAfterEnd(_load1));
  model.add(_process2.startsAfterEnd(_unload2));
  model.add(_load2.startsAfterEnd(_process2));
  model.add(_unloadS.startsAfterEnd(_load2));


  /* ADD TROLLEY POSITION REQUIREMENTS  */
  model.add(_loadA.requires(trolley,(IloAny)AREAA));
  model.add(_unload1.requires(trolley,(IloAny)_area1));
  model.add(_load1.requires(trolley,(IloAny)_area1));
  model.add(_unload2.requires(trolley,(IloAny)_area2));
  model.add(_load2.requires(trolley,(IloAny)_area2));
  model.add(_unloadS.requires(trolley,(IloAny)AREAS));

  /* ADD TROLLEY CAPACITY REQUIREMENTS  */
  IloNum delay = transitionParam.getValue((IloInt)_loadA.getTransitionType(),

(IloInt)_unload1.getTransitionType());
  IloIntVar durationA1(model.getEnv(),
                       2*(IloInt)loadDuration + (IloInt)delay,
                       (IloInt)makespan.getUB());
  IloActivity onTrolleyA1(model.getEnv(), durationA1);
  onTrolleyA1.shareStartWithStart(_loadA);
  onTrolleyA1.shareEndWithEnd(_unload1);
  model.add(onTrolleyA1.requires(trolleyCapacity));
  delay = transitionParam.getValue((IloInt)_load1.getTransitionType(),
                                   (IloInt)_unload2.getTransitionType());
  IloIntVar duration12(model.getEnv(),
                       2*(IloInt)loadDuration + (IloInt)delay,
                       (IloInt)makespan.getUB());
  IloActivity onTrolley12(model.getEnv(), duration12);
  onTrolley12.shareStartWithStart(_load1);
  onTrolley12.shareEndWithEnd(_unload2);
  model.add(onTrolley12.requires(trolleyCapacity));
  delay = transitionParam.getValue((IloInt)_load2.getTransitionType(),
                                   (IloInt)_unloadS.getTransitionType());
  IloIntVar duration2S(model.getEnv(),
```

```
                      2*(IloInt)loadDuration + (IloInt)delay,
                      (IloInt)makespan.getUB());
  IloActivity onTrolley2S(model.getEnv(), duration2S);
  onTrolley2S.shareStartWithStart(_load2);
  onTrolley2S.shareEndWithEnd(_unloadS);
  model.add(onTrolley2S.requires(trolleyCapacity));

  /* ADD MAKESPAN CONSTRAINT */
  model.add(_unloadS.endsBefore(makespan));
}
```

The same trade-off between efficiency of computation time and quality of the solution is made as for the previous version of the trolley example.

Complete Program and Output

You can see the entire program `trolley3.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

const IloInt MACH1 = 1; // Machine 1 area
const IloInt MACH2 = 2; // Machine 2 area
const IloInt MACH3 = 3; // Machine 3 area
const IloInt AREAA = 4; // Arrival area
const IloInt AREAS = 5; // Stock area

const IloNum trolleyMaxCapacity = 3;
const IloNum loadDuration     = 20;
const IloInt numberOfJobs     = 6;
const char* JobNames  [] = { "J1",   "J2",   "J3",   "J4",   "J5",   "J6"};
IloInt Machines1      [] = { 1,      2,      2,      1,      3,      2   };
IloNum Durations1     [] = { 80,     120,    80,     160,    180,    140 };
IloInt Machines2      [] = { 2,      3,      1,      3,      2,      3   };
IloNum Durations2     [] = { 60,     80,     60,     100,    80,     60  };


                              // M1   M2   M3   A     S
IloNum TransitDurations[] = {    0,  50,  60,  50,   90,    // M1
                                50,   0,  60,  90,   50,    // M2
                                60,  60,  0,   80,   80,    // M3
                                50,  90,  80,   0,  120,    // A
                                90,  50,  80, 120,    0   };// S

IloNum getMaxTransitDuration(IloInt nbArea, IloNum* transitDurations){
  IloNum max = 0;
  IloInt size = nbArea*nbArea;
  for (IloInt i = 0; i < size; i++)
    if (transitDurations[i] > max) max = transitDurations[i];
  return max;
}
```

```
////////////////////////////////////////////////////////////////////////////
//
// DEFINITION OF THE JOB CLASS
//
////////////////////////////////////////////////////////////////////////////
class Job {
private:
  const char* _name;
  IloActivity _loadA;
  IloActivity _unload1;
  IloActivity _process1;
  IloActivity _load1;
  IloActivity _unload2;
  IloActivity _process2;
  IloActivity _load2;
  IloActivity _unloadS;
  IloInt      _area1;
  IloInt      _area2;
  IloUnaryResource _machine1;
  IloUnaryResource _machine2;
public:
  Job(IloEnv env,
      const char*,
      IloNum loadDuration,
      IloUnaryResource machine1, IloNum duration1, IloInt area1,
      IloUnaryResource machine2, IloNum duration2, IloInt area2);
  void addToModel(IloModel model,
                  IloStateResource trolley,
                  IloNumVar makespan,
                  IloDiscreteResource trolleyCapacity,
                  IloTransitionParam transitionParam);
  ~Job();
  void printSolution(IlcScheduler scheduler, ILOSTD(ostream)& out);
  const char* getName() const { return  _name;}

  IloActivity getLoadA() const { return _loadA;}
  IloActivity getUnload1() const { return _unload1;}
  IloActivity getProcess1() const { return _process1;}
  IloActivity getLoad1() const { return _load1;}
  IloActivity getUnload2() const { return _unload2;}
  IloActivity getProcess2() const { return _process2;}
  IloActivity getLoad2() const { return _load2;}
  IloActivity getUnloadS() const { return _unloadS;}

};
Job::~Job()
{}
Job::Job(IloEnv env,
   const char* name,
   IloNum loadDuration,
   IloUnaryResource machine1, IloNum duration1, IloInt area1,
   IloUnaryResource machine2, IloNum duration2, IloInt area2)
  : _name(name),
    _loadA(      env,loadDuration),
    _unload1(    env,loadDuration),
    _process1(   env,duration1),
    _load1(      env,loadDuration),
```

```
    _unload2(    env,loadDuration),
    _process2(   env,duration2),
    _load2(      env,loadDuration),
    _unloadS(    env,loadDuration),
    _area1(area1),
    _area2(area2),
    _machine1(machine1),
    _machine2(machine2)
{
 char buffer[256];

  sprintf(buffer, “arrival_load_%s”, name);
  _loadA.setName(buffer);

  sprintf(buffer, “area%ld_unload_%s”, area1, name);
  _unload1.setName(buffer);

  sprintf(buffer, “machine%ld_%s”, area1, name);
  _process1.setName(buffer);

  sprintf(buffer, “area%ld_load_%s”, area1, name);
  _load1.setName(buffer);

  sprintf(buffer, “area%ld_unload_%s”, area2, name);
  _unload2.setName(buffer);

  sprintf(buffer, “machine%ld_%s”, area2, name);
  _process2.setName(buffer);

  sprintf(buffer, “area%ld_load_%s”, area2, name);
  _load2.setName(buffer);

  sprintf(buffer, “stock_load_%s”, name);
  _unloadS.setName(buffer);

  /* ADD TRANSITION TYPE  */
  _loadA.setTransitionType(AREAA - 1);
  _unload1.setTransitionType(area1 - 1);
  _load1.setTransitionType(area1 - 1);
  _unload2.setTransitionType(area2 - 1);
  _load2.setTransitionType(area2 - 1);
  _unloadS.setTransitionType(AREAS - 1);

}

void Job::addToModel(IloModel model,
                     IloStateResource trolley,
                     IloNumVar makespan,
                     IloDiscreteResource trolleyCapacity,
                     IloTransitionParam transitionParam)
{

  /* ADD MACHINE REQUIREMENT CONSTRAINTS */
  model.add(_process1.requires(_machine1));
  model.add(_process2.requires(_machine2));

  /* ADD TEMPORAL CONSTRAINTS BETWEEN ACTIVITIES */
```

```
  model.add(_unload1.startsAfterEnd(_loadA));
  model.add(_process1.startsAfterEnd(_unload1));
  model.add(_load1.startsAfterEnd(_process1));
  model.add(_unload2.startsAfterEnd(_load1));
  model.add(_process2.startsAfterEnd(_unload2));
  model.add(_load2.startsAfterEnd(_process2));
  model.add(_unloadS.startsAfterEnd(_load2));


  /* ADD TROLLEY POSITION REQUIREMENTS  */
  model.add(_loadA.requires(trolley,(IloAny)AREAA));
  model.add(_unload1.requires(trolley,(IloAny)_area1));
  model.add(_load1.requires(trolley,(IloAny)_area1));
  model.add(_unload2.requires(trolley,(IloAny)_area2));
  model.add(_load2.requires(trolley,(IloAny)_area2));
  model.add(_unloadS.requires(trolley,(IloAny)AREAS));

  /* ADD TROLLEY CAPACITY REQUIREMENTS  */
  IloNum delay = transitionParam.getValue((IloInt)_loadA.getTransitionType(),

(IloInt)_unload1.getTransitionType());
  IloIntVar durationA1(model.getEnv(),
                       2*(IloInt)loadDuration + (IloInt)delay,
                       (IloInt)makespan.getUB());
  IloActivity onTrolleyA1(model.getEnv(), durationA1);
  onTrolleyA1.shareStartWithStart(_loadA);
  onTrolleyA1.shareEndWithEnd(_unload1);
  model.add(onTrolleyA1.requires(trolleyCapacity));
  delay = transitionParam.getValue((IloInt)_load1.getTransitionType(),
                                   (IloInt)_unload2.getTransitionType());
  IloIntVar duration12(model.getEnv(),
                       2*(IloInt)loadDuration + (IloInt)delay,
                       (IloInt)makespan.getUB());
  IloActivity onTrolley12(model.getEnv(), duration12);
  onTrolley12.shareStartWithStart(_load1);
  onTrolley12.shareEndWithEnd(_unload2);
  model.add(onTrolley12.requires(trolleyCapacity));
  delay = transitionParam.getValue((IloInt)_load2.getTransitionType(),
                                   (IloInt)_unloadS.getTransitionType());
  IloIntVar duration2S(model.getEnv(),
                       2*(IloInt)loadDuration + (IloInt)delay,
                       (IloInt)makespan.getUB());
  IloActivity onTrolley2S(model.getEnv(), duration2S);
  onTrolley2S.shareStartWithStart(_load2);
  onTrolley2S.shareEndWithEnd(_unloadS);
  model.add(onTrolley2S.requires(trolleyCapacity));

  /* ADD MAKESPAN CONSTRAINT */
  model.add(_unloadS.endsBefore(makespan));
}


void Job::printSolution(IlcScheduler scheduler, ILOSTD(ostream)&)
{
  /* PRINT JOB */
  scheduler.getSolver().out()
  << “JOB “ << _name << endl
  << “\t Load at area A: “
```

```
  << scheduler.getActivity( _loadA ) << endl
  << "\t Unload at area " << (long)_area1 << ": "
  << scheduler.getActivity( _unload1 )   << endl
  << "\t Process on machine " << (long)_area1 << ": "
  << scheduler.getActivity( _process1 )  << endl
  << "\t Load at area "   << (long)_area1 << ": "
  << scheduler.getActivity( _load1 )     << endl
  << "\t Unload at area " << (long)_area2 << ": "
  << scheduler.getActivity( _unload2 )   << endl
  << "\t Process on machine " << (long)_area2 << ": "
  << scheduler.getActivity( _process2 )  << endl
  << "\t Load at area "   << (long)_area2 << " : "
  << scheduler.getActivity( _load2 )     << endl
  << "\t Unload at area S: "
  << scheduler.getActivity( _unloadS )   << endl;
}


////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////

#if defined(ILO_SDXLOUTPUT)
#include "sdxltrolley.h"
#endif

IloModel DefineModel(IloEnv& env,
                     const char** jobNames,
                     IloNum*      transitDurations,
                     IloInt*      machines1,
                     IloNum*      durations1,
                     IloInt*      machines2,
                     IloNum*      durations2,
                     IloAnyArray&  jobs,
                     IloStateResource& trolley,
                     IloArray<IloUnaryResource>& machines,
                     IloNumVar&   makespan)
{
  IloInt i, j;
  IloModel model(env);

  /* CREATE THE MAKESPAN VARIABLE. */
  IloNum horizon =
    numberOfJobs * ((6 * loadDuration) +
                    (4 * getMaxTransitDuration(5, transitDurations)));
  for (i = 0; i < numberOfJobs; i++)
    horizon += durations1[i] + durations2[i];
  makespan = IloNumVar(env,0,horizon,ILOINT);

  /* CREATE THE TROLLEY POSITION RESOURCE */
  IloAnyArray arrayOfPossibleAreas(env,
                                   5,
                                   (IloAny)AREAA,
                                   (IloAny)MACH1,
                                   (IloAny)MACH2,
                                   (IloAny)MACH3,
```

```
                                    (IloAny)AREAS);
  IloAnySet   setOfPossibleAreas(env,arrayOfPossibleAreas);
  trolley = IloStateResource(env,setOfPossibleAreas, "trolley");

  /* CREATE THE (SYMMETRIC) TRANSITION TIME TABLE */
  IloTransitionParam transitionParam(env, 5, IloTrue);
  for (i = 0; i < 5; i++)
    for (j = i; j < 5; j++)
      transitionParam.setValue(i,j,transitDurations[j + (5 * i)]);

  /* ASSOCIATE A TRANSITION TIME TO THE RESOURCE */
  IloTransitionTime transitionTime(trolley,transitionParam);

  /* SELECT APPROPRIATE ENFORCEMENT LEVEL */
  trolley.setTransitionTimeEnforcement(IloBasic);
  trolley.setCapacityEnforcement(IloMediumHigh);

  /* CREATE THE TROLLEY CAPACITY RESOURCE */
  IloDiscreteResource trolleyCapacity(env,trolleyMaxCapacity);

  /* SELECT APPROPRIATE ENFORCEMENT LEVEL */
  trolleyCapacity.setCapacityEnforcement(IloMediumHigh);

  /* CREATE THE MACHINES RESOURCES */
  char buffer[256];
  machines = IloArray<IloUnaryResource>(env, 3);
  for (i = 0; i < 3; i++) {
    sprintf(buffer, "machine%ld", i + 1);
    machines[i] = IloUnaryResource(env, buffer);
  }


  /* CREATION OF JOB INSTANCES */
  jobs = IloAnyArray(env, numberOfJobs);
  for (i = 0; i < numberOfJobs; i++) {
    Job* job = new (env) Job(env,
                             jobNames[i],
                             loadDuration,
                             machines[machines1[i] - 1],
                             durations1[i], machines1[i],
                             machines[machines2[i] - 1],
                             durations2[i], machines2[i]);
    jobs[i] = job;
    job->addToModel(model,
                    trolley, makespan,trolleyCapacity, transitionParam);
  }
 /* WE LOOK FOR AN OPTIMAL SOLUTION */
  model.add(IloMinimize(env, makespan));

  /* RETURN THE CREATED MODEL */
  return model;
}

////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////////////////
```

```
void PrintSolution(IloSolver solver,
                   IloAnyArray jobs,
                   IloStateResource trolley,
                   IloArray<IloUnaryResource> machines,
                   IloNumVar makespan)
{
  IlcScheduler scheduler(solver);

  solver.out() << "makespan = " << solver.getMin(makespan) << endl;
  for (IloInt i = 0; i < numberOfJobs; i++)
    ((Job*) jobs[i])->printSolution(scheduler, solver.out());

  solver.out() << "TROLLEY POSITIONS: " << endl;

  IlcStateResourceIterator ite(scheduler);
  IlcAnyTimetable tt = scheduler.getStateResource(trolley).getTimetable();
  for (IlcAnyTimetableCursor cr(tt,0); cr.ok(); ++cr)
    if (cr.isBound())
      solver.out() << "\t [" << cr.getTimeMin()
                   << "," << cr.getTimeMax()
                   << "): Position " << (long)cr.getValue()
                   << endl;

  solver.printInformation();

#if defined(ILO_SDXLOUTPUT)
  IloSDXLTrolleyOutput output(solver.getEnv());
  ofstream outFile("trolley3.xml");
  output.writeTrolley(scheduler,
                      outFile,
                      jobs,
                      trolley,
                      trolleyMaxCapacity,
                      machines,
                      makespan);
  outFile.close();
#endif

}

////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////

int main(int argc, char** argv) {

  IloInt failLimit          = 1000;
  if (argc > 1)
    failLimit = atol(argv[1]);

  try {
    IloEnv env;
    IloAnyArray jobs;
    IloStateResource trolley;
    IloArray<IloUnaryResource> machines;
```

```
    IloNumVar makespan;
    IloModel model = DefineModel(env,
        JobNames,
        TransitDurations,
        Machines1, Durations1,
        Machines2, Durations2,
        jobs,
        trolley,
        machines,
        makespan);

    IloBool solved = IloFalse;
    IloNum  minMakespan = 0.;
    IloSolver solver(model);

    // create limited search goal
    IloGoal unlimitedGoal = IloSetTimesForward(env, makespan);
    IloGoal limitedGoal = IloLimitSearch(env, unlimitedGoal,
IloFailLimit(env,failLimit) );
    solver.startNewSearch(limitedGoal);

    while (solver.next()==IlcTrue) {
             solved = IloTrue;
      minMakespan = solver.getMin(makespan);
      solver.out() << "SOLUTION WITH MAKESPAN "
                          << minMakespan << endl;
    }
    solver.endSearch();

    if (solved){
      makespan.setBounds(minMakespan,minMakespan);
      solver.solve(unlimitedGoal);
      PrintSolution(solver,jobs,trolley,machines,makespan);
    } else {
      solver.out() << "NO SOLUTION FOUND" << endl;
    }

    env.end();

  } catch (IloException& exc) {
    cout << exc << endl;
  }

  return 0;
}

////////////////////////////////////////////////////////////////////////////
//
// RESULTS
//
////////////////////////////////////////////////////////////////////////////

/*
  SOLUTION WITH MAKESPAN 1540
  SOLUTION WITH MAKESPAN 1450
  SOLUTION WITH MAKESPAN 1440
  SOLUTION WITH MAKESPAN 1370
  makespan = 1370
```

```
JOB J1
       Load at area A: [460 -- 20 --> 480]
       Unload at area 1: [530 -- 20 --> 550]
       Process on machine 1: [550 -- 80 --> 630]
       Load at area 1: [870 -- 20 --> 890]
       Unload at area 2: [940 -- 20 --> 960]
       Process on machine 2: [960 -- 60 --> 1020]
       Load at area 2 : [1080 -- 20 --> 1100]
       Unload at area S: [1150 -- 20 --> 1170]
JOB J2
       Load at area A: [140 -- 20 --> 160]
       Unload at area 2: [320 -- 20 --> 340]
       Process on machine 2: [340 -- 120 --> 460]
       Load at area 2: [600 -- 20 --> 620]
       Unload at area 3: [700 -- 20 --> 720]
       Process on machine 3: [880 -- 80 --> 960]
       Load at area 3 : [1250 -- 20 --> 1270]
       Unload at area S: [1350 -- 20 --> 1370]
JOB J3
       Load at area A: [460 -- 20 --> 480]
       Unload at area 2: [600 -- 20 --> 620]
       Process on machine 2: [620 -- 80 --> 700]
       Load at area 2: [800 -- 20 --> 820]
       Unload at area 1: [870 -- 20 --> 890]
       Process on machine 1: [890 -- 60 --> 950]
       Load at area 1 : [1010 -- 20 --> 1030]
       Unload at area S: [1150 -- 20 --> 1170]
JOB J4
       Load at area A: [0 -- 20 --> 20]
       Unload at area 1: [70 -- 20 --> 90]
       Process on machine 1: [90 -- 160 --> 250]
       Load at area 1: [390 -- 20 --> 410]
       Unload at area 3: [700 -- 20 --> 720]
       Process on machine 3: [720 -- 100 --> 820]
       Load at area 3 : [1250 -- 20 --> 1270]
       Unload at area S: [1350 -- 20 --> 1370]
JOB J5
       Load at area A: [0 -- 20 --> 20]
       Unload at area 3: [240 -- 20 --> 260]
       Process on machine 3: [260 -- 180 --> 440]
       Load at area 3: [720 -- 20 --> 740]
       Unload at area 2: [800 -- 20 --> 820]
       Process on machine 2: [820 -- 80 --> 900]
       Load at area 2 : [940 -- 20 --> 960]
       Unload at area S: [1150 -- 20 --> 1170]
JOB J6
       Load at area A: [0 -- 20 --> 20]
       Unload at area 2: [320 -- 20 --> 340]
       Process on machine 2: [460 -- 140 --> 600]
       Load at area 2: [620 -- 20 --> 640]
       Unload at area 3: [700 -- 20 --> 720]
       Process on machine 3: [820 -- 60 --> 880]
       Load at area 3 : [1250 -- 20 --> 1270]
       Unload at area S: [1350 -- 20 --> 1370]
TROLLEY POSITIONS:
       [0,20): Position 4
       [70,90): Position 1
       [140,160): Position 4
```

```
        [240,260): Position 3
        [320,340): Position 2
        [390,410): Position 1
        [460,480): Position 4
        [530,550): Position 1
        [600,640): Position 2
        [700,740): Position 3
        [800,820): Position 2
        [870,890): Position 1
        [940,960): Position 2
        [1010,1030): Position 1
        [1080,1100): Position 2
        [1150,1170): Position 5
        [1250,1270): Position 3
        [1350,1370): Position 5
*/
```

This solution corresponds to a maximal number of fails per optimization step of 1000. It is represented in Figure 34.1. Because the trolley capacity is now limited, 17 moves of the trolley are necessary to achieve the schedule (compared to 11 moves for a trolley without capacity limits).

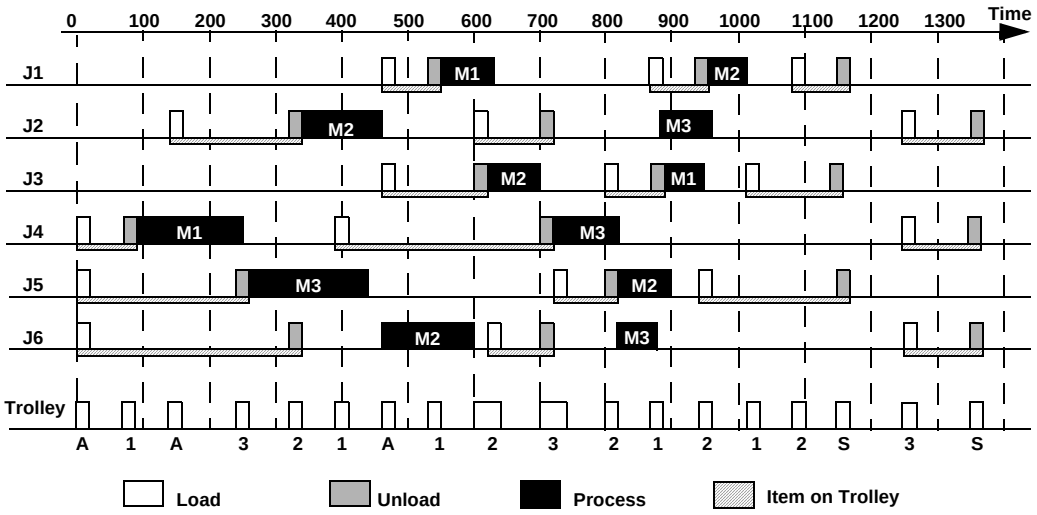


Figure 34.1 *A Solution for the Trolley Problem with Transition Times and Limited Capacity*

C H A P T E R

35

Representing Time-Varying Resource Capacity

The preceding examples in this manual assume that resource *capacity* does not change over time (even though the demand for a resource or the duration of an activity may vary). This chapter presents a simple example in which the capacity of a resource varies over time. In this example, the resource to consider consists of a team of interchangeable people, where the number of available people changes each day. We'll use the member function `setCapacityMax` of the class `IloDiscreteResource` to represent the variation of resource capacity over time.

Describing the Problem

The problem consists of 10 activities to perform over a period of 20 days. For each activity, the following table specifies the duration of the activity and the number of people it requires.

Table 35.1 *Activity, Duration, and Required Capacity*

Activity	Duration	Required Capacity
A0	2	1
A1	4	1

Table 35.1 *Activity, Duration, and Required Capacity (Continued)*

Activity	Duration	Required Capacity
A2	5	1
A3	6	1
A4	8	1
A5	2	2
A6	1	2
A7	3	1
A8	4	2
A9	6	1

The corresponding data are represented in the arrays `Durations` and `RequiredCapacities`. In addition, the array `AvailableCapacities` defines the number of people available each day, that is, from day 0 to day 19.

```
IloInt NumberOfActivities = 10;
IloInt NumberOfDays = 20;
IloNum AvailableCapacities[] = {2, 5, 5, 5, 3, 3, 2, 1, 0, 5,
                                2, 4, 5, 3, 2, 1, 4, 2, 1, 3};
IloNum Durations[] = {2, 4, 5, 6, 8, 2, 1, 3, 4, 6};
IloNum RequiredCapacities[] = {1, 1, 1, 1, 1, 2, 2, 1, 2, 1};
```

Figure 35.1 displays a graph of the evolution of the resource capacity over time.

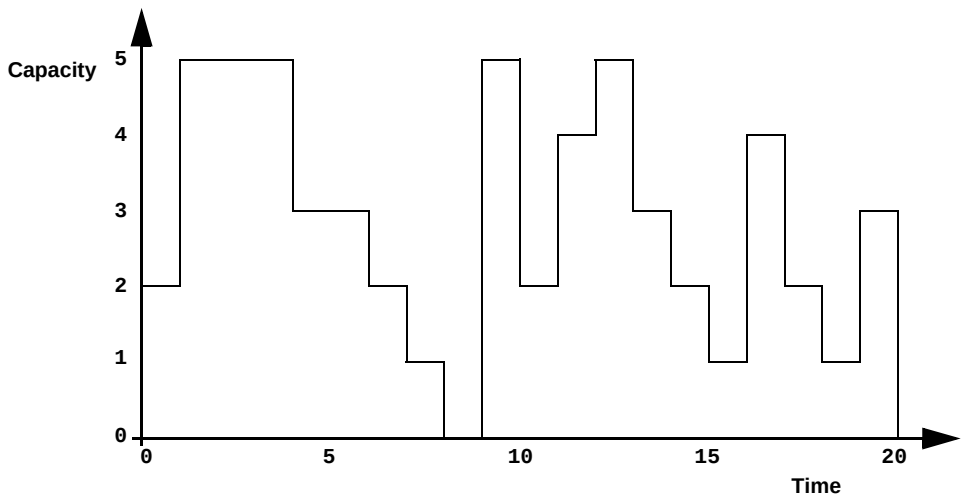


Figure 35.1 *Resource Capacity Varies Over Time.*

Only the `main` function refers to the global variables just defined. When we pass the value of a global variable as an argument to another function, we derive the name of the argument from the global variable, but do not capitalize the first character of the argument name.

Defining the Problem, Designing a Model

To create the corresponding resource and, especially, the resource *timetable*, we have to determine the *maximal theoretic capacity.* The following code performs this computation and keeps the result in a C++ variable, `capacity`.

With the member function `setCapacityMax`, we specify the number of people actually available each day. As its arguments, this function receives two points in time, `timeMin` and `timeMax`, (defining an interval `[timeMin timeMax)`) and an integer specifying the maximal capacity available over this interval. Here, for each `day`, `timeMin` is equal to `day`, `timeMax` is equal to `day+1`, and the available capacity is equal to `availableCapacities[day]`.

```
/* CREATE THE RESOURCE. */
IloNum capacity = 0;
IloInt day;
for (day = 0; day < numberOfDays; day++)
  if (capacity < availableCapacities[day])
    capacity = availableCapacities[day];
resource = IloDiscreteResource(env, capacity);
/* POST THE MAXIMAL CAPACITY CONSTRAINTS. */
for (day = 0; day < NumberOfDays; day++)
  resource.setCapacityMax(day, day + 1, availableCapacities[day]);
```

The following code creates the activities and specifies the number of people required for each activity.

```
/* CREATE THE ACTIVITIES AND POST THE RESOURCE REQUIREMENT
   CONSTRAINTS. */
char buffer[128];
IloInt i;
for (i = 0; i < numberOfActivities; i++) {
  IloActivity activity(env, durations[i]);
  sprintf(buffer, “A%ld”, i);
  activity.setName(buffer);
  model.add(activity.requires(resource, requiredCapacities[i]));
}
```

Solving the Problem

We use the predefined Solver function, `IloGenerate`, to generate a solution to the problem.

```
/* FILL AN ARRAY WITH THE VARIABLES REPRESENTING THE START TIMES
   OF THE ACTIVITIES. */
IloNumVarArray startTimes(env, NumberOfActivities, 0, IloInfinity, ILOINT);
IloInt i = 0;
for (IloIterator<IloActivity> iterator(env);
     iterator.ok();
     ++iterator) {
  IloActivity activity = *iterator;
  model.add(startTimes[i] == activity.getStartExpr());
  i++;
}
/* DEFINE A GOAL THAT GENERATE A SOLUTION TO THE PROBLEM. */
IloGoal goal = IloGenerate(env, startTimes, IlcChooseMinMinInt);
```

Complete Program and Output

You can see the entire program `capvar.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

#if defined(ILO_SDXLOUTPUT)
#include “sdxloutput.h”
#endif


IloInt NumberOfActivities = 10;
IloInt NumberOfDays = 20;
IloNum AvailableCapacities[] = {2, 5, 5, 5, 3, 3, 2, 1, 0, 5,
                                2, 4, 5, 3, 2, 1, 4, 2, 1, 3};
IloNum Durations[] = {2, 4, 5, 6, 8, 2, 1, 3, 4, 6};
IloNum RequiredCapacities[] = {1, 1, 1, 1, 1, 2, 2, 1, 2, 1};
////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////

IloModel
DefineProblem(IloEnv env,
              IloInt numberOfDays,
              IloNum* availableCapacities,
              IloInt numberOfActivities,
              IloNum* durations,
              IloNum* requiredCapacities,
              IloDiscreteResource& resource)
```

```
{
  /* CREATE THE MODEL. */
  IloModel model(env);
  IloSchedulerEnv schedEnv(env);
  schedEnv.setOrigin(0);
  schedEnv.setHorizon(numberOfDays);
  /* CREATE THE RESOURCE. */
  IloNum capacity = 0;
  IloInt day;
  for (day = 0; day < numberOfDays; day++)
    if (capacity < availableCapacities[day])
      capacity = availableCapacities[day];
  resource = IloDiscreteResource(env, capacity);
  /* POST THE MAXIMAL CAPACITY CONSTRAINTS. */
  for (day = 0; day < NumberOfDays; day++)
    resource.setCapacityMax(day, day + 1, availableCapacities[day]);
  /* CREATE THE ACTIVITIES AND POST THE RESOURCE REQUIREMENT
     CONSTRAINTS. */
  char buffer[128];
  IloInt i;
  for (i = 0; i < numberOfActivities; i++) {
    IloActivity activity(env, durations[i]);
    sprintf(buffer, "A%ld", i);
    activity.setName(buffer);
    model.add(activity.requires(resource, requiredCapacities[i]));
  }
  /* RETURN THE MODEL. */
  return model;
}

////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////////////////

void
PrintSolution(IloSolver solver, IloDiscreteResource resource)
{
  IlcScheduler scheduler(solver);
  IlcDiscreteResource schedResource =
    scheduler.getDiscreteResource(resource);
  for(IlcResourceConstraintIterator iterator(schedResource);
      iterator.ok();
      ++iterator) {
    IlcActivity activity = iterator.getActivity();
    IlcCapRct capRct(*iterator);
    solver.out() << " Activity " << activity.getName() << " requires "
           << capRct.getCapacity() << " person(s) from day "
           << activity.getStartMin() << " to day "
           << activity.getEndMin() << endl;
  }
  IlcInt timeMin = scheduler.getTimeMin();
  IlcInt timeMax = scheduler.getTimeMax();
  for (IlcInt day = timeMin; day < timeMax; day++)
    solver.out() << " Work on day " << day << " requires "
           << schedResource.getCapacityMin(day) << " out of "
           << AvailableCapacities[day] << " person(s)" << endl;
```

```
}


////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////

int main()
{
  try {
    IloEnv env;
    IloDiscreteResource resource;
    IloModel model = DefineProblem(env,
                                   NumberOfDays,
                                   AvailableCapacities,
                                   NumberOfActivities,
                                   Durations,
                                   RequiredCapacities,
                                   resource);

    IloSolver solver(model);
    /* FILL AN ARRAY WITH THE VARIABLES REPRESENTING THE START TIMES
       OF THE ACTIVITIES. */
    IloNumVarArray startTimes(env, NumberOfActivities, 0, IloInfinity, ILOINT);
    IloInt i = 0;
    for (IloIterator<IloActivity> iterator(env);
         iterator.ok();
         ++iterator) {
      IloActivity activity = *iterator;
      model.add(startTimes[i] == activity.getStartExpr());
      i++;
    }
    /* DEFINE A GOAL THAT GENERATE A SOLUTION TO THE PROBLEM. */
    IloGoal goal = IloGenerate(env, startTimes, IlcChooseMinMinInt);
    if (solver.solve(goal)) {
      PrintSolution(solver, resource);
      solver.printInformation();
#if defined(ILO_SDXLOUTPUT)
      IloSDXLOutput output(env);
      ofstream outFile("capvar.xml");
      output.write(IlcScheduler(solver), outFile);
      outFile.close();
#endif
    } else
      solver.out() << "No solution!" << endl;
    env.end();
  } catch (IloException& exc) {
    cout << exc << endl;
  }

  return 0;
}

////////////////////////////////////////////////////////////////////////////
//
// RESULTS
```

```
//
////////////////////////////////////////////////////////////////////////////

/*
 Activity A9 requires 1 person(s) from day 2 to day 8
 Activity A8 requires 2 person(s) from day 9 to day 13
 Activity A7 requires 1 person(s) from day 1 to day 4
 Activity A6 requires 2 person(s) from day 9 to day 10
 Activity A5 requires 2 person(s) from day 12 to day 14
 Activity A4 requires 1 person(s) from day 11 to day 19
 Activity A3 requires 1 person(s) from day 1 to day 7
 Activity A2 requires 1 person(s) from day 1 to day 6
 Activity A1 requires 1 person(s) from day 0 to day 4
 Activity A0 requires 1 person(s) from day 0 to day 2
 Work on day 0 requires 2 out of 2 person(s)
 Work on day 1 requires 5 out of 5 person(s)
 Work on day 2 requires 5 out of 5 person(s)
 Work on day 3 requires 5 out of 5 person(s)
 Work on day 4 requires 3 out of 3 person(s)
 Work on day 5 requires 3 out of 3 person(s)
 Work on day 6 requires 2 out of 2 person(s)
 Work on day 7 requires 1 out of 1 person(s)
 Work on day 8 requires 0 out of 0 person(s)
 Work on day 9 requires 4 out of 5 person(s)
 Work on day 10 requires 2 out of 2 person(s)
 Work on day 11 requires 3 out of 4 person(s)
 Work on day 12 requires 5 out of 5 person(s)
 Work on day 13 requires 3 out of 3 person(s)
 Work on day 14 requires 1 out of 2 person(s)
 Work on day 15 requires 1 out of 1 person(s)
 Work on day 16 requires 1 out of 4 person(s)
 Work on day 17 requires 1 out of 2 person(s)
 Work on day 18 requires 1 out of 1 person(s)
 Work on day 19 requires 0 out of 3 person(s)
*/
```

Figure 35.2 displays the solution. The gray areas indicate the capacity used in the solution, and the white areas indicate unused capacity.

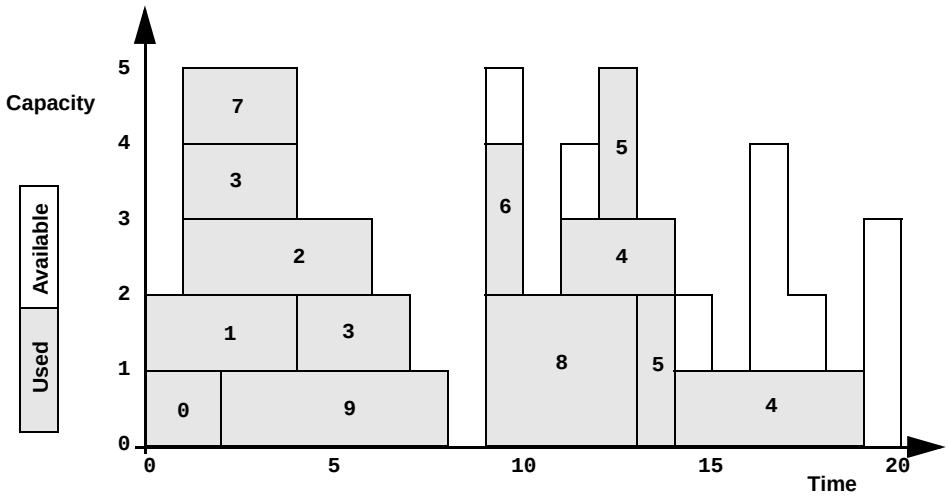


Figure 35.2 *Results with Varying Capacity*

C H A P T E R

36

Scheduling with Alternative Resources: Interleaving Resource Allocation and Scheduling Decisions

As noted in *Part I, Getting Started with Scheduler*, we can distinguish three main categories of scheduling problems:

1. Scheduling problems that are primarily concerned with placing activities over time.
2. Resource allocation problems that are concerned with assigning activities to resources.
3. Joint scheduling and resource allocation problems where both the resources assigned and the timing of the activities must be decided.

In this chapter we address the final category and, in particular, demonstrate the use of texture measurements in guiding the search. Resource allocation decisions and scheduling decisions are dynamically interleaved by using texture measurements to heuristically estimate which is more important in a particular search state.

Describing the Problem

The problem addressed in this chapter is also addressed in Chapter 28, *Tabu Search for the Jobshop Problem with Alternatives*. It is a variation on the job shop scheduling problem

using the problem instances MT06, MT10, and MT20 (see Chapter 15, *Scheduling with Unary Resources: the Job-Shop Problem*). The difference is that instead of each activity requiring only one resource, each activity has a possibility of two alternative resources. Furthermore, the processing time of the activity depends on the resource to which it is assigned.

The objective is to assign each activity to a resource and schedule the activities on each resource so that the overall makespan is minimized.

Defining the Problem, Designing a Model

Because the problem addressed here is a variation on the job shop scheduling problem, we can re-use much of modeling infrastructure for job shop problems. Rather than changing the static definition of a problem instance (that is, by predefining the resource alternatives and processing times for an activity), we use the original job shop instance as one resource alternative and dynamically generate the other alternative and processing time using a random number generator. Obviously, this is a somewhat artificial way to generate a problem, however it does demonstrate the primary difference in modeling from standard job shop models: the use of alternative resources sets and the specification of the differing processing times of an activity depending on the resource to which it is assigned. Readers who are uncomfortable with the artificiality of this example are invited to imagine replacing the calls to the random number generator with database accesses which retrieve the “real” data.

Generating the Data for the Alternative Resources

There are three important points about how the alternative resources are assigned. First, as mentioned above, the resource and processing time defined in the original job shop model are preserved in one of the alternatives. Second, the resource alternatives are generated such that the original resource requirement is replaced by an alternative resource set. That set contains two resources: the original resource defined in the job shop problem instance and another resource chosen with uniform probability from the other resources in the original job shop problem. This means that all the activities which required resource *R0* in the original problem now require either *R0* or another resource, for example, *R5*.

```
/* CREATE THE ALTERNATIVE RESOURCE SETS */
env.out() << “Creating resource sets” << endl;
IloInt *altResourceNumbers = new (env) IloInt[numberOfResources];
IloAltResSet *altResSets =
  new (env) IloAltResSet[numberOfResources];
for (j = 0; j < numberOfResources; j++) {
  altResSets[j] = IloAltResSet(env);
  altResSets[j].add(resources[j]);

  // RANDOMLY PICK ANOTHER RESOURCE TO BE IN THE SET
  assert(numberOfResources > 1);
```

```
  IloInt index = randomGenerator.getInt(numberOfResources);
  while(index == j)
    index = randomGenerator.getInt(numberOfResources);

  altResSets[j].add(resources[index]);
  altResourceNumbers[j] = index;
  env.out() << "Set #" << j << ":\t" << resources[j].getName()
            << " " << resources[index].getName() << endl;
}
```

The final point about the model generation is that the new processing time for an activity is assigned to be equal to or greater than the processing time defined in the original job shop instance. That is, the original processing time and resource are still possible for an activity. If, however, the other resource is chosen, the processing time of the activity will be equal to or greater than defined in the original problem instance.

```
/* CREATE THE ALTERNATIVE DURATIONS */
IloNum *altDurations = new (env) IloNum[numberOfActivities];
for(k = 0; k < numberOfActivities; k++) {
  IloNum multiplier = 1.0 + (randomGenerator.getFloat() / 2.0);
  altDurations[k] = IloCeil(multiplier * durations[k]);
}
```

This is a useful property as it guarantees that the optimal solution to the original job shop problem instance is an upper bound on the optimal solution to the alternative resource problem. Indeed, because the processing time is larger when the resource alternative not in the original job shop problem is selected, we can expect this to be quite a good upper bound.

Declaring the Alternative Resource Sets and Processing Times

Given the alternative resource and processing time data generated above, the only difference in modeling between the standard job shop model and this problem is the declaration of the activities and the resource constraints.

Rather than having a constant processing time, the activities in this problem have an upper and lower bound. We, therefore, create each activity with a Solver variable representing its processing time.

```
IloNum ptMin = IloMin(durations[k], altDurations[k]);
IloNum ptMax = IloMax(durations[k], altDurations[k]);
IloNumVar ptVar(env, ptMin, ptMax, ILOINT);

IloActivity activity(env, ptVar);
```

We then declare that the activity requires one of the resources in an alternative resource set by creating an alternative resource constraint and adding it to the model. To specify the differing processing times depending on the resource selected, we use the `setProcessingTimeMax` and `setProcessingTimeMin` methods on the `IloAltResConstraint`.

```
IloResourceConstraint rc =
  activity.requires(altResSets[resourceNumbers[k]]);

// SET THE DIFFERENT DURATIONS DEPENDING ON
// RESOURCE SELECTION
rc.setProcessingTimeMax(resources[resourceNumbers[k]],
                                                    durations[k]);
rc.setProcessingTimeMin(resources[resourceNumbers[k]],
                                                    durations[k]);

rc.setProcessingTimeMax(
       resources[altResourceNumbers[resourceNumbers[k]]],
       altDurations[k]);
rc.setProcessingTimeMin(
       resources[altResourceNumbers[resourceNumbers[k]]],
       altDurations[k]);

model.add(rc);
```

Solving the Problem

We solve this problem with a goal that uses texture measurements to heuristically choose between making resource allocation decisions or scheduling decisions. Due to the difficulty in the search, we arbitrarily limit the amount of time spent on solving to 200 seconds.

```
IloNum timeLimit = 200;
IloGoal goal = AltTextureIloGoal(env, makespan);
IloGoal limitedGoal = IloLimitSearch(env, goal,
                                     IloTimeLimit(env,
                                                  timeLimit));

solver.startNewSearch(limitedGoal);
while(solver.next()) {
  solver.out() << "Solution with makespan: "
               << solver.getIntVar(makespan).getValue() << endl;
  solution.store(scheduler);
  solved = IloTrue;
}

solver.endSearch();
```

Texture Measurements

As discussed in *Texture Measurement* in the *IBM ILOG Scheduler Reference Manual*, from a general perspective, a texture measurement is a measurement of some aspect of a search state. In the Scheduler Engine, we have implemented a generic resource texture measurement `IlcResourceTexture`, which maintains the criticality of each resource across the scheduling horizon. Resource criticality at a time point is a floating point value with user-definable semantics and range. You can define your own class to calculate

criticality at a time point by subclassing the `IlcTextureCriticalityCalculatorI` class. Note that the assumption is made within the `IlcResourceTexture` class that given two time points the more critical time point is the one with the greater floating point value for its criticality.

The second component of a texture measurement, after the criticality calculator, is the factory object which creates `IlcRCTexture` objects. These objects contain a curve over time which represents the extent to which a single resource constraint requires a specific resource. The user can also define the characteristics of such a requirement by subclassing `IlcRCTextureFactoryI` and `IlcRCTextureI` classes.

At the Scheduler level, there are a number of facilities for defining Concert objects that are counterparts to these Scheduler Engine objects (see, for example, the `ILORCTEXTUREFACTORY` macro). We also have a number of predefined texture objects both in the engine and in the modeling layer.

In this example, we use a predefined factory and criticality calculator and so the following code is sufficient to declare texture measurements on all the discrete resources in the model.

```
IloRCTextureFactory rcFactory = IloRCTextureProbabilisticFactory(env);
schedEnv.getTextureParam().setRCTextureFactory(rcFactory);

IloTextureCriticalityCalculator critCalc =
  IloProbabilisticCriticalityCalculator(env);
schedEnv.getTextureParam().setCriticalityCalculator(critCalc);
```

Heuristic Overview

At a search state, the heuristic decision is structured as follows:

1. Select a resource, *R*.
2. Identify a pair of unsequenced resource constraints on resource *R*.
3. Identify an alternative resource constraint on resource *R*.
4. Choose between one of the two following choice points:

   a) Constrain one of the pair of resource constraints to be before the other with the alternative being the opposite sequence.

   b) Constrain the alternative resource constraint to not execute on this resource with the alternative being that it must execute on this resource.

Selecting a Resource

A resource is chosen simply by selecting the one with the maximum criticality at any time point. Ties are broken arbitrarily.

```
IlcResourceTexture AltTextureGoalI::chooseResource() {

  IlcResourceTexture selectedTexture = 0;
  IlcFloat criticality = 0;

  for(IlcResourceIterator iter(_schedule); iter.ok(); ++iter) {
    if ((*iter).isDiscreteResource()) {
      IlcDiscreteResource res = (IlcDiscreteResource) (*iter);
      IlcResourceTexture texture = res.getTextureMeasurement();

      if (texture.getMaxCriticality() > criticality) {
        selectedTexture = texture;
        criticality = texture.getMaxCriticality();
      }
    }
  }

  return selectedTexture;
}
```

Selecting a Pair of Resource Constraints

Given the critical resource, a pair of resource constraints are identified. These resource constraints:

1. Are not alternative resource constraints.
2. Have a non-zero maximum possible requirement for the resource at its most critical time point.
3. Are not already sequenced with each other.

```
  IlcRCTextureArray rcTextures =
    texture.getCriticalityOrderedRCTextures();
  IlcInt nbRCTextures = rcTextures.getSize();

  IlcInt i;
  for(i = 0; i < nbRCTextures - 1;  ++i) {

    IlcFloat c = rcTextures[i].getCriticalContribution();
    if (c == 0)
      return IlcFalse;

    if (!rcTextures[i].hasAlternatives()) {
      rctA = rcTextures[i].getResourceConstraint();
      maxCriticality = c;

      IlcInt j;
      for(j = i+1;
          (j < nbRCTextures) &&
           (rcTextures[j].getCriticalContribution() > 0); ++j) {
```

```
      if (!rcTextures[j].hasAlternatives()) {
        rctB = rcTextures[j].getResourceConstraint();

      if (!rctA.isSucceededBy(rctB) &&
          !rctB.isSucceededBy(rctA)) {
          // found the pair to sequence
          return IlcTrue;
        }
      }
    }
  }
}

return IlcFalse;
```

The `rcTextures` array in the above code is an array of `IlcRCTexture` for each activity on the resource. The array is sorted in descending order of the contribution that the activity has for the texture curve at the critical time point. Recall that each activity has an individual texture curve and that these texture curves are aggregated to form the resource texture curve. The ordering of this array means that the pair of resource constraints that is selected consists of:

1. The resource constraint, *A*, with the highest contribution to the critical time point that meets the first two criteria above.
2. The resource constraint, *B*, with the highest contribution to the critical point that meets the first two criteria above and which is not equal to *A* nor already sequenced with *A*.

This code only selects the pair of resource constraints, *A* and *B*, not the actual sequence, *A* before *B* or *B* before *A*. The sequence is selected only after it is determined that a precedence constraint will be added at this search node. See Interleaving Resource Allocation and Scheduling.

Selecting an Alternative Resource Constraint

The alternative resource constraint that is chosen is simply the one that has the highest contribution of all alternative resource constraints to the critical time point of the resource.

```
IlcRCTextureArray rcTextures =
  texture.getCriticalityOrderedRCTextures();
IlcInt nbRCTextures = rcTextures.getSize();

for(IlcInt i = 0; i < nbRCTextures; ++i) {
  IlcFloat c = rcTextures[i].getCriticalContribution();
  if (c == 0)
    return IlcFalse;
  if (rcTextures[i].hasAlternatives()) {
    // rcTextures is sorted in descending order of
    // criticality so the first rct we find with
    // alternatives is the one with the highest crit.
    altRC = rcTextures[i].getResourceConstraint();
    criticality = c;
    return IlcTrue;
  }
```

```
}

return IlcFalse;
```

Interleaving Resource Allocation and Scheduling

The final decision is to choose whether the scheduling decision (that is, sequencing of the pair of resource constraints) or the resource allocation decision (constraining the alternative resource constraint to execute on another resource) should be made. This is done by comparing the maximum criticality of a resource constraint in the pair with that of the alternative resource constraint. The decision accompanying the resource constraint with higher criticality is chosen.

```
IlcBool pairFound = choosePair(selectedTexture,
                               rctA, rctB, maxPairCriticality);
IlcBool altFound = chooseAlternative(selectedTexture,
                                     rctC, altCriticality);

if (!pairFound && !altFound) {
  // If neither a pair nor an alternative is found, then
  // there are no commitments that this goal can make at
  // the critical point.
  selectedTexture.setNoCommitmentsAtCriticalPoint();
  selectedTexture = chooseResource();
}
else {
  if (!pairFound ||
      (altFound && (maxPairCriticality <= altCriticality)))
    return IlcAnd(IlcTryNotPossible(rctC.getResource(),
                                    rctC.getAlternative()),
                  this);
  else {
    IlcResourceConstraint before, after;
    IlcBool sequenceOK = chooseSequence(before, after,
                                        rctA, rctB);
    assert(sequenceOK);

    return IlcAnd(IlcTrySetSuccessor(before, after), this);
  }
}
```

The `chooseSequence` function chooses the actual sequence of the two resource constraints which will be tried in the first branch of the OR-goal. This calculation is done based on an analysis of the slack time that results from the two possible sequences. Further information can be found in the paper that is referenced in the full code of the example.

The Possibility of No Commitments

The previous code sample accounts for the possibility that neither an alternative resource constraint nor a pair of unsequenced resource constraints could be found.

This can happen because the texture measurement is an estimate of the criticality of the resource. The fact that no possible commitments could be found at the critical time point means that all resource constraints that can possibly execute at that time point have no

alternatives and are completely sequenced. Therefore, the criticality of the resource at the time point is actually 0. This fact can be communicated to the texture measurement data structure using the `setNoCommitmentsAtCriticalPoint()` method. This method identifies the largest temporal interval for which the set of possible resource constraints is equal to the set of possible resource constraints at the critical time point. The criticality of that interval is set to 0.

Complete Program and Output

You can see the entire program `alttext.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

IloInt ResourceNumbers06 [] = {2, 0, 1, 3, 5, 4,
                               1, 2, 4, 5, 0, 3,
                               2, 3, 5, 0, 1, 4,
                               1, 0, 2, 3, 4, 5,
                               2, 1, 4, 5, 0, 3,
                               1, 3, 5, 0, 4, 2};

IloNum Durations06 [] = { 1,  3,  6,  7,  3,  6,
                          8,  5, 10, 10, 10,  4,
                          5,  4,  8,  9,  1,  7,
                          5,  5,  5,  3,  8,  9,
                          9,  3,  5,  4,  3,  1,
                          3,  3,  9, 10,  4,  1};

IloInt ResourceNumbers10 [] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
                               0, 2, 4, 9, 3, 1, 6, 5, 7, 8,
                               1, 0, 3, 2, 8, 5, 7, 6, 9, 4,
                               1, 2, 0, 4, 6, 8, 7, 3, 9, 5,
                               2, 0, 1, 5, 3, 4, 8, 7, 9, 6,
                               2, 1, 5, 3, 8, 9, 0, 6, 4, 7,
                               1, 0, 3, 2, 6, 5, 9, 8, 7, 4,
                               2, 0, 1, 5, 4, 6, 8, 9, 7, 3,
                               0, 1, 3, 5, 2, 9, 6, 7, 4, 8,
                               1, 0, 2, 6, 8, 9, 5, 3, 4, 7};

IloNum Durations10 [] = {29, 78,  9, 36, 49, 11, 62, 56, 44, 21,
                         43, 90, 75, 11, 69, 28, 46, 46, 72, 30,
                         91, 85, 39, 74, 90, 10, 12, 89, 45, 33,
                         81, 95, 71, 99,  9, 52, 85, 98, 22, 43,
                         14,  6, 22, 61, 26, 69, 21, 49, 72, 53,
                         84,  2, 52, 95, 48, 72, 47, 65,  6, 25,
                         46, 37, 61, 13, 32, 21, 32, 89, 30, 55,
                         31, 86, 46, 74, 32, 88, 19, 48, 36, 79,
                         76, 69, 76, 51, 85, 11, 40, 89, 26, 74,
                         85, 13, 61,  7, 64, 76, 47, 52, 90, 45};

IloInt ResourceNumbers20 [] = {0, 1, 2, 3, 4,
```

```
                                0, 1, 3, 2, 4,
                                1, 0, 2, 4, 3,
                                1, 0, 4, 2, 3,
                                2, 1, 0, 3, 4,
                                2, 1, 4, 0, 3,
                                1, 0, 2, 3, 4,
                                2, 1, 0, 3, 4,
                                0, 3, 2, 1, 4,
                                1, 2, 0, 3, 4,
                                1, 3, 0, 4, 2,
                                2, 0, 1, 3, 4,
                                0, 2, 1, 3, 4,
                                2, 0, 1, 3, 4,
                                0, 1, 4, 2, 3,
                                1, 0, 3, 4, 2,
                                0, 2, 1, 3, 4,
                                0, 1, 4, 2, 3,
                                1, 2, 0, 3, 4,
                                0, 1, 2, 3, 4};

IloNum Durations20 [] = {29,  9, 49, 62, 44,
                         43, 75, 69, 46, 72,
                         91, 39, 90, 12, 45,
                         81, 71,  9, 85, 22,
                         14, 22, 26, 21, 72,
                         84, 52, 48, 47,  6,
                         46, 61, 32, 32, 30,
                         31, 46, 32, 19, 36,
                         76, 76, 85, 40, 26,
                         85, 61, 64, 47, 90,
                         78, 36, 11, 56, 21,
                         90, 11, 28, 46, 30,
                         85, 74, 10, 89, 33,
                         95, 99, 52, 98, 43,
                          6, 61, 69, 49, 53,
                          2, 95, 72, 65, 25,
                         37, 13, 21, 89, 55,
                         86, 74, 88, 48, 79,
                         69, 51, 11, 89, 74,
                         13,  7, 76, 52, 45};


////////////////////////////////////////////////////////////////////
//
// TEXTURE CHOICE POINT FOR SCHEDULING WITH ALTERNATIVES
//
////////////////////////////////////////////////////////////////////

class AltTextureGoalI : public IlcGoalI {
private:
  IlcSchedule _schedule;

protected:
  IlcResourceTexture chooseResource();
  IlcBool choosePair(IlcResourceTexture,
                     IlcResourceConstraint&,
                     IlcResourceConstraint&, IlcFloat&);
  IlcBool chooseSequence(IlcResourceConstraint&,
```

```
                        IlcResourceConstraint&,
                        IlcResourceConstraint,
                        IlcResourceConstraint);
  IlcBool calcBiasedSlack(IlcResourceConstraint,
                          IlcResourceConstraint,
                          IlcFloat&, IlcFloat&);
  IlcBool chooseAlternative(IlcResourceTexture,
                            IlcResourceConstraint&, IlcFloat&);

public:
  AltTextureGoalI(IlcSchedule s)
    : IlcGoalI(s.getSolver()), _schedule(s) {}
  ~AltTextureGoalI() {}

  virtual IlcGoal execute();
};

IlcGoal AltTextureGoalI::execute() {

  IlcResourceTexture selectedTexture = chooseResource();
  while(0 != selectedTexture.getImpl()) {

    IlcResourceConstraint rctA, rctB, rctC;
    IlcFloat maxPairCriticality = 0;
    IlcFloat altCriticality = 0;
    IlcBool pairFound = choosePair(selectedTexture,
                                   rctA, rctB, maxPairCriticality);
    IlcBool altFound = chooseAlternative(selectedTexture,
                                         rctC, altCriticality);

    if (!pairFound && !altFound) {
      // If neither a pair nor an alternative is found, then
      // there are no commitments that this goal can make at
      // the critical point.
      selectedTexture.setNoCommitmentsAtCriticalPoint();
      selectedTexture = chooseResource();
    }
    else {
      if (!pairFound ||
          (altFound && (maxPairCriticality <= altCriticality)))
        return IlcAnd(IlcTryNotPossible(rctC.getResource(),
                                        rctC.getAlternative()),
                      this);
      else {
        IlcResourceConstraint before, after;
        IlcBool sequenceOK = chooseSequence(before, after,
                                            rctA, rctB);
        assert(sequenceOK);

        return IlcAnd(IlcTrySetSuccessor(before, after), this);
      }
    }
  }

  return 0;
}
```

```
IlcResourceTexture AltTextureGoalI::chooseResource() {

  IlcResourceTexture selectedTexture = 0;
  IlcFloat criticality = 0;

  for(IlcResourceIterator iter(_schedule); iter.ok(); ++iter) {
    if ((*iter).isDiscreteResource()) {
      IlcDiscreteResource res = (IlcDiscreteResource) (*iter);
      IlcResourceTexture texture = res.getTextureMeasurement();

      if (texture.getMaxCriticality() > criticality) {
        selectedTexture = texture;
        criticality = texture.getMaxCriticality();
      }
    }
  }

  return selectedTexture;
}

IlcBool AltTextureGoalI::choosePair(IlcResourceTexture texture,
                                    IlcResourceConstraint& rctA,
                                    IlcResourceConstraint& rctB,
                                    IlcFloat& maxCriticality) {

  IlcRCTextureArray rcTextures =
    texture.getCriticalityOrderedRCTextures();
  IlcInt nbRCTextures = rcTextures.getSize();

  IlcInt i;
  for(i = 0; i < nbRCTextures - 1;  ++i) {

    IlcFloat c = rcTextures[i].getCriticalContribution();
    if (c == 0)
      return IlcFalse;

    if (!rcTextures[i].hasAlternatives()) {
      rctA = rcTextures[i].getResourceConstraint();
      maxCriticality = c;

      IlcInt j;
      for(j = i+1;
          (j < nbRCTextures) &&
           (rcTextures[j].getCriticalContribution() > 0); ++j) {

        if (!rcTextures[j].hasAlternatives()) {
          rctB = rcTextures[j].getResourceConstraint();

        if (!rctA.isSucceededBy(rctB) &&
            !rctB.isSucceededBy(rctA)) {
            // found the pair to sequence
            return IlcTrue;
          }
        }
      }
    }
  }
```

```
  return IlcFalse;
}

IlcBool AltTextureGoalI::chooseSequence(
                  IlcResourceConstraint& selectedRct1,
                  IlcResourceConstraint& selectedRct2,
                  IlcResourceConstraint rctA,
                  IlcResourceConstraint rctB) {

  IlcFloat biasedSlackBA, biasedSlackAB;
  if (!calcBiasedSlack(rctA, rctB,
                       biasedSlackAB, biasedSlackBA))
    return IlcFalse;

  selectedRct1 = rctA;
  selectedRct2 = rctB;

  // pick sequence that preserves the most slack
  if (biasedSlackAB < biasedSlackBA) {
    selectedRct1 = rctB;
    selectedRct2 = rctA;
  }

  return IlcTrue;
}

IlcBool AltTextureGoalI::calcBiasedSlack(
                        IlcResourceConstraint rctA,
                        IlcResourceConstraint rctB,
                        IlcFloat& biasedSlackAB,
                        IlcFloat& biasedSlackBA)
{
  // The PCPSlack sequencing calculation from:
  // @InProceedings{Cheng93,
  //   author = “Smith, S. F. and Cheng, C. C.”,
  //   title = “Slack-based heuristics for constraint satisfaction
  //            scheduling”,
  //   key = “scheduling”,
  //   year = “1993”,
  //   pages = “139--144”,
  //   booktitle = “Proceedings of the Eleventh National
  //                Conference on Artificial Intelligence
  //                (AAAI-93)”,
  // }

  // Returns IlcTrue if the pair has a valid biasedSlack value.
  // IlcFalse means there is no valid value for this pair (for
  // example, because they do not overlap).

  IlcActivity actA = rctA.getActivity();
  IlcActivity actB = rctB.getActivity();

  IlcInt estA = actA.getStartMin();
  IlcInt lftA = actA.getEndMax();
  IlcInt estB = actB.getStartMin();
  IlcInt lftB = actB.getEndMax();

  if ((lftA <= estB) || (lftB <= estA))
```

```
    // Activities  already sequenced
    return IlcFalse;

  IlcInt lambda = lftA - estB;
  IlcInt mu = lftB - estA;
  IlcInt delta = actA.getProcessingTimeMin() +
    actB.getProcessingTimeMin();

  IlcFloat slackAB = mu - delta;
  IlcFloat slackBA = lambda - delta;
  IlcFloat rtS;

  // Either (or both) of slackAB and slackBA might be 0!
  // Assign to 0.1 (note that lambda, delta, and mu are
  // all integral it is not possible for slackAB or
  // slackBA to actually be 0.1 by themselves)
  if (mu == delta)
    slackAB = 0.1;

  if (lambda == delta)
    slackBA = 0.1;

  if (slackAB < slackBA)
    rtS = sqrt(slackAB / slackBA);
  else
    rtS = sqrt(slackBA / slackAB);

  biasedSlackAB = slackAB / rtS;
  biasedSlackBA = slackBA / rtS;

  return IlcTrue;
}

IlcBool AltTextureGoalI::chooseAlternative(
                           IlcResourceTexture texture,
                           IlcResourceConstraint& altRC,
                           IlcFloat& criticality) {

  IlcRCTextureArray rcTextures =
    texture.getCriticalityOrderedRCTextures();
  IlcInt nbRCTextures = rcTextures.getSize();

  for(IlcInt i = 0; i < nbRCTextures; ++i) {
    IlcFloat c = rcTextures[i].getCriticalContribution();
    if (c == 0)
      return IlcFalse;
    if (rcTextures[i].hasAlternatives()) {
      // rcTextures is sorted in descending order of
      // criticality so the first rct we find with
      // alternatives is the one with the highest crit.
      altRC = rcTextures[i].getResourceConstraint();
      criticality = c;
      return IlcTrue;
    }
  }

  return IlcFalse;
}
```

```
ILOCPGOALWRAPPER1(AltTextureIloGoal, solver, IloNumVar, cost) {
  IlcScheduler scheduler(solver);
  return IlcAnd(IlcGoal(new (solver.getHeap())
                                AltTextureGoalI(scheduler)),
                IlcInstantiate(solver.getIntVar(cost)));
}

////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////////

void PrintRange(IloEnv& env, IloNum min, IloNum max) {
  if (min == max)
    env.out() << (IlcInt)min;
  else
    env.out() << (IlcInt)min << "..” << (IlcInt)max;
}

void PrintSolution(IloEnv& env,
                   const IloSchedulerSolution solution,
                   const IloNumVar makespan)
{
  env.out() << “Solution with makespan [“
            << solution.getMin(makespan) << “..”
            << solution.getMax(makespan) << “]” << endl;

  for (IloSchedulerSolution::ResourceConstraintIterator
                                                          iter(solution);
       iter.ok();
       ++iter)
  {
    IloResourceConstraint rc = *iter;
    if (!solution.isResourceSelected(rc))
      IloSchedulerException(“No resource assigned!”);

    IloActivity activity = rc.getActivity();
    env.out() << activity.getName() << “[“;
    PrintRange(env,
               solution.getStartMin(activity),
               solution.getStartMax(activity));
    env.out() << “ -- “;
    PrintRange(env,
               solution.getDurationMin(activity),
               solution.getDurationMax(activity));
    env.out() << “ --> “;
    PrintRange(env,
               solution.getEndMin(activity),
               solution.getEndMax(activity));
    env.out() << “]: “ << solution.getSelected(rc).getName()
              << endl;
  }
}

////////////////////////////////////////////////////////////////////
```

```
//
// SOLVE THE MODEL USING TEXTURE-BASED HEURISTIC
//
////////////////////////////////////////////////////////////////////
IloBool SolveModel(IloModel model, IloNumVar makespan,
                   IloSchedulerSolution solution) {

  IloEnv env = model.getEnv();
  IloSolver solver(model);
  IlcScheduler scheduler(solver);
  IloBool solved = IloFalse;

  IloNum timeLimit = 200;
  IloGoal goal = AltTextureIloGoal(env, makespan);
  IloGoal limitedGoal = IloLimitSearch(env, goal,
                                       IloTimeLimit(env,
                                                    timeLimit));

  solver.startNewSearch(limitedGoal);
  while(solver.next()) {
    solver.out() << "Solution with makespan: "
                 << solver.getIntVar(makespan).getValue() << endl;
    solution.store(scheduler);
    solved = IloTrue;
  }

  solver.endSearch();

  solver.printInformation();
  solver.end();
  return solved;
}

////////////////////////////////////////////////////////////////////
//
// DEFINE THE MODEL WITH ALTERNATIVE RESOURCES
//
////////////////////////////////////////////////////////////////////
IloModel
DefineModel(IloEnv& env,
            IloInt numberOfJobs,
            IloInt numberOfResources,
            IloInt* resourceNumbers,
            IloNum* durations,
            IloRandom randomGenerator,
            IloSchedulerSolution solution,
            IloNumVar& makespan)
{
  IloModel model(env);

  /* CREATE THE MAKESPAN VARIABLE. */
  IloInt numberOfActivities = numberOfJobs * numberOfResources;
  IloNum horizon = 0;
  IloInt k;

  for (k = 0; k < numberOfActivities; k++)
    horizon += durations[k];
```

```
makespan = IloNumVar(env, 0, horizon, ILOINT);

/* CREATE THE RESOURCES. */
IloSchedulerEnv schedEnv(env);
schedEnv.getResourceParam().
  setCapacityEnforcement(IloMediumHigh);
schedEnv.getResourceParam().
  setPrecedenceEnforcement(IloMediumHigh);

IloRCTextureFactory rcFactory = IloRCTextureProbabilisticFactory(env);
schedEnv.getTextureParam().setRCTextureFactory(rcFactory);

IloTextureCriticalityCalculator critCalc =
  IloProbabilisticCriticalityCalculator(env);
schedEnv.getTextureParam().setCriticalityCalculator(critCalc);

char buffer[128];
IloInt j;
IloUnaryResource *resources =
  new (env) IloUnaryResource[numberOfResources];
for (j = 0; j < numberOfResources; j++) {
  sprintf(buffer, "R%ld", j);
  resources[j] = IloUnaryResource(env, buffer);
}

/* CREATE THE ALTERNATIVE RESOURCE SETS */
env.out() << "Creating resource sets" << endl;
IloInt *altResourceNumbers = new (env) IloInt[numberOfResources];
IloAltResSet *altResSets =
  new (env) IloAltResSet[numberOfResources];
for (j = 0; j < numberOfResources; j++) {
  altResSets[j] = IloAltResSet(env);
  altResSets[j].add(resources[j]);

  // RANDOMLY PICK ANOTHER RESOURCE TO BE IN THE SET
  assert(numberOfResources > 1);
  IloInt index = randomGenerator.getInt(numberOfResources);
  while(index == j)
    index = randomGenerator.getInt(numberOfResources);

  altResSets[j].add(resources[index]);
  altResourceNumbers[j] = index;
  env.out() << "Set #" << j << ":\t" << resources[j].getName()
            << " " << resources[index].getName() << endl;
}

/* CREATE THE ALTERNATIVE DURATIONS */
IloNum *altDurations = new (env) IloNum[numberOfActivities];
for(k = 0; k < numberOfActivities; k++) {
  IloNum multiplier = 1.0 + (randomGenerator.getFloat() / 2.0);
  altDurations[k] = IloCeil(multiplier * durations[k]);
}

/* CREATE THE ACTIVITIES. */
env.out() << "Setting alternative processing times" << endl;
k = 0;
IloInt i;
for (i = 0; i < numberOfJobs; i++) {
```

```
    IloActivity previousActivity;
    for (j = 0; j < numberOfResources; j++) {

      IloNum ptMin = IloMin(durations[k], altDurations[k]);
      IloNum ptMax = IloMax(durations[k], altDurations[k]);
      IloNumVar ptVar(env, ptMin, ptMax, ILOINT);

      IloActivity activity(env, ptVar);

      sprintf(buffer, "J%ldS%ldR%ld", i, j, resourceNumbers[k]);
      activity.setName(buffer);
      solution.add(activity);

      IloResourceConstraint rc =
        activity.requires(altResSets[resourceNumbers[k]]);

      // SET THE DIFFERENT DURATIONS DEPENDING ON
      // RESOURCE SELECTION
      rc.setProcessingTimeMax(resources[resourceNumbers[k]],
                                                              durations[k]);
      rc.setProcessingTimeMin(resources[resourceNumbers[k]],
                                                              durations[k]);

      rc.setProcessingTimeMax(
             resources[altResourceNumbers[resourceNumbers[k]]],
             altDurations[k]);
      rc.setProcessingTimeMin(
             resources[altResourceNumbers[resourceNumbers[k]]],
             altDurations[k]);

      model.add(rc);

      solution.add(rc);
      env.out() << activity.getName()
                << ":\tProcessing Time("
                << resources[resourceNumbers[k]].getName()
                << "): " << durations[k]
                << "\n\tProcessing Time("
                << resources[altResourceNumbers[
                                  resourceNumbers[k]]].getName()
                << "): " << altDurations[k]
                << endl;


      if (j != 0)
        model.add(activity.startsAfterEnd(previousActivity));
      previousActivity = activity;
      k++;
    }
    model.add(previousActivity.endsBefore(makespan));
  }

  model.add(IloMinimize(env, makespan));
  solution.getSolution().add(makespan);

  /* RETURN THE MODEL. */
  return model;
}
```

```
////////////////////////////////////////////////////////////////
//
// INITIALIZE THE PROGRAM ARGUMENTS
//
////////////////////////////////////////////////////////////////
void
InitParameters(int argc,
               char** argv,
               IloInt& numberOfJobs,
               IloInt& numberOfResources,
               IloInt*& resourceNumbers,
               IloNum*& durations)
{
  numberOfJobs = 6;
  numberOfResources = 6;
  resourceNumbers = ResourceNumbers06;
  durations = Durations06;

  if (argc > 1) {
    IloInt number = atol(argv[1]);
    if (number == 10) {
      numberOfJobs = 10;
      numberOfResources = 10;
      resourceNumbers = ResourceNumbers10;
      durations = Durations10;
    }
    else if (number == 20) {
      numberOfJobs = 20;
      numberOfResources = 5;
      resourceNumbers = ResourceNumbers20;
      durations = Durations20;
    }
  }
}

////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////

int main(int argc, char** argv)
{
  IloInt numberOfJobs;
  IloInt numberOfResources;
  IloInt* resourceNumbers;
  IloNum* durations;

  InitParameters(argc,
                 argv,
                 numberOfJobs,
                 numberOfResources,
                 resourceNumbers,
                 durations);

  try {
```

```
    IloEnv env;
    IloNumVar makespan;
    IloRandom randGen(env, 8975324);
    IloSchedulerSolution solution(env);
    IloModel model = DefineModel(env,
                                 numberOfJobs,
                                 numberOfResources,
                                 resourceNumbers,
                                 durations,
                                 randGen,
                                 solution,
                                 makespan);

    IloBool solved = SolveModel(model, makespan, solution);
    if (!solved)
      throw IloSchedulerException( “No solution found” );

    PrintSolution(env, solution, makespan);
    env.end();

  }
  catch (IloSchedulerException& exc) {
    cout << exc << endl;
  }
  catch (IloException& exc) {
    cout << exc << endl;
  }

  return 0;
}

/*
% alttext
Creating resource sets
Set #0: R0 R5
Set #1: R1 R2
Set #2: R2 R3
Set #3: R3 R0
Set #4: R4 R1
Set #5: R5 R2
Setting alternative processing times
J0S0R2: Processing Time(R2): 1
        Processing Time(R3): 2
J0S1R0: Processing Time(R0): 3
        Processing Time(R5): 4
J0S2R1: Processing Time(R1): 6
        Processing Time(R2): 9
J0S3R3: Processing Time(R3): 7
        Processing Time(R0): 9
J0S4R5: Processing Time(R5): 3
        Processing Time(R2): 5
J0S5R4: Processing Time(R4): 6
        Processing Time(R1): 7
J1S0R1: Processing Time(R1): 8
        Processing Time(R2): 9
J1S1R2: Processing Time(R2): 5
        Processing Time(R3): 7
```

```
J1S2R4: Processing Time(R4): 10
        Processing Time(R1): 14
J1S3R5: Processing Time(R5): 10
        Processing Time(R2): 11
J1S4R0: Processing Time(R0): 10
        Processing Time(R5): 13
J1S5R3: Processing Time(R3): 4
        Processing Time(R0): 6
J2S0R2: Processing Time(R2): 5
        Processing Time(R3): 6
J2S1R3: Processing Time(R3): 4
        Processing Time(R0): 6
J2S2R5: Processing Time(R5): 8
        Processing Time(R2): 11
J2S3R0: Processing Time(R0): 9
        Processing Time(R5): 12
J2S4R1: Processing Time(R1): 1
        Processing Time(R2): 2
J2S5R4: Processing Time(R4): 7
        Processing Time(R1): 8
J3S0R1: Processing Time(R1): 5
        Processing Time(R2): 7
J3S1R0: Processing Time(R0): 5
        Processing Time(R5): 6
J3S2R2: Processing Time(R2): 5
        Processing Time(R3): 6
J3S3R3: Processing Time(R3): 3
        Processing Time(R0): 4
J3S4R4: Processing Time(R4): 8
        Processing Time(R1): 9
J3S5R5: Processing Time(R5): 9
        Processing Time(R2): 11
J4S0R2: Processing Time(R2): 9
        Processing Time(R3): 14
J4S1R1: Processing Time(R1): 3
        Processing Time(R2): 4
J4S2R4: Processing Time(R4): 5
        Processing Time(R1): 8
J4S3R5: Processing Time(R5): 4
        Processing Time(R2): 6
J4S4R0: Processing Time(R0): 3
        Processing Time(R5): 4
J4S5R3: Processing Time(R3): 1
        Processing Time(R0): 2
J5S0R1: Processing Time(R1): 3
        Processing Time(R2): 4
J5S1R3: Processing Time(R3): 3
        Processing Time(R0): 5
J5S2R5: Processing Time(R5): 9
        Processing Time(R2): 11
J5S3R0: Processing Time(R0): 10
        Processing Time(R5): 11
J5S4R4: Processing Time(R4): 4
        Processing Time(R1): 6
J5S5R2: Processing Time(R2): 1
        Processing Time(R3): 2
ILOG Scheduler 5.000, licensed to “ILOG Gentilly”.
ILOG Solver 5.000, licensed to “ILOG Gentilly”, options: LS
```

```
Solution with makespan: 61
Solution with makespan: 59
Solution with makespan: 58
Solution with makespan: 57
Solution with makespan: 56
Solution with makespan: 54
Solution with makespan: 53
Solution with makespan: 51
Solution with makespan: 50
Solution with makespan: 49
Solution with makespan: 48
Number of fails               : 260
Number of choice points       : 270
Number of variables           : 146
Number of constraints         : 0
Reversible stack (bytes)      : 88464
Solver heap (bytes)           : 172884
Solver global heap (bytes)    : 237204
And stack (bytes)             : 4044
Or stack (bytes)              : 8064
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 11144
Total memory used (bytes)     : 525848
Running time since creation   : 0.73
Solution with makespan [48..48]
J0S0R2[0 -- 2 --> 2]: R3
J0S1R0[2 -- 4 --> 6]: R5
J0S2R1[8..19 -- 6 --> 14..25]: R1
J0S3R3[25 -- 7 --> 32]: R3
J0S4R5[32 -- 5 --> 37]: R2
J0S5R4[37..41 -- 7 --> 44..48]: R1
J1S0R1[0 -- 9 --> 9]: R2
J1S1R2[9 -- 5 --> 14]: R2
J1S2R4[14 -- 10 --> 24]: R4
J1S3R5[24 -- 10 --> 34]: R5
J1S4R0[34 -- 10 --> 44]: R0
J1S5R3[44 -- 4 --> 48]: R3
J2S0R2[6 -- 6 --> 12]: R3
J2S1R3[12 -- 4 --> 16]: R3
J2S2R5[16 -- 8 --> 24]: R5
J2S3R0[25 -- 9 --> 34]: R0
J2S4R1[34..40 -- 1 --> 35..41]: R1
J2S5R4[37..41 -- 7 --> 44..48]: R4
J3S0R1[3..5 -- 5 --> 8..10]: R1
J3S1R0[8..10 -- 5 --> 13..15]: R0
J3S2R2[16 -- 6 --> 22]: R3
J3S3R3[22 -- 3 --> 25]: R3
J3S4R4[25..29 -- 8 --> 33..37]: R4
J3S5R5[37 -- 11 --> 48]: R2
J4S0R2[14..18 -- 9 --> 23..27]: R2
J4S1R1[23..27 -- 3 --> 26..30]: R1
J4S2R4[26..30 -- 8 --> 34..38]: R1
J4S3R5[34..38 -- 4 --> 38..42]: R5
J4S4R0[38..42 -- 4 --> 42..46]: R5
J4S5R3[44..46 -- 2 --> 46..48]: R0
J5S0R1[0 -- 3 --> 3]: R1
J5S1R3[3 -- 3 --> 6]: R3
J5S2R5[6 -- 9 --> 15]: R5
```

```
J5S3R0[15 -- 10 --> 25]: R0
J5S4R4[33..37 -- 4 --> 37..41]: R4
J5S5R2[37..42 -- 2 --> 39..44]: R3
*/
```

C H A P T E R

37

Incremental Scheduling: Using Scheduler with Multiple Problem Sub-models

In many real scheduling applications, the scheduling model is not static. There may be an initial problem to solve and then a series of changes such as resource breakdowns or the arrival of new orders to schedule that must be incorporated. With such an application, especially one of reasonable size, it is unrealistic to re-solve the entire problem from scratch. Such a re-solve may take too long, especially if the schedule is in the process of being executed, and the new solution may have little in common with the original solution.

Similarity between consecutive solutions is important when a schedule has external impact. For example, if the original schedule was used by an inventory manager to arrange the delivery of raw materials, radically changing it can have costly repercussions through-out a company's supply chain.

The goal of this chapter is to demonstrate how Scheduler can be applied to a situation where jobs arrive incrementally and the desire is to incorporate the new jobs into the existing scheduling, while satisfying the twin objectives of achieving high schedule-quality and minimizing changes to the existing schedule. To do this we make significant use of the modeling capabilities of Concert Technology, maintaining multiple models of parts of the overall scheduling problem. The communication among these models forms a significant part of the solution to our problem, and therefore this chapter serves as an example of how to solve scheduling problems by combining the facilities of the Scheduler Engine with the modeling capabilities of IBM© ILOG© Concert Technology.

Describing the Problem

The initial problem is to schedule a set of jobs on a set of discrete resources so as to minimize the overall makespan. After this initial problem is solved, new jobs are added, one at a time, simulating the arrival of new orders in a factory. When each new job arrives, it is integrated into the existing schedule with the twin optimization criteria of minimizing the changes that must be made to the existing schedule while also minimizing the overall makespan.

An obvious approach to this problem would be to reschedule from scratch each time a new job is added. It would be possible to explicitly deal with the two optimization criteria by cleverly expressing the fact that the solution to the expanded problem should, as much as possible, retain the start times assigned in the existing schedule. To add a further level of verisimilitude, however, we further assume that due to time and memory limitations, we cannot solve a single monolithic model of the expanded scheduling problem. Rather, we must process the model of the expanded problem to create a number of inter-related sub-models with a fraction of the resources and activities in the whole problem. We can then solve each of these sub-models in sequence and combine the solutions into a solution to the whole problem.

It should be noted that the actual problem instances that we solve in this chapter are not really too big for the expanded model to be solved as a whole problem without sub-models. Indeed, as we will see, in order to compare the performance of the sub-model approach with the single model approach, at the end of the example we solve the final problem as a single model. The size of the problem instances reflects the desire to keep the example to a manageable size for the reader.

In real applications of IBM© ILOG© Scheduler, with hundreds of resources, thousands of activities, and tight time requirements, problem decomposition into sub-models is a very important and often-used technique. This chapter is designed to present an example of this technique so that users can then extend it to problem instances where it is necessary.

Defining the Problem

The basis of the problem solved in this chapter is the job shop scheduling model. We use the job shop examples introduced in Chapter 15, *Scheduling with Unary Resources: the Job-Shop Problem*, with two changes:

1. Rather than using unary resources, we use discrete resources with a capacity of 3.

2. Rather than having one instantiation of each job, the original problem (before the new jobs are incrementally added) has 3 instantiations of each job from the underlying job shop example. So, for example, the underlying jobshop problem corresponding to our smallest instance has 6 jobs each with 6 activities. The original multicapacity problem that we solve has 18 jobs each with 6 activities.

These variations make the problem instances larger and more interesting for searching for incremental solutions.

Adding New Jobs

The source of the incrementality in this example is the arrival of a new job to be scheduled. In a real application, such an arrival may correspond to a new order from a customer. In our example, we add n new jobs, where *n* is the number of jobs in the underlying job shop problem. Since the original multicapacity scheduling problem has *3n* jobs, the final problem contains *4n* jobs.

Each new job is a variation of a job defined in the original problem. Initially, the resource requirements and durations of the new job are identical to the original job. However, to add some variety, we regenerate the duration and randomly permute the resource requirement of each activity. The durations are selected with uniform probability from the interval of durations of the original job, as shown in the following code.

```
IloNum minDur = IloInfinity;
IloNum maxDur = -IloInfinity;
IloInt i;
for(i = 0; i < numberOfJobs * numberOfResources; ++i) {
  if (existingDurations[i] < minDur)
    minDur = existingDurations[i];
  if (existingDurations[i] > maxDur)
    maxDur = existingDurations[i];
}

IloNum diff = maxDur - minDur + 1;
IloNumArray durations(env, numberOfResources);
for(i = 0; i < numberOfResources; ++i)
  durations[i] = randGen.getInt(IloTrunc(diff)) + minDur;
```

The resource requirements of the activities in the new job are randomly permuted.

```
IloArray<IloDiscreteResource> resources(env, numberOfResources);
i = 0;
for(IloIterator<IloDiscreteResource> iter(env); iter.ok(); ++iter) {
  resources[i] = *iter;
  ++i;
}

// RANDOMLY PERMUTE RESOURCE REQUIREMENTS
for(i = 0; i < 1000; ++i) {
  IloInt index1 = randGen.getInt(numberOfResources);
  IloInt index2 = randGen.getInt(numberOfResources);
  while (index1 == index2)
    index2 = randGen.getInt(numberOfResources);
  IloDiscreteResource tmp = resources[index1];
  resources[index1] = resources[index2];
  resources[index2] = tmp;
}
```

The following table provides a summary of the example problems that we use in this chapter.

Table 37.1 *Summary of the Example Problems*

Underlying job shop instance	Number of jobs in job shop instance	Number of jobs in original multicapacity problem	Number of jobs in final problem
MT06	6	18	24
MT10	10	30	40
MT20	20	60	80

Designing the Models

Unlike many of the examples in this manual, a significant part of the actual problem solving for this example rests not in the non-deterministic search facilities of the Scheduler Engine and IBM© ILOG© Solver, but rather in the formulation, coordination, and manipulation of multiple problem models. While the non-deterministic search facilities are still needed, the model manipulation functionality supplied by IBM ILOG Concert Technology is relied on heavily.

In this section, we discuss the models that are created and how information is communicated amongst them. We do not discuss how each submodel is solved using the Scheduler Engine, but rather concentrate on the level of Concert Technology. Solving the Models presents how each model is solved using the Scheduler Engine.

Modeling the Initial Problem

The initial problem is similar to the job shop problems we have examined in previous chapters. The differences are the number of activities and the use of discrete resources with capacity 3.

Therefore, the modeling of the original problem is quite straightforward. Users are referred to the function `CreateOriginalModel` and to Chapter 15, *Scheduling with Unary Resources: the Job-Shop Problem* where the modeling of a job shop problem is described.

When a new job is created it is added to the initial model. Our assumption is that this model (the original model plus the new job) is too large to be solved as a single problem given the time and/or memory limitations of a real application. However, this does not prevent us from having a model of the entire problem because:

1) We do not solve this model as a whole.

2) A model in Concert Technology has a significantly lighter memory consumption than the same model extracted by a solver.

Therefore, once we have a solution to the original problem (see Solving the Original Model) and have added a new job to the original model, our task is to design secondary models that can be extracted and solved with a small memory and CPU time impact. The solutions to these submodels will then be efficiently combined into a solution for the overall problem. Thus, without actually having to extract and solve the overall problem, we will form a solution for it.

Each time we add a new job, the overall model and the solution to it (found via the combination of sub-model solutions) is changed. We will refer to the overall model, therefore, as the *evolving* model in that at any time it contains the model of the original problem plus whatever new jobs have been added: it has evolved over time. Similarly, we will refer to the solution to the evolving model as the *evolving* solution.

Creating the Default Solution

Given the existing solution and the new job that is to be incorporated into it, we can calculate an upper-bound on the makespan of the new model by adding the makespan from the existing solution to the sum of the durations of the activities in the new job. This is true since we can find a new solution simply by copying the existing solution and placing the activities of the new job as the final activities on each resource. This will extend the makespan by, at most, the sum of the durations of the activities of the new job.

As a first step in incorporating the new job, we create an `IloSchedulerSolution` object that represents this solution. This is the “default” solution. We hope that we can find a better

solution through search, however, it is possible that this search will fail due to lack of time or other computational resources. In such a case, we need a solution to fall back on.

```
IloSchedulerSolution CreateDefaultSolution(
   IloSchedulerSolution originalSolution,
   IloResourceConstraint* newJob,
   IloNumVar makespan,
   IloInt numberOfResources) {
  // The simplest way to add the new job is simply to place it at the
  // end of the existing schedule.
  IloEnv env = originalSolution.getEnv();

  IloNum newActStartMin = 0;
  IloSchedulerSolution defaultSolution = originalSolution.makeClone(env);
  for(IloInt i = 0; i < numberOfResources; ++i) {
    IloResource resource = newJob[i].getResource();

    // FIND THE ACTIVITY ON THE RESOURCE WITH THE MAX endMin
    for(IloIterator<IloResourceConstraint> iter(env);
        iter.ok(); ++iter) {
      IloResourceConstraint rct = *iter;
      if ((rct.getImpl() != newJob[i].getImpl()) &&
  (rct.getResource().getImpl() == resource.getImpl()) &&
  (defaultSolution.getEndMin(rct.getActivity()) > newActStartMin))
newActStartMin = defaultSolution.getEndMin(rct.getActivity());
    }

    IloActivity newAct = newJob[i].getActivity();
    defaultSolution.add(newAct);
    defaultSolution.setStartMin(newAct, newActStartMin);
    newActStartMin += newAct.getProcessingTimeVariable().getLB();
  }

  IloNum mkspValue = defaultSolution.getSolution().getMin(makespan);
  if (mkspValue < newActStartMin) {
    defaultSolution.getSolution().setMin(makespan, newActStartMin);
    defaultSolution.getSolution().setMax(makespan, newActStartMin);
  }

  return defaultSolution;
}
```

Creating the Temporal Model

The first submodel we create represents only the activities of the problem and the temporal constraints within each job. In particular, there is no representation of resources, resource constraints, or optimization criteria.

The temporal model is used to maintain the temporal windows of each activity given the temporal constraints within a job in the evolving model and given the evolving solution. In particular, we make the following assumptions:

1. Given that the evolving solution was found by attempting to minimize the makespan of the evolving model, we assume that the start time assigned to each activity in the evolving solution is a lower bound on the start time it can have in the next evolving solution; that is, in the solution to the next iteration of the evolving model, when a new job has been added. However, this is not necessarily the case. If we solved the evolving model from scratch, it is likely that some activities would be assigned to earlier start times than they were in the previous evolving solution. However, given that the previous evolving solution attempted to minimize makespan, it is unlikely that earlier start times will be found without a re-assignment of many start times. Since we are trying to minimize the changes from the previous evolving solution, a reasonable constraint is that the new start times can only be greater than they were in the previous iteration.
2. The upper-bound on the makespan of the new model is the one stored in the default solution.

So, with these two assumptions, we constrain the makespan of the temporal model to be less than our new upper bound. Remember that our default solution is one in which we match this new upper bound with no changes in the existing activities. Therefore, we only want to search for a solution with a smaller makespan.

```
IloNum maxMakespanIncrease = 0;
IloInt i;
for(i = 0; i < numberOfResources; ++i)
  maxMakespanIncrease +=
    newJob[i].getActivity().getProcessingTimeVariable().getUB();

temporalModel.add(makespan <
  originalSolution.getSolution().getValue(makespan) +
  maxMakespanIncrease);
```

We then add to the temporal model all the precedence constraints existing in the evolving model and constrain the minimum start time of each activity to be equal to its minimum start time in the evolving solution from the previous iteration.

Note also that we have created an `IloSchedulerSolution` object for the temporal model. This solution will be a key data structure for the communication of information between models.

Solving the temporal model (see Solving the Temporal Model for details) establishes a time window for each activity. It is important to understand that we are guaranteed that the time windows of each activity define a search space that includes at least one solution to the full evolving model, that is, to the model that includes all resources and resource constraints.

Creating the Submodels

Given the solution to the temporal model, we now create a series of submodels each of which contain one resource and the activities that require that resource within a specific time window. The process is as follows:

1. Using the temporal solution, identify a submodel for the i^{th} activity in the new job. (Initially $i = 1$.)
2. Solve the submodel. If we cannot find a solution, stop trying to incorporate this job and return the default solution.
3. Add a subset of the constraints found in solving the submodel to the temporal model.
4. Solve the temporal model.
5. Return to step 1 until all activities in the new job are scheduled.

We schedule the activities in the new job in chronological order, starting with a submodel that contains the first activity in the new job, then moving to a second submodel that contains the second activity, and so on.

Identifying a Submodel

To identify the first submodel, we examine the first activity in the new job and in particular, we use its earliest start time and latest end time as defined by the solution to the temporal model. This activity and its time window are added to the submodel.

```
IloActivity newAct = newRC.getActivity();
IloDiscreteResource reschedRes = newRC.getResource();

// GET THE TIME BOUNDS OF THE TO-BE-ADDED ACTIVITY
IloNum minTime = temporalSolution.getStartMin(newAct);
IloNum maxTime = temporalSolution.getEndMax(newAct);

env.out() << “Resource “ << reschedRes.getName()
  << “ on interval: [“ << minTime << “..” << maxTime << “]”
  << endl;

// ADD THE NEW TIME BOUNDS AND RESOURCE CONSTRAINT
subModel.add(newAct.startsAfter(minTime));
subModel.add(newAct.endsBefore(maxTime));
subModel.add(newRC);
subSolution.add(newAct);
env.out() << “Adding “ << newAct.getName() << endl;
```

We then iterate over all the resource constraints and identify those which both require the same resource as the new activity and have a time window completely within the time window of the new activity. These activities are also added to the submodel and constrained to execute between their earliest start time and latest end time as defined by the temporal model. They are also constrained to end before the variable representing the maximum end time for all activities in the submodel.

We do not add to the submodel those activities whose time window partially overlaps the time window of the new activity. Instead, we specify that the initial occupation of the resource in the submodel includes the resource capacity used by these activities. This ensures that a solution to the submodel will respect the resource reservations of those activities that are not in the submodel. This use of the initial occupation of a resource is

similar in spirit to that which is done with durable resources (see Chapter 24, *Scheduling with Durable Resources*).

It is important to understand the contents of a submodel. It contains:

1. One activity from the new job.
2. The resource required by that new activity.
3. All the other activities that execute on this resource fully within the time window fully of the new activity.
4. The resource occupation of all those activities on this resource that partially overlap the time window of the new activity.

The contents of the submodel can, perhaps, be better understood with Figure 37.1 , *Adding Activities Between EST and LFT*. The figure demonstrates that we have one activity from the new job. The earliest start time (`est`) and the latest finish time (`lft`) of the new activity are the temporal boundaries of the submodel. All existing activities whose time window is wholly within the temporal boundaries are added to the model. Those activities that overlap the temporal boundary are added as initial occupation on the resource.

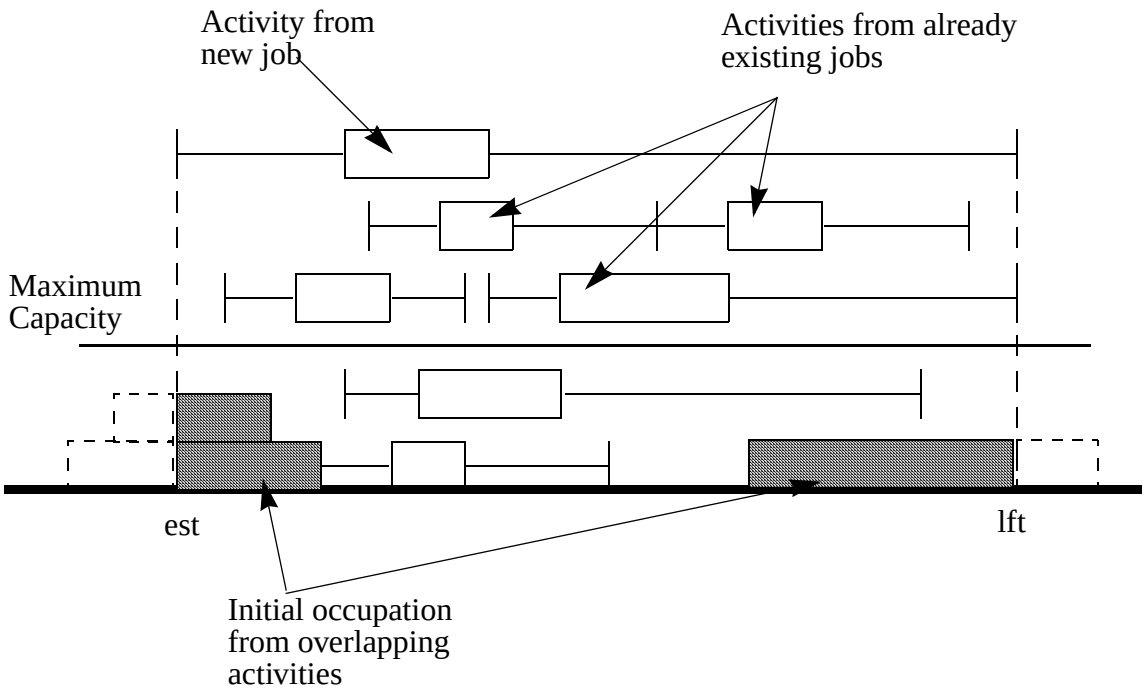


Figure 37.1 *Adding Activities Between EST and LFT*

As shown in Solving the Submodels, we limit the search that is done in solving each submodel. We are not, therefore, guaranteed that we will find a solution to each submodel. If we fail to find a solution to any submodel within the resource limits, we abandon the attempt to intelligently schedule the new job and simply incorporate it using the default solution: the activities of the new job are simply placed as the last activity on each resource. We hope, obviously, that we will not have to do this, as the default solution is not of high quality. However, the ability to respond to time limitations is critical in many applications and in many cases more critical than finding the optimal solution.

Incorporating the Submodel Solution into the Temporal Model

If the submodel is solved, we incorporate the solution into the temporal model by adding the minimum start time found for each activity in the submodel solution as the actual start time in the temporal model.

```
for(IloSchedulerSolution::ActivityIterator iter(subSolution);
    iter.ok(); ++iter) {
  IloActivity act = *iter;
  tModel.add(act.startsAt(subSolution.getStartMin(act)));
}

if (!tSolver.solve(IloGoalTrue(tSolver.getEnv())
     , IloSynchronizeAndContinue
   )
    )
  throw IloSchedulerException( “No solution for temporal model” );
IlcScheduler scheduler(tSolver);
tSolution.store(scheduler);
```

Solving the temporal model again (with the new constraints) results in a set of time windows for the non-assigned activities. We can then create the next submodel with the next activity in the new job.

This loop of creating and solving a submodel followed by adding the submodel solution to the temporal model, and then creating and solving the next submodel continues until all the activities in the new job are scheduled.

At that point, the solution to the temporal model is also a solution to the current evolving model. We therefore update the evolving solution by copying the temporal solution and return to await the arrival of the next job.

Solving the Models

The previous section presented the set of models that are used in this problem: the evolving model, the temporal model, and a number of submodels (one for each activity in each new job). In this section, we discuss how each of these models is solved using the non-deterministic search facilities of Scheduler Engine.

Solving the Original Model

The original model is simply a variation of the job shop scheduling problem with resources of capacity 3 (instead of unary resources) and with 3 instantiations of each job. The optimization criteria is the minimization of the makespan.

As with any non-deterministic search using the Scheduler Engine, there are three components to solving the problem:

1) The level of constraint propagation, defined by the enforcement levels of the model.

2) The heuristic used to guide the non-deterministic search.

3) The technique used for backtracking from a search state where we have discovered that there is no solution.

Constraint Propagation

The constraint propagation is defined by the enforcement levels of the problem model. For this problem, we have decided to set the capacity enforcement level to `IloHigh` and the precedence enforcement level to `IloMediumHigh`.

```
  IloSchedulerEnv schedEnv(env);
  schedEnv.getResourceParam().setCapacityEnforcement(IloHigh);
  schedEnv.getResourceParam().setPrecedenceEnforcement(IloMediumHigh);
```

These levels mean that for capacity enforcement, we use the disjunctive global constraint and the edge-finder, and for the precedence enforcement, we use the precedence graph global constraint. See *Resource Enforcement as Global Constraint Declaration* in the *IBM ILOG Scheduler Reference Manual* for more information.

Heuristic Search Guidance

The search goal used to solve the original model is based on the creation of texture measurements on each resource and the dynamic focusing of search interest on resources and time points that the texture measurements reveal to be critical. See *Texture Measurements* in the *IBM ILOG Scheduler Reference Manual* for more information on texture measurements.

Unlike in the texture measurement example for alternative resource scheduling (Chapter 36, *Scheduling with Alternative Resources: Interleaving Resource Allocation and Scheduling Decisions*), for this problem we generate the textures as if each activity executes at its earliest start time. This is specified in the creation of the model, specifically using the `IloTextureParam`.

```
  IloTextureParam textureParam = schedEnv.getTextureParam();
  textureParam.setCriticalityCalculator(
IloProbabilisticCriticalityCalculator(env));
  textureParam.setRCTextureFactory(IloRCTextureESTFactory(env));
  textureParam.setRandomGenerator(randGen);
```

By setting a random number generator in the texture parameter, we specify that the critical time point on a resource should be randomly selected from all those time points with maximal criticality. Without the random number generator, the critical time point is arbitrarily chosen from the time points of maximal criticality. This inserts a stochastic element into our heuristic as, given an identical configuration of activities, different critical time points will be chosen and therefore different search decisions will be made. We further

extend this stochasticity in choosing the overall critical resource (see the following code sample) and use the stochastic nature in the design of our backtracking technique.

At a high level, a single heuristic decision is made with the following code:

```
IlcGoal TextureGoalI::execute() {

  IlcResourceTexture selectedTexture = chooseResource();
  while(0 != selectedTexture.getImpl()) {

    IlcResourceConstraint rctA, rctB;
    if (!choosePrecedence(selectedTexture, rctA, rctB)) {
      selectedTexture.setNoCommitmentsAtCriticalPoint();
      selectedTexture = chooseResource();
    }
    else
      return IlcAnd(IlcTrySetSuccessor(rctA, rctB), this);
  }

  return 0;
}
```

The resource is selected by calculating the texture measurements stochastically, and choosing a critical point for each resource (if the resource's maximum criticality is greater than 0). From the set of resources with a non-zero criticality, we then choose one, with uniform probability. This adds another stochastic component to our heuristic.

```
  for(IlcResourceIterator iter(_schedule); iter.ok(); ++iter) {
    if ((*iter).isDiscreteResource()) {
      IlcDiscreteResource res = (IlcDiscreteResource) (*iter);
      IlcResourceTexture texture = res.getTextureMeasurement();

      if (texture.getMaxCriticality() > 0) {
nbNonzero++;
if ((0 == selectedTexture.getImpl()) ||
    (_randGen.getFloat() < 1/(IlcFloat) nbNonzero)) {
  selectedTexture = texture;
}
      }
    }
  }
```

Once we have selected a resource, we then select two resource constraints which compete for the critical resource and time point and which are not sequenced with each other. We then post a new precedence constraint between this pair. The heuristic for choosing this pair and the heuristically-preferred precedence constraint is based on a heuristic from the scheduling literature designed to minimize the impact of the new precedence constraint. The reverse precedence constraint is used upon backtracking.

Note that it is possible that a suitable precedence constraint on the critical resource cannot be found. This is because the texture measurement is only an estimate of the competition for the resource. In such a case the member function `setNoCommitmentsAtCriticalPoint` can

be used to inform the texture measurement that there are no commitments and therefore the critical point has been falsely identified. See `IlcResourceTexture` in the *IBM ILOG Scheduler Reference Manual* for details.

The choice point created here does not lead to a complete search. We create a binary choice point dealing with two activities, A and B. The choice point specifies: (A --> B) OR (B --> A). However, because we are dealing with discrete resources with capacity greater than 1, there is a third possibility. It may be that in an optimal solution, no precedence constraint exists: rather A and B overlap execution. Therefore, a search using this heuristic is not guaranteed to find a solution if one exists. Such a search cannot be used (by itself) to prove the optimality of a solution.

This is a design decision. The hope is that such an incomplete technique will be useful in quickly finding good solutions and that in this application quick generation of good solutions is more important than proving optimality. If there is more computation time available after a good solution has been found by an incomplete technique, we can switch to a complete search technique to prove optimality (and perhaps find even better solutions).

Backtracking

Because of the incompleteness and stochasticity of the heuristic search, it does not appear useful to perform standard chronological backtracking. Such an approach would limit the effect of stochastically on the search and, even if we completely explored the backtrack tree, we cannot guarantee that no better solutions exist. Therefore, we specify a limited number of fails to be expended in the search for a solution and use a backtracking technique that combines restart with some chronological backtracking.

Recall that because of our stochastic heuristic search, restarting the search is likely to move us to a very different area of the search space. While this diversity is often very useful, it may also be counterproductive: with a good heuristic and strong propagators, we might expect that we could find better solutions in the same area of the search space as our current best solution. Restarting immediately draws the search away from such an area. With these intuitions in mind, we use a search that combines restart with chronological backtracking. That is, we initially start chronological backtracking with a very low limit on the number of fails allows (such as 1). When the fail limit is reached, search is restarted and the fail limit is increased. In our case, we double the fail limit each time we restart up to a predefined limit on the total number of fails. This search technique attempts to balance the effort spent in searching in one area of the search space (by increasing the fail limit) with the effort spent exploring new areas of the search space (through restart). Further details about this backtracking technique can be found in the scheduling literature.

To implement our backtracking technique we make use of the `IloLimitSearch` goal in IBM ILOG Solver. Each time we find a solution, we store it in an `IloSchedulerSolution` object.

```
    IloGoal limitedGoal = IloLimitSearch(env, goal,
 IloFailLimit(env, localFailLimit));

    solver.startNewSearch(limitedGoal);

    while(!foundOpt && solver.next()) {
      bestOptVal = solver.getIntVar(optVar).getValue();
      solver.out() << “Solution with optimization value: “
   <<  bestOptVal << endl;
      solution.store(scheduler);

      if (bestOptVal == minOpt) {
solver.out() << “Locally optimal solution” << endl;
foundOpt = IloTrue;
      }
      solved = IloTrue;
    }
```

As noted above, every time we exhaust the fail limit, we double it up to the specified `failLimit`. We create a new limited goal, with the higher limit, and continue search.

```
    nbOfFails += solver.getNumberOfFails();
    localFailLimit *= 2;
    IloInt failsLeft = failLimit - nbOfFails;
    if (localFailLimit > failsLeft)
      localFailLimit = failsLeft;
```

While we are doing an incomplete search, there is a condition under which we can soundly conclude that we have an optimal solution. Before starting the search, we simply propagate all constraints and save the minimal value of the optimization criteria. This value is clearly a lower-bound on the optimization criteria. At any time during search, if we discover a solution whose optimization criteria is equal to this lower-bound, we can immediately stop the search and conclude that we have an optimal solution.

```
  solver.solve(IloGoalTrue(env));
  IlcInt minOpt = solver.getIntVar(optVar).getMin();
  solver.out() << “Initial propagation lower-bound optimization value: “
       << minOpt << endl;
```

Solving the Temporal Model

Recall that the temporal model is composed of all activities and precedence constraints in the original problem plus all activities and precedence constraints in any new jobs that have been added. It may also contain time bound constraints that restrict or even assign the start

times of some activities. It contains no resource, resource constraints, or optimization criteria. See Creating the Temporal Model for more details.

Such a model is quite easy to solve because, without any resource constraints, simple constraint propagation is a complete technique. Because, we use the temporal model for communication among the submodels and because we are only interested in finding a temporal window for each activity (to be then used in the submodel), propagation is sufficient for our needs.

```
temporalSolver = IloSolver(temporalModel);
if (!temporalSolver.solve(IloGoalTrue(env)))
  throw IloSchedulerException( “No solution for temporal model.” );
IlcScheduler temporalScheduler(temporalSolver);
temporalSolution.store(temporalScheduler);
```

Solving the Submodels

After the original problem is solved, new jobs are incorporated into the schedule by using the information provided by the temporal model to define a submodel that includes one resource, one activity from the new job, and all the activities that use that resource during the time window of the new job. See Creating the Submodels for details of the submodels.

A submodel shares some of the characteristics of the original problem. Therefore, we are able to use the same approach to search as was used to solve the original problem. The major difference in solving the submodel is the definition of the optimization criteria. Recall that we want to incorporate the new activity so as to minimize the makespan of the overall schedule, but also to minimize the changes from the original solution. This suggests that the optimization criteria for a submodel should be some combination of minimization of the makespan of the submodel (which will tend to reduce the makespan of the overall schedule) and of minimization of the number of activities whose start time is changed. Therefore, we define the optimization criteria as follows:

```
optCriteria = IloNumVar(env, 0, IloIntMax, ILOINT);
subModel.add(optCriteria == 1000 * maxEnd + IloSum(hasChanged));
subModel.add(IloMinimize(env,  optCriteria));
```

In that code, `maxEnd` is an `IloNumVar` that is greater than or equal to the end times of all the activities in the submodel and `hasChanged` is an array of variables, one for each activity (except for the activity of the new job). A variable in the `hasChanged` array will be equal to 1 only if the activity to which it corresponds is not equal to its start time in the previous temporal model. Otherwise it will be equal to 0.

Note that the makespan is multiplied by 1000 in the optimization criteria. While this indicates that the makespan is more important than the number of changes, it should be noted that the definition of the model is constrained in such a way that the makespan for a submodel will never be less than it was before the new activity was added. This is because

the activity start times in the previous iteration are used as lower bounds on the activity start times in the submodel and temporal model. In other words, even though the makespan is much more important in the optimization criteria, the submodel is constrained such that the start times of activities will not be changed simply to decrease the local makespan. Activity start times may be changed to incorporate the new activity, but they will not be changed otherwise.

The role of the texture-based heuristic should also be noted in the solving of the submodel. The texture measurements are built assuming that the activities will execute at their earliest start time. If such a start time assignment satisfies the resource constraints, then the search does nothing. Furthermore, because the earliest start times are based on an existing solution, we know that they are close to a solution. Incorporation of the new activity may therefore be as simple as assigning it to its earliest start time; all the other activities are likely to be consistent with each other. We cannot, of course, depend on this to be the case, especially in tightly constrained problems. However, in many cases the submodel can be solved to (local) optimality with no search whatsoever due to the combination of its highly constrained definition, the role of the optimization criteria, and the role of the texture based heuristics.

Problem Solving Summary

The problem presented in this chapter is to solve an initial scheduling problem and then to incorporate new jobs into the existing schedule. The constraints on this incorporation are that:

1) The makespan of the overall schedule should be as small as possible.

2) The changes from the original schedule should be as small as possible.

3) The computational resources taken to incorporate a new job are critical. We cannot attempt to solve the whole problem (original problem plus new jobs).

In order to achieve these goals, a significant effort is spent in forming a series of submodels, each of which contains only one resource from the original problem, a subset of activities, and one activity from the new job. Solving the subproblem makes use of a texture-based heuristic and an optimization criteria that tend to both minimize the makespan of the submodel and the number of changes from the original schedule. This heuristic is part of an incomplete, stochastic search whose goal is to find a good, but not optimal solution in a very short period of time.

Using a temporal model of the whole problem (which does not include any resources or resource constraints), the solution to each submodel is sequentially incorporated to ensure that a solution to the overall problem is found.

Computational Results

Even though one of the assumptions leading to the need to solve multiple submodels was that we could not solve the overall problem as one model, the example problems we use are not quite that large and it is, indeed, possible to solve them as a single model. In order to provide some comparison of the computational effort that arises from such an effort, after solving the overall problem via incrementally solving submodels described above, we then solve the entire problem.

We simply use the same technique used to solve the original model (see Solving the Original Model). In particular, it should be noted that the optimization criteria is simply the minimization of the makespan. No attention was paid to the start times assigned by the original solution. Furthermore, we specify a fail limit in solving the overall problem. Therefore, neither the incremental solutions nor the final solution found from scratch necessarily represent optimal solutions. These results are provided simply as a comparison to what was done with multiple submodels.

The smallest problem instances for this example is based on the 6X6 MT06 jobshop problem. The original problem we solve has 18 jobs of 6 activities each, to be scheduled on discrete resources of capacity 3. The final problem has 24 jobs of 6 activities each. The computation results (running on a Sun UltraSparc10[TM]) are shown in the following table.

Table 37.2

	Original Solve	Incremental Solves	Solve from scratch
Makespan	56	83	73
% Makespan Increase	n/a	48.2143%	30.3571%
% Activities Changed	n/a	2.77778% (3)	86.1111% (93)
Running Time	11.93	7.55	19.31

We see that finding a solution with the whole problem, not surprisingly, achieves a lower makespan while completely changing the schedule. The incremental solves, however, have incorporated 6 new jobs, that is 36 new activities, while only perturbing the start times of 3 of the existing activities. Also, the results here indicate each job is scheduled in a mean time of 1.26 seconds.

The second problem instance is based on the 10x10, MT10 job shop problem. The original problem has 30 jobs of 10 activities each and the final problem has 40 jobs of 10 activities each. Our computation results are as follows.

Table 37.3

	Original Solve	Incremental Solves	Solve from scratch
Makespan	1096	1792	1367
% Makespan Increase	n/a	63.5036%	24.7263%
% Activities Changed	n/a	4.66667% (14)	97.3333% (292)
Running Time	83.91	23.4	128.88

These results indicate that resolving the entire problem can substantially decrease the makespan while, again, disturbing almost all of the original activities. The incremental solves have incorporated 10 new jobs (100 activities) while perturbing the start times of only 14 activities. Furthermore, the solve time is on average 2.34 seconds for each job.

Finally, the largest problem instance we look at here is based on the 20X5, MT20 job shop benchmark. The original problem solved contains 60 jobs each with 5 activities and the final problem contains 80 jobs with 5 activities each. Our results are shown in the following table.

Table 37.4

	Original Solve	Incremental Solves	Solve from scratch
Makespan	1645	1732	1742
% Makespan Increase	n/a	5.28875%	5.89666%
% Activities Changed	n/a	32.6667% (98)	98% (294)
Running Time	204.63	183.41	308.48

Here we find that not only is the incremental solve able to find a solution with fewer changed activities than solving from scratch, but that the incremental solve finds a solution with a smaller makespan as well. Each job is incorporated with a mean solve time of 9.17 seconds.

Complete Program

You can see the entire program `resched2.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN
```

```
IloInt ResourceNumbers06 [] = {2, 0, 1, 3, 5, 4,
                               1, 2, 4, 5, 0, 3,
                               2, 3, 5, 0, 1, 4,
                               1, 0, 2, 3, 4, 5,
                               2, 1, 4, 5, 0, 3,
                               1, 3, 5, 0, 4, 2};

IloNum Durations06 [] = { 1,  3,  6,  7,  3,  6,
                          8,  5, 10, 10, 10,  4,
                          5,  4,  8,  9,  1,  7,
                          5,  5,  5,  3,  8,  9,
                          9,  3,  5,  4,  3,  1,
                          3,  3,  9, 10,  4,  1};

IloInt ResourceNumbers10 [] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
                               0, 2, 4, 9, 3, 1, 6, 5, 7, 8,
                               1, 0, 3, 2, 8, 5, 7, 6, 9, 4,
                               1, 2, 0, 4, 6, 8, 7, 3, 9, 5,
                               2, 0, 1, 5, 3, 4, 8, 7, 9, 6,
                               2, 1, 5, 3, 8, 9, 0, 6, 4, 7,
                               1, 0, 3, 2, 6, 5, 9, 8, 7, 4,
                               2, 0, 1, 5, 4, 6, 8, 9, 7, 3,
                               0, 1, 3, 5, 2, 9, 6, 7, 4, 8,
                               1, 0, 2, 6, 8, 9, 5, 3, 4, 7};

IloNum Durations10 [] = {29, 78,  9, 36, 49, 11, 62, 56, 44, 21,
                         43, 90, 75, 11, 69, 28, 46, 46, 72, 30,
                         91, 85, 39, 74, 90, 10, 12, 89, 45, 33,
                         81, 95, 71, 99,  9, 52, 85, 98, 22, 43,
                         14,  6, 22, 61, 26, 69, 21, 49, 72, 53,
                         84,  2, 52, 95, 48, 72, 47, 65,  6, 25,
                         46, 37, 61, 13, 32, 21, 32, 89, 30, 55,
                         31, 86, 46, 74, 32, 88, 19, 48, 36, 79,
                         76, 69, 76, 51, 85, 11, 40, 89, 26, 74,
                         85, 13, 61,  7, 64, 76, 47, 52, 90, 45};

IloInt ResourceNumbers20 [] = {0, 1, 2, 3, 4,
                               0, 1, 3, 2, 4,
                               1, 0, 2, 4, 3,
                               1, 0, 4, 2, 3,
                               2, 1, 0, 3, 4,
                               2, 1, 4, 0, 3,
                               1, 0, 2, 3, 4,
                               2, 1, 0, 3, 4,
                               0, 3, 2, 1, 4,
                               1, 2, 0, 3, 4,
                               1, 3, 0, 4, 2,
                               2, 0, 1, 3, 4,
                               0, 2, 1, 3, 4,
                               2, 0, 1, 3, 4,
                               0, 1, 4, 2, 3,
                               1, 0, 3, 4, 2,
                               0, 2, 1, 3, 4,
                               0, 1, 4, 2, 3,
                               1, 2, 0, 3, 4,
                               0, 1, 2, 3, 4};
```

```
IloNum Durations20 [] = {29,  9, 49, 62, 44,
                         43, 75, 69, 46, 72,
                         91, 39, 90, 12, 45,
                         81, 71,  9, 85, 22,
                         14, 22, 26, 21, 72,
                         84, 52, 48, 47,  6,
                         46, 61, 32, 32, 30,
                         31, 46, 32, 19, 36,
                         76, 76, 85, 40, 26,
                         85, 61, 64, 47, 90,
                         78, 36, 11, 56, 21,
                         90, 11, 28, 46, 30,
                         85, 74, 10, 89, 33,
                         95, 99, 52, 98, 43,
                                                 6, 61, 69, 49, 53,
                                                 2, 95, 72, 65, 25,
                         37, 13, 21, 89, 55,
                         86, 74, 88, 48, 79,
                         69, 51, 11, 89, 74,
                         13,  7, 76, 52, 45};


////////////////////////////////////////////////////////////////////////////
//
// TEXTURE CHOICE POINT FOR SCHEDULING AND RESCHEDULING
//
////////////////////////////////////////////////////////////////////////////
class TextureGoalI : public IlcGoalI {
private:
  IlcSchedule _schedule;
  IloRandom _randGen;

protected:
  IlcResourceTexture chooseResource();
  IlcBool choosePrecedence(IlcResourceTexture,
                           IlcResourceConstraint&, IlcResourceConstraint&);
public:
  TextureGoalI(IlcSchedule s, IloRandom r)
    : IlcGoalI(s.getSolver()), _schedule(s), _randGen(r) { }
  ~TextureGoalI() {}

  virtual IlcGoal execute();
};

IlcGoal TextureGoalI::execute() {

  IlcResourceTexture selectedTexture = chooseResource();
  while(0 != selectedTexture.getImpl()) {

    IlcResourceConstraint rctA, rctB;
    if (!choosePrecedence(selectedTexture, rctA, rctB)) {
      selectedTexture.setNoCommitmentsAtCriticalPoint();
      selectedTexture = chooseResource();
    }
    else
      return IlcAnd(IlcTrySetSuccessor(rctA, rctB), this);
  }
```

```
  return 0;
}


IlcResourceTexture TextureGoalI::chooseResource() {

  IlcResourceTexture selectedTexture = 0;
  IlcInt nbNonzero = 0;

  // Randomly choose a texture from all of those with non-zero max
  // criticality.
  for(IlcResourceIterator iter(_schedule); iter.ok(); ++iter) {
    if ((*iter).isDiscreteResource()) {
      IlcDiscreteResource res = (IlcDiscreteResource) (*iter);
      IlcResourceTexture texture = res.getTextureMeasurement();

      if (texture.getMaxCriticality() > 0) {
                                                                  nbNonzero++;
                                        if ((0 == selectedTexture.getImpl()) ||
                                (_randGen.getFloat() < 1/(IlcFloat) nbNonzero)) {
                                                    selectedTexture = texture;
                                                                          }
      }
    }
  }

  return selectedTexture;
}


IlcBool TextureGoalI::choosePrecedence(IlcResourceTexture texture,
                                                      IlcResourceConstraint& before,
                                                     IlcResourceConstraint& after) {

  // Find a pair of unsequenced resource constraints which minimize a
  // measure of temporal slack due to:
  //
  // Laborie, P. and Ghallab, M.
  // Planning with Sharable Resource Constraints
  // Proceedings of the 14th International Joint Conference on
  // Artificial Intelligence, IJCAI-95, 1995

  before = 0;
  after = 0;

  IlcRCTextureArray rcArray = texture.getCriticalityOrderedRCTextures();
  IlcInt nbRCs = rcArray.getSize();

  IlcFloat minCommitVal = IlcFloatMax;
  for(IlcInt i = 0; (i < nbRCs - 1) &&
                             (rcArray[i].getCriticalContribution() > 0); ++i) {
    IlcResourceConstraint ctI = (rcArray[i]).getResourceConstraint();
    IlcInt startMinI = ctI.getActivity().getStartMin();
    IlcInt startMaxI = ctI.getActivity().getStartMax();
    IlcInt endMinI = ctI.getActivity().getEndMin();
    IlcInt endMaxI = ctI.getActivity().getEndMax();

    for(IlcInt j = i+1; (j < nbRCs) &&
```

```
                              (rcArray[j].getCriticalContribution() > 0); ++j) {

      IlcResourceConstraint ctJ = (rcArray[j]).getResourceConstraint();
      if (!ctI.isSucceededBy(ctJ) && !ctJ.isSucceededBy(ctI)) {
                            IlcInt startMinJ = ctJ.getActivity().getStartMin();
                            IlcInt startMaxJ = ctJ.getActivity().getStartMax();
                                IlcInt endMinJ = ctJ.getActivity().getEndMin();
                                IlcInt endMaxJ = ctJ.getActivity().getEndMax();

                                                        // calculate I before J
                  IlcInt denom = (endMaxI - endMinI) + (startMaxJ - startMinJ);
                                                                  IlcInt numer;
                                                         IlcFloat commitVal = 1;

                                    if ((endMinI <= startMaxJ) && (denom > 0)) {
                                 numer = (IlcMin(endMaxI, startMaxJ) - endMinI) +
                                        (startMaxJ - IlcMax(startMinJ, endMinI));
                  commitVal = 1 - IlcMax(((IlcFloat)numer/(IlcFloat)denom), 0);

                                                      if (commitVal < minCommitVal) {
                                                            minCommitVal = commitVal;
                                                                        before = ctI;
                                                                         after = ctJ;
                                                                                }
                                                                                }

                                                        // calculate J before I
                         denom = (endMaxJ - endMinJ) + (startMaxI - startMinI);
                                    if ((endMinJ <= startMaxI) && (denom > 0)) {
                                 numer = (IlcMin(endMaxJ, startMaxI) - endMinJ) +
                                        (startMaxI - IlcMax(startMinI, endMinJ));
                  commitVal = 1 - IlcMax(((IlcFloat)numer/(IlcFloat)denom), 0);

                                                      if (commitVal < minCommitVal) {
                                                            minCommitVal = commitVal;
                                                                        before = ctJ;
                                                                         after = ctI;
                                                                                }
                                                                                }
      }
    }
  }

  return ((before.getImpl() != 0) && (after.getImpl() != 0));
}


ILCGOAL1(AssignESTArray,
                                                          IlcIntVarArray, stArray) {
  IloSolver solver = getSolver();
  IlcInt size = stArray.getSize();
  for(IlcInt i = 0; i < size; ++i)
    stArray[i].setValue(stArray[i].getMin());
  return 0;
}

ILOCPGOALWRAPPER2(TextureGoal, solver, IloNumVar, cost, IloRandom, rand) {
```

```
  IlcScheduler scheduler(solver);

  IlcIntVarArray varArray(solver, scheduler.getNumberOfActivities()+1);
  IlcInt index = 0;
  for(IlcActivityIterator iter(scheduler); iter.ok(); ++iter)
    varArray[index++] = (*iter).getStartVariable();

  varArray[index] = solver.getIntVar(cost);

  return IlcAnd(IlcGoal(new (solver.getHeap())
                                              TextureGoalI(scheduler, rand)),
                                          AssignESTArray(solver, varArray));
}

////////////////////////////////////////////////////////////////////////////
//
// ORIGINAL MODEL DEFINITION
//
////////////////////////////////////////////////////////////////////////////

void CreateOriginalModel(const IloEnv env,
                         IloNum resourceCapacity,
                         IloInt numberOfJobs,
                                                  IloInt numberOfResources,
                         IloInt* resourceNumbers,
                                                         IloNum* durations,
                         IloNumVar& makespan,
                         IloModel& model,
                         IloSchedulerSolution& solution)
{
  IloRandom randGen(env, 8975324);

  IloInt i, j, k;
  IloInt numberOfActivities = numberOfJobs * numberOfResources;
  char buffer[128];

  model = IloModel(env);
  solution = IloSchedulerSolution(env);

  // CREATE THE MAKESPAN VARIABLE.
  IloNum horizon = 0;
  for (i = 0; i < numberOfActivities; i++)
    horizon += durations[i];
  makespan = IloNumVar(env, 0, horizon, ILOINT);
  solution.getSolution().add(makespan);

  // CREATE THE RESOURCES.
  IloSchedulerEnv schedEnv(env);
  schedEnv.getResourceParam().setCapacityEnforcement(IloHigh);
  schedEnv.getResourceParam().setPrecedenceEnforcement(IloMediumHigh);

  IloTextureParam textureParam = schedEnv.getTextureParam();
  textureParam.setCriticalityCalculator(
                                    IloProbabilisticCriticalityCalculator(env));
  textureParam.setRCTextureFactory(IloRCTextureESTFactory(env));
  textureParam.setRandomGenerator(randGen);

  IloDiscreteResource *resources =
```

```
    new (env) IloDiscreteResource[numberOfResources];
  for (j = 0; j < numberOfResources; j++) {
    sprintf(buffer, "R%ld", j);
    resources[j] = IloDiscreteResource(env,resourceCapacity,buffer);
  }

  IloInt r;
  for(r = 0; r < resourceCapacity; ++r) {
    // CREATE THE ACTIVITIES.
    k = 0;
    for (i = 0; i < numberOfJobs; i++) {
      IloActivity previousActivity;
      for (j = 0; j < numberOfResources; j++) {
              sprintf(buffer, "O%ldJ%ldS%ldR%ld", r, i, j, resourceNumbers[k]);
                              IloActivity activity(env, durations[k], buffer);
                  IloDiscreteResource resource = resources[resourceNumbers[k]];
                                      model.add(activity.requires(resource, 1));
                                                                  if (j != 0)
                           model.add(activity.startsAfterEnd(previousActivity));

                                                               solution.add(activity);
                                                        previousActivity = activity;
                                                                          k++;
      }
      model.add( previousActivity.endsBefore(makespan) );
    }
  }

  // SET THE OBJECTIVE
  model.add(IloMinimize(env, makespan));
}

////////////////////////////////////////////////////////////////////////////
//
// SOLVE THE MODEL (ORIGINAL OR SUBMODEL)
//
////////////////////////////////////////////////////////////////////////////

IloBool SolveModel(const IloModel model, IloSchedulerSolution solution,
                                                          const IloNumVar optVar,
                                       IloInt failLimit, IloNum& runningTime) {

  IloEnv env = model.getEnv();

  IloSolver solver(model);
  IlcScheduler scheduler(solver);
  IloBool solved = IloFalse;

  // PROPAGATE TO FIND LOWER BOUND ON OPT-VAR
  solver.solve(IloGoalTrue(env));
  IlcInt minOpt = solver.getIntVar(optVar).getMin();
  solver.out() << "Initial propagation lower-bound optimization value: "
                                                                << minOpt << endl;

  IloGoal goal = TextureGoal(env, optVar, IloRandom(env, 45874052));

  IloInt nbOfFails = 0;
  runningTime = 0;
```

```
  IloBool foundOpt = IloFalse;
  IloInt localFailLimit = 1;
  IloNum bestOptVal = optVar.getUB();
  IloNum lastBestOptVal = bestOptVal;
  IloConstraint optVarConstraint;
  while(!foundOpt && (localFailLimit > 0)) {

    IloGoal limitedGoal = IloLimitSearch(env, goal,
                                         IloFailLimit(env, localFailLimit));

    solver.startNewSearch(limitedGoal);

    while(!foundOpt && solver.next()) {
      bestOptVal = solver.getIntVar(optVar).getValue();
      solver.out() << "Solution with optimization value: "
                                                    <<  bestOptVal << endl;
      solution.store(scheduler);

      if (bestOptVal == minOpt) {
                          solver.out() << "Locally optimal solution" << endl;
                                                        foundOpt = IloTrue;
      }
      solved = IloTrue;
    }

    runningTime += solver.getTime();
    nbOfFails += solver.getNumberOfFails();
    localFailLimit *= 2;
    IloInt failsLeft = failLimit - nbOfFails;
    if (localFailLimit > failsLeft)
      localFailLimit = failsLeft;

    solver.endSearch();

    if (bestOptVal < lastBestOptVal) {
      if (optVarConstraint.getImpl() != 0)
                                             model.remove(optVarConstraint);

      optVarConstraint = (optVar < bestOptVal);
      model.add(optVarConstraint);
      lastBestOptVal = bestOptVal;
    }
  }

  solver.out() << "Number of fails         : " << nbOfFails
                               << "\nRunning time            : " << runningTime
                                                                  << endl;
  solver.end();

  if (optVarConstraint.getImpl() != 0)
    model.remove(optVarConstraint);

  return solved;
}

////////////////////////////////////////////////////////////////////////////
//
// RESCHEDULING EVENT (ADD A NEW JOB)
```

```
//
////////////////////////////////////////////////////////////////////////////

void AddNewJob(IloModel model,
                                                        IloInt newJobNumber,
                                                IloResourceConstraint *rcArray,
                                                         IloNumVar makespan,
                                                        IloInt numberOfJobs,
                                                   IloInt numberOfResources,
                                               IloNum *existingDurations) {

  IloEnv env = model.getEnv();
  IloRandom randGen(env, 98303001);

  // CREATE DURATIONS
  IloNum minDur = IloInfinity;
  IloNum maxDur = -IloInfinity;
  IloInt i;
  for(i = 0; i < numberOfJobs * numberOfResources; ++i) {
    if (existingDurations[i] < minDur)
      minDur = existingDurations[i];
    if (existingDurations[i] > maxDur)
      maxDur = existingDurations[i];
  }

  IloNum diff = maxDur - minDur + 1;
  IloNumArray durations(env, numberOfResources);
  for(i = 0; i < numberOfResources; ++i)
    durations[i] = randGen.getInt(IloTrunc(diff)) + minDur;

  // CREATE RESOURCE USAGE
  IloArray<IloDiscreteResource> resources(env, numberOfResources);
  i = 0;
  for(IloIterator<IloDiscreteResource> iter(env); iter.ok(); ++iter) {
    resources[i] = *iter;
    ++i;
  }

  // RANDOMLY PERMUTE RESOURCE REQUIREMENTS
  for(i = 0; i < 1000; ++i) {
    IloInt index1 = randGen.getInt(numberOfResources);
    IloInt index2 = randGen.getInt(numberOfResources);
    while (index1 == index2)
      index2 = randGen.getInt(numberOfResources);
    IloDiscreteResource tmp = resources[index1];
    resources[index1] = resources[index2];
    resources[index2] = tmp;
  }

  // CREATE THE ACTIVITIES
  IloActivity previousActivity;
  char buffer[128];
  for(i = 0; i < numberOfResources; ++i) {
    sprintf(buffer, "NJ%ldS%ld%s", newJobNumber, i, resources[i].getName());
    IloActivity activity(env, durations[i], buffer);
    rcArray[i] = activity.requires(resources[i], 1);
    model.add(rcArray[i]);
    if (i != 0)
```

```
      model.add(activity.startsAfterEnd(previousActivity));
    previousActivity = activity;
  }

  model.add(previousActivity.endsBefore(makespan));
  randGen.end();
}


////////////////////////////////////////////////////////////////////////////
//
// DEFINE AND SOLVE TEMPORAL MODEL
//
////////////////////////////////////////////////////////////////////////////
void
CreateTemporalModel(IloSchedulerSolution originalSolution,
                                                  IloResourceConstraint *newJob,
                                                       IloInt numberOfResources,
                                                             IloNumVar makespan,
                                                        IloModel& temporalModel,
                                          IloSchedulerSolution& temporalSolution) {

  IloEnv env = originalSolution.getEnv();
  temporalModel = IloModel(env);
  temporalSolution = IloSchedulerSolution(env);
  temporalSolution.add(makespan);

  // CALCULATE MAXIMUM INCREASE IN MAKESPAN FROM THE NEW JOB
  IloNum maxMakespanIncrease = 0;
  IloInt i;
  for(i = 0; i < numberOfResources; ++i)
    maxMakespanIncrease +=
      newJob[i].getActivity().getProcessingTimeVariable().getUB();

  temporalModel.add(makespan <
                            originalSolution.getSolution().getValue(makespan) +
                                                              maxMakespanIncrease);

  // Put all the precedence constraints from the original model into
  // the temporal model but not the resource constraints.
  for (IloIterator<IloPrecedenceConstraint> pcIter(env);
       pcIter.ok();
       ++pcIter)
    {
      IloPrecedenceConstraint pc = (*pcIter);
      IloActivity nextAct = pc.getFollowingActivity();
      IloActivity prevAct = pc.getPrecedingActivity();

      temporalModel.add(pc);
      if (!temporalSolution.contains(nextAct))
                                                      temporalSolution.add(nextAct);
      if (!temporalSolution.contains(prevAct))
                                                      temporalSolution.add(prevAct);
      if (originalSolution.contains(prevAct))
                                         temporalModel.add(prevAct.startsAfter(
                                       originalSolution.getStartMin(prevAct)));
      if (originalSolution.contains(nextAct))
                                         temporalModel.add(nextAct.startsAfter(
```

```
                                        originalSolution.getStartMin(nextAct)));
    }

  // search for the last activity, and add
  // its time-bound constraint over the makespan.
  for (IloIterator<IloTimeBoundConstraint> tbcIter(env);
       tbcIter.ok();
       ++tbcIter) {
    IloTimeBoundConstraint tbc = (*tbcIter);
    if (tbc.getType() == IloEndsBefore)
      temporalModel.add(tbc);
  }
}


void CreateAndSolveTemporalModel(IloSchedulerSolution originalSolution,
                                                 IloResourceConstraint *newJob,
                                                      IloInt numberOfResources,
                                                            IloNumVar makespan,
                                                    IloSolver& temporalSolver,
                                                      IloModel& temporalModel,
                                           IloSchedulerSolution& temporalSolution) {


  CreateTemporalModel(originalSolution,
                                                                        newJob,
                                                                numberOfResources,
                                                                        makespan,
                                                            temporalModel,
                                                                temporalSolution);

  IloEnv env = originalSolution.getEnv();

  // SOLVE TEMPORAL MODEL (PROPAGATE)
  temporalSolver = IloSolver(temporalModel);
  if (!temporalSolver.solve(IloGoalTrue(env)))
    throw IloSchedulerException( "No solution for temporal model." );
  IlcScheduler temporalScheduler(temporalSolver);
  temporalSolution.store(temporalScheduler);

  IloInt i;
  for(i = 0; i < numberOfResources; ++i) {
    IloActivity act = newJob[i].getActivity();
    env.out() << "\t" << act.getName() << "["
                                          << temporalSolution.getStartMin(act)
                                    << ".." << temporalSolution.getStartMax(act)
                              << " -- " << act.getProcessingTimeVariable().getLB()
                                    << " --> " << temporalSolution.getEndMin(act)
                                       << ".." << temporalSolution.getEndMax(act)
                                                             << "]" << endl;
  }
  env.out() << "Maximum new makespan: "
                    << temporalSolution.getSolution().getMax(makespan) << endl;
}

////////////////////////////////////////////////////////////////////////////
//
// UPDATE TEMPORAL MODEL
```

```
//
////////////////////////////////////////////////////////////////////////////
void UpdateTemporalModel(IloModel tModel,
                                                        IloSolver tSolver,
                                                IloSchedulerSolution tSolution,
                                              IloSchedulerSolution subSolution) {

  // ADD THE START TIMES IN tSolution TO THE TEMPORAL MODEL
  for(IloSchedulerSolution::ActivityIterator iter(subSolution);
      iter.ok(); ++iter) {
    IloActivity act = *iter;
    tModel.add(act.startsAt(subSolution.getStartMin(act)));
  }

  if (!tSolver.solve(IloGoalTrue(tSolver.getEnv())
                                                     , IloSynchronizeAndContinue
                                                                          )
      )
    throw IloSchedulerException( "No solution for temporal model" );
  IlcScheduler scheduler(tSolver);
  tSolution.store(scheduler);
}

////////////////////////////////////////////////////////////////////////////
//
// CREATE THE DEFAULT SOLUTION
//
////////////////////////////////////////////////////////////////////////////
IloSchedulerSolution CreateDefaultSolution(
                                         IloSchedulerSolution originalSolution,
                                                 IloResourceConstraint* newJob,
                                                         IloNumVar makespan,
                                                    IloInt numberOfResources) {
  // The simplest way to add the new job is simply to place it at the
  // end of the existing schedule.
  IloEnv env = originalSolution.getEnv();

  IloNum newActStartMin = 0;
  IloSchedulerSolution defaultSolution = originalSolution.makeClone(env);
  for(IloInt i = 0; i < numberOfResources; ++i) {
    IloResource resource = newJob[i].getResource();

    // FIND THE ACTIVITY ON THE RESOURCE WITH THE MAX endMin
    for(IloIterator<IloResourceConstraint> iter(env);
        iter.ok(); ++iter) {
      IloResourceConstraint rct = *iter;
      if ((rct.getImpl() != newJob[i].getImpl()) &&
                          (rct.getResource().getImpl() == resource.getImpl()) &&
                (defaultSolution.getEndMin(rct.getActivity()) > newActStartMin))
                  newActStartMin = defaultSolution.getEndMin(rct.getActivity());
    }

    IloActivity newAct = newJob[i].getActivity();
    defaultSolution.add(newAct);
    defaultSolution.setStartMin(newAct, newActStartMin);
    newActStartMin += newAct.getProcessingTimeVariable().getLB();
  }
```

```
  IloNum mkspValue = defaultSolution.getSolution().getMin(makespan);
  if (mkspValue < newActStartMin) {
    defaultSolution.getSolution().setMin(makespan, newActStartMin);
    defaultSolution.getSolution().setMax(makespan, newActStartMin);
  }

  return defaultSolution;
}



////////////////////////////////////////////////////////////////////////////
//
// SUB-MODEL DEFINITION
//
////////////////////////////////////////////////////////////////////////////

void
CreateSubmodel(IloSchedulerSolution temporalSolution,
                                                          IloModel& subModel,
                                              IloSchedulerSolution& subSolution,
                                                        IloNumVar& optCriteria,
                                           IloSchedulerSolution originalSolution,
                                                   IloResourceConstraint newRC) {

  IloEnv env = temporalSolution.getEnv();
  subModel = IloModel(env);
  subSolution = IloSchedulerSolution(env);

  IloActivity newAct = newRC.getActivity();
  IloDiscreteResource reschedRes = newRC.getResource();

  // GET THE TIME BOUNDS OF THE TO-BE-ADDED ACTIVITY
  IloNum minTime = temporalSolution.getStartMin(newAct);
  IloNum maxTime = temporalSolution.getEndMax(newAct);

  env.out() << "Resource " << reschedRes.getName()
                      << " on interval: [" << minTime << ".." << maxTime << "]"
                                                                    << endl;

  // ADD THE NEW TIME BOUNDS AND RESOURCE CONSTRAINT
  subModel.add(newAct.startsAfter(minTime));
  subModel.add(newAct.endsBefore(maxTime));
  subModel.add(newRC);
  subSolution.add(newAct);
  env.out() << "Adding " << newAct.getName() << endl;

  IloNumVar maxEnd(env, minTime, maxTime, ILOINT);
  subSolution.add(maxEnd);
  subModel.add(newAct.endsBefore(maxEnd));

  IloSchedulerEnv schedEnv(env);

  IloNumToNumStepFunction initialOcc = schedEnv.getInitialOccupationParam();
  initialOcc.setValue(initialOcc.getDefinitionIntervalMin(),
                                    initialOcc.getDefinitionIntervalMax(), 0);

  IloInt nbActsInSubmodel = 0;
```

```
  IloNumVarArray hasChanged(env);

  // ITERATE OVER THE RESOURCE CONSTRAINTS
  for(IloIterator<IloResourceConstraint> iter(env); iter.ok(); ++iter) {
    IloDiscreteResource res = (*iter).getResource();
    IloActivity act = (*iter).getActivity();

    if ((res.getImpl() == reschedRes.getImpl()) &&
                                        (act.getImpl() != newAct.getImpl())) {
      IloNum actStart = temporalSolution.getStartMin(act);
      IloNum actEnd = temporalSolution.getEndMin(act);

      if ((actStart >= minTime) && (actEnd <= maxTime)) {
                    // ADD TO MODEL THOSE ACTIVITIES HAVE A [MIN-START MIN-END]
                     // INTERVAL WHOLLY WITHIN THE INTERVAL OF THE NEW ACTIVITY
                                                    subModel.add((*iter));
                                          subModel.add(act.startsAfter(actStart));
                 subModel.add(act.endsBefore(temporalSolution.getEndMax(act)));

                                           subModel.add(act.endsBefore(maxEnd));
                                                 subSolution.add(act);

                                                   ++nbActsInSubmodel;

hasChanged.add(IloNumVar(act.startsAfter(originalSolution.getStartMin(act)+1)))
;
      }
      else if ((actEnd > minTime) && (actStart < maxTime)) {
                   // FOR THE ACTIVITIES PARTIALLY IN THE INTERVAL, ADD THEM AS
                                           // INITIAL OCCUPATION ON THE RESOURCE
                                           actStart = IloMax(actStart, minTime);
                                               actEnd = IloMin(actEnd, maxTime);
                  initialOcc.addValue(actStart, actEnd, (*iter).getCapacity());
      }
    }
  }

  // SET OPTIMIZATION CRITERIA TO BE A COMBINATION OF THE (LOCAL)
  // MAKESPAN AND THE CHANGES FROM THE ORIGINAL MODEL
  optCriteria = IloNumVar(env, 0, IloIntMax, ILOINT);
  subModel.add(optCriteria == 1000 * maxEnd + IloSum(hasChanged));
  subModel.add(IloMinimize(env,  optCriteria));
}

IloBool CreateAndSolveSubmodel(IloSchedulerSolution temporalSolution,
                                            IloSchedulerSolution& subSolution,
                                        IloSchedulerSolution originalSolution,
                                                  IloResourceConstraint newRC,
                                                          IloNum& runningTime)
{
  IloNumVar optCriteria;
  IloModel subModel;
  CreateSubmodel(temporalSolution, subModel, subSolution,
                                        optCriteria, originalSolution, newRC);

  IloInt failLimit = 1000;
  IloBool solved = SolveModel(subModel, subSolution, optCriteria,
                                                     failLimit, runningTime);
```

```
  // UNSET THE INITIAL OCCUPATION (MAY HAVE BEEN MODIFIED IN SUBMODEL)
  IloSchedulerEnv schedEnv(temporalSolution.getEnv());
  IloNumToNumStepFunction initialOcc = schedEnv.getInitialOccupationParam();
  initialOcc.setValue(initialOcc.getDefinitionIntervalMin(),
                                    initialOcc.getDefinitionIntervalMax(), 0);

  //subModel.end();
  return solved;
}



////////////////////////////////////////////////////////////////////////////
//
// COUNT CHANGES IN START TIME ASSIGNMENTS
//
////////////////////////////////////////////////////////////////////////////
IloInt CountChanges(IloSchedulerSolution originalSolution,
                                       IloSchedulerSolution currentSolution) {

  IloInt nbChanges = 0;
  for(IloSchedulerSolution::ActivityIterator iter(currentSolution);
      iter.ok(); ++iter) {
    IloActivity act = *iter;
    if (originalSolution.contains(act) &&
                                          (originalSolution.getStartMin(act) !=
                                             currentSolution.getStartMin(act)))
      ++nbChanges;
  }

  return nbChanges;
}

////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////////////////

void PrintRange(IloEnv& env, IloNum min, IloNum max) {
  if (min == max)
    env.out() << (IlcInt)min;
  else
    env.out() << (IlcInt)min << ".." << (IlcInt)max;
}

void PrintSolution(const IloSchedulerSolution solution,
                   const IloNumVar makespan)
{
  IloEnv env = solution.getEnv();
  env.out() << "Solution with makespan ["
            << solution.getSolution().getMin(makespan) << ".."
            << solution.getSolution().getMax(makespan) << "]" << endl;

  for (IloSchedulerSolution::ActivityIterator iter(solution);
       iter.ok();
       ++iter)
```

```
    {
      IloActivity activity = *iter;
      env.out() << activity.getName() << "[";
      PrintRange(env,
                                              solution.getStartMin(activity),
                                             solution.getStartMax(activity));
      env.out() << " -- ";
      PrintRange(env,
                                           solution.getDurationMin(activity),
                                          solution.getDurationMax(activity));
      env.out() << " --> ";
      PrintRange(env,
                                                solution.getEndMin(activity),
                                               solution.getEndMax(activity));
      env.out() << "]" << endl;
    }
}

void PrintSolutionComparison(IloSchedulerSolution currentSolution,
                                        IloSchedulerSolution previousSolution,
                                        IloSchedulerSolution originalSolution,
                                                    IloNumVar makespan) {

  IloInt incrementalChanges = CountChanges(previousSolution,
                                                          currentSolution);
  IloInt totalChanges = CountChanges(originalSolution,
                                                          currentSolution);
  IloNum incrementalPercentIncrease =
    (IloNum(currentSolution.getSolution().getValue(makespan)) -
     IloNum(previousSolution.getSolution().getValue(makespan))) /
    IloNum(previousSolution.getSolution().getValue(makespan)) * 100;

  IloNum totalPercentIncrease =
    (IloNum(currentSolution.getSolution().getValue(makespan)) -
     IloNum(originalSolution.getSolution().getValue(makespan))) /
    IloNum(originalSolution.getSolution().getValue(makespan)) * 100;

  IloEnv env = currentSolution.getEnv();
  env.out() << "\n\nNew job successfully added.\n\tTotal makespan: "
                             << currentSolution.getSolution().getValue(makespan)
                                   << "\n\t% makespan increase (incremental) : "
                                            << incrementalPercentIncrease << "%"
                                   << "\n\t% makespan increase (overall)     : "
                                                    << totalPercentIncrease << "%"
                                   << "\n\tNumber of changes (incremental)   : "
                                                          << incrementalChanges
                                   << "\n\tNumber of changes (overall)       : "
                                                 << totalChanges << "\n" << endl;
}

void PrintFinalSolutionComparison(IloSchedulerSolution fromScratchSolution,
                                                      IloNum fromScratchSolveTime,
                                                IloSchedulerSolution incSolution,
                                                        IloNum incSolveTime,
                                               IloSchedulerSolution origSolution,
                                                        IloNum origSolveTime,
                                                         IloNumVar makespan,
                                                        IloNum nbOrigActs) {
```

```
  // PRINT COMPARISON OF THE SOLVE-FROM-SCRATCH VS INCREMENTAL SOLVES
  IloInt incChanges = CountChanges(origSolution,
                                                          incSolution);
  IloInt scratchChanges = CountChanges(origSolution,
                                                  fromScratchSolution);

  IloNum incPercentIncrease =
    (IloNum(incSolution.getSolution().getValue(makespan)) -
     IloNum(origSolution.getSolution().getValue(makespan))) /
    IloNum(origSolution.getSolution().getValue(makespan)) * 100;

  IloNum scratchPercentIncrease =
    (IloNum(fromScratchSolution.getSolution().getValue(makespan)) -
     IloNum(origSolution.getSolution().getValue(makespan))) /
    IloNum(origSolution.getSolution().getValue(makespan)) * 100;

  IloEnv env = origSolution.getEnv();
  env.out() << "\nOriginal solve:"
                                          << "\nMakespan            : "
                               << origSolution.getSolution().getValue(makespan)
                                << "\nRunning time         : " << origSolveTime
                                             << "\n\nIncremental solves:"
                                          << "\nMakespan            : "
                                << incSolution.getSolution().getValue(makespan)
                     << "\n% Makespan increase  : " << incPercentIncrease << "%"
                                          << "\n% Activities changed : "
                               << IloNum(incChanges) / nbOrigActs * 100 << "% ("
                                                 << incChanges << ")"
                                  << "\nRunning time         : " << incSolveTime
                                             << "\n\nSolve from scratch:"
                                          << "\nMakespan            : "
                           << fromScratchSolution.getSolution().getValue(makespan)
                 << "\n% Makespan increase  : " << scratchPercentIncrease << "%"
                                          << "\n% Activities changed : "
                             << IloNum(scratchChanges) / nbOrigActs * 100 << "% ("
                                               << scratchChanges << ")"
                           << "\nRunning time       : " << fromScratchSolveTime
                                                       << "\n" << endl;

}

////////////////////////////////////////////////////////////////////////////
//
// SOLVE MODEL AFTER ADDING NEW JOB
//
////////////////////////////////////////////////////////////////////////////

void ScheduleJob(IloModel evolvingModel,
                                      IloSchedulerSolution originalSolution,
                                    IloSchedulerSolution& evolvingSolution,
                                            IloResourceConstraint* newJob,
                                                      IloNumVar makespan,
                                                 IloInt numberOfResources,
                                                 IloNum& incRunningTime) {

  IloEnv env = evolvingModel.getEnv();
```

```
// CREATE DEFAULT SOLUTION IN CASE WE FAIL TO INCORPORATE NEW JOB
IloSchedulerSolution defaultSolution =
  CreateDefaultSolution(evolvingSolution, newJob,
                                              makespan, numberOfResources);

// CREATE AND PROPAGATE TEMPORAL MODEL INCLUDING NEW JOB
IloModel temporalModel;
IloSchedulerSolution temporalSolution;
IloSolver temporalSolver;
CreateAndSolveTemporalModel(evolvingSolution, newJob,
                                                numberOfResources, makespan,
                                             temporalSolver, temporalModel,
                                                        temporalSolution);

// SCHEDULE EACH ACTIVITY IN THE NEW JOB
IloInt i;
IloBool solved = IloTrue;
for(i = 0; (i < numberOfResources) && solved; ++i) {

  env.out() << "\n**** Submodel: " << i << " ****" << endl;

  IloSchedulerSolution subSolution;
  IloNum subRunningTime;
  solved = CreateAndSolveSubmodel(temporalSolution,
                                               subSolution, evolvingSolution,
                                                  newJob[i], subRunningTime);
  incRunningTime += subRunningTime;

  if (solved) {
    // UPDATE TEMPORAL MODEL WITH SUBMODEL SOLUTION
    UpdateTemporalModel(temporalModel, temporalSolver,
                                              temporalSolution, subSolution);

    IloInt nbChanges = CountChanges(evolvingSolution,
                                                                  subSolution);

    env.out() << "Subproblem solved.\n\tTotal makespan: "
                                        << temporalSolution.getMin(makespan)
                                 << "\n\tNumber of changes in sub-solution: "
                                                     << nbChanges << endl;
  }
  else {
    env.out() << "Failure to solve subproblem!  "
                                << "Reverting to default solution."  << endl;
  }

  subSolution.end();
}

if (solved) {
  // ASSIGN THE MAKESPAN TO MIN VALUE
  temporalSolution.getSolution().setMax(makespan,
                            temporalSolution.getSolution().getMin(makespan));

  PrintSolutionComparison(temporalSolution, evolvingSolution,
                                                originalSolution, makespan);

  evolvingSolution.end();
```

```
    evolvingSolution = temporalSolution.makeClone(env);
  }
  else {
    // ONE OF THE SUBMODELS COULD NOT BE SOLVED WITHIN THE FAILURE
    // LIMITS.  FALL BACK TO THE DEFAULT SOLUTION
    PrintSolutionComparison(defaultSolution, evolvingSolution,
                                                    originalSolution, makespan);

    evolvingSolution.end();
    evolvingSolution = defaultSolution.makeClone(env);
  }

  temporalSolution.end();
  temporalSolver.end();
  defaultSolution.end();
}


////////////////////////////////////////////////////////////////////////////
//
// PARAMETERS
//
////////////////////////////////////////////////////////////////////////////

void InitParameters(int argc,
                    char** argv,
                                                          IloNum& resourceCap,
                    IloInt& numberOfJobs,
                    IloInt& numberOfResources,
                    IloInt*& resourceNumbers,
                    IloNum*& durations)
{
  resourceCap = 3;
  numberOfJobs = 6;
  numberOfResources = 6;
  resourceNumbers = ResourceNumbers06;
  durations = Durations06;

  if (argc > 1) {
    IloInt number = atol(argv[1]);
    if (number == 10) {
      numberOfJobs = 10;
      numberOfResources = 10;
      resourceNumbers = ResourceNumbers10;
      durations = Durations10;
    }
    else if (number == 20) {
      numberOfJobs = 20;
      numberOfResources = 5;
      resourceNumbers = ResourceNumbers20;
      durations = Durations20;
    }
  }

  if (argc > 2)
    resourceCap = atol(argv[2]);
}
```

```
////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////

int main(int argc, char** argv)
{
  IloNum resourceCapacity;
  IloInt numberOfJobs;
  IloInt numberOfResources;
  IloInt* resourceNumbers;
  IloNum* durations;

  InitParameters(argc,
                 argv,
                                                          resourceCapacity,
                 numberOfJobs,
                 numberOfResources,
                 resourceNumbers,
                 durations);

  try {

    IloEnv env;
    IloNumVar makespan;

    env.out() << "\n**** Original Model ****" << endl;
    IloSchedulerSolution originalSolution;
    IloModel originalModel;

    // DEFINE ORIGINAL MODEL
    CreateOriginalModel(env, resourceCapacity,
                        numberOfJobs, numberOfResources,
                        resourceNumbers, durations, makespan,
                                          originalModel, originalSolution);

    // SOLVE ORIGINAL MODEL
    IloInt failLimit = 5000;
    IloNum originalRunningTime;
    IloBool solved = SolveModel(originalModel, originalSolution,
                                                                 makespan,
                                           failLimit, originalRunningTime);

    if (!solved)
      throw IloSchedulerException( "No solution found to original model." );

    PrintSolution(originalSolution, makespan);

    IloNum incRunningTime = 0;
    IloSchedulerSolution evolvingSolution = originalSolution.makeClone(env);

    // ADD NEW JOBS, EVERY TIME A JOB IS ADDED, INCREMENTALLY RE-SOLVE
    IloInt nbNewJobs = numberOfJobs;
    IloResourceConstraint *newJob =
      new (env) IloResourceConstraint[numberOfResources];

    for(IloInt j = 0; j < nbNewJobs; ++j) {
```

```
      env.out() << "\nAdding new job: NJ" << j << endl;

      // CREATE NEW JOB
      AddNewJob(originalModel, j, newJob, makespan,
                                numberOfJobs, numberOfResources, durations);

      // INCORPORATE NEW JOB INTO SCHEDULE
      ScheduleJob(originalModel, originalSolution, evolvingSolution,
                                          newJob, makespan, numberOfResources,
                                                              incRunningTime);
    }

    // SOLVE COMPLETE MODEL (ORIGINAL + NEW JOBS) FROM SCRATCH
    env.out() << "\n **** Complete Model ***" << endl;

    IloSchedulerSolution totalSolution = evolvingSolution.makeClone(env);
    IloNum totalRunningTime;
    solved = SolveModel(originalModel, totalSolution,
                                                                    makespan,
                                                  failLimit, totalRunningTime);

    if (!solved)
      throw IloSchedulerException( "No solution found for overall model." );

    // COMPARE ORIGINAL SOLUTION TO INCREMENTAL AND FROM SCRATCH SOLUTIONS
    IloNum nbOrigActs = numberOfJobs * numberOfResources * resourceCapacity;
    PrintFinalSolutionComparison(totalSolution, totalRunningTime,
                                            evolvingSolution, incRunningTime,
                                       originalSolution, originalRunningTime,
                                                 makespan, nbOrigActs);

    PrintSolution(totalSolution, makespan);

    totalSolution.end();
    evolvingSolution.end();
    originalSolution.end();
    env.end();
  } catch (IloSchedulerException& exc) {
    cout << exc << endl;
  } catch (IloException& exc) {
    cout << exc << endl;
  }

  return 0;
}
```

C H A P T E R

38

Writing Multi-Threaded Applications Using Durable Resources

This chapter shows you how to use *durable resources* in a multi-threaded application. Note that Scheduler is multi-thread safe if each thread uses a different environment for creating scheduler objects and those objects are not accessed across different threads.

Durable resources may be concurrently used by different threads through the methods `lock` and `unlock` of the class `IlcSchedule`. Locking a set of resources may block a thread while it waits for all resources in the set to be unlocked by other threads. Unlocking of resources allows other threads to gain access to these resources.

Describing the Problem

Let us consider again the problem described in Chapter 22, *Using Strong Propagation on Reservoirs: the Balance Constraint*. This time, rather than handling all calls by a single person, there are several operators dealing with customer requests.

When a new request is received, the customer is dispatched to an available operator. The operator engages in the same procedure as in the previous example, either proposing a delivery date or declining to deliver the product by the next month. If the customer accepts the proposed date, it becomes a due date on which the enterprise commits to deliver the ordered product.

Designing a Model

As in Chapter 22, the workers are modeled as unary durable resources with breaklists and with the corresponding timetable constraints added. The fact that there are now several operators simultaneously handling requests translates into running several threads, each one corresponding to an operator and each one disposing of its own environment (object of type `IloEnv`) and of its own solver (object of type `IloSolver`). These threads are created by constructing an object of type `IlcWorkServer`. This object allows you to dispatch the requests among threads transparently.

Solving The Problem

There are a few differences in the code when compared to Chapter 22. One difference is that closing the durable schedule now becomes mandatory. Closing a durable schedule ensures that no further durable resources will be created. The durable schedule can be closed at the end of the function `CreateDurableResources`.

```
IlcScheduler CreateDurableResources(IloSolver solver0,
                                    IloInt nbOfWorkers,
                                    IloUnaryResource* workers) {
  /* CREATE A DURABLE MODEL. */
  IloEnv env0 = solver0.getEnv();
  IloModel model0(env0);
  IloSchedulerEnv schedEnv0(env0);

  IloIntervalList blist(env0, 0, 100);
  blist.addPeriodicInterval(5, 2, 7, 100);
  schedEnv0.setBreakListParam(blist);

  IlcInt i=0;
  for (i = 0; i < nbOfWorkers; i++){
    workers[i] = IloUnaryResource(env0);
    model0.add(workers[i]);
  }

  IlcScheduler durSched(solver0);
  durSched.setDurable();
  solver0.extract(model0);

  /* CLOSE THE DURABLE SCHEDULER. */
  durSched.close();

  return durSched;
}
```

The treatment of a request by an operator remains made by the goal `TreatRequest`.

```
ILCGOAL3(TreatRequestIlc,
         IlcInt, reqIndex,
         IlcUnaryResource*, team,
         IlcInt, releaseDate){
```

A given number of threads are created by constructing an object of type `IlcWorkServer`. The first thread prints the output in the file called `output0.out`, the second in the file `output1.out`, etc.

```
/* CREATING THE WORKER THREADS. */
IloInt nbOfThreads = 3;
if (argc >= 2)
  nbOfThreads = atol(argv[1]);
IlcWorkServer server(sched0, nbOfThreads, “output”);
```

The main loop of the application reads requests one by one from the input stream of requests and dispatches them to be treated on different threads. At each iteration, you must get the environment of an available thread, construct the goal `TreatRequest` for that environment, and launch the request treatment. The goal is then extracted to the solver associated to the corresponding thread and its code is executed asynchronously on the thread.

In this case, you don't have to delete the environment of the used solver, as its memory is managed by the `IlcWorkServer` object.

```
/* TREATING THE REQUESTS. */
for (IloInt j = 0; j < HowManyReqs; j++) {
  // Reading a request :
  IloInt requestType = StreamOfReqs[j][0];
  IloInt releaseDate = StreamOfReqs[j][1];

  // Getting a computation environment :
  IloEnv env = server.getIdleEnv();
  // Solving the specific problem associated with the request:
  IloUnaryResource* team = (requestType == 1 ?
                            firstTeam :
                            secondTeam);
  server.launch(TreatRequest(env, sched0, j, team, releaseDate));
}
```

After all requests are processed, the threads are stopped by deleting the `IlcWorkServer` object.

```
/* STOPPING THE WORKER THREADS. */
server.end();
```

Complete Program and Output

You can see the entire program `durablem.cpp` here or view it online in the standard distribution.

```
#include <ilsched/iloscheduler.h>
#include <ilsched/workserv.h>

ILOSTLBEGIN

const IloInt Horizon = 30;

////////////////////////////////////////////
//
// CREATING THE DURABLE RESOURCES
//
////////////////////////////////////////////

IlcScheduler CreateDurableResources(IloSolver solver0,
                                    IloInt nbOfWorkers,
                                    IloUnaryResource* workers) {
  /* CREATE A DURABLE MODEL. */
  IloEnv env0 = solver0.getEnv();
  IloModel model0(env0);
  IloSchedulerEnv schedEnv0(env0);

  IloIntervalList blist(env0, 0, 100);
  blist.addPeriodicInterval(5, 2, 7, 100);
  schedEnv0.setBreakListParam(blist);

  IlcInt i=0;
  for (i = 0; i < nbOfWorkers; i++){
    workers[i] = IloUnaryResource(env0);
    model0.add(workers[i]);
  }

  IlcScheduler durSched(solver0);
  durSched.setDurable();
  solver0.extract(model0);

  /* CLOSE THE DURABLE SCHEDULER. */
  durSched.close();

  return durSched;
}

////////////////////////////////////////////
//
// TREATING A REQUEST
//
////////////////////////////////////////////

ILCGOAL3(TreatRequestIlc,
         IlcInt, reqIndex,
         IlcUnaryResource*, team,
         IlcInt, releaseDate){
```

```
  /* CREATING A SCHEDULE. */
  IloSolver solver = getSolver();
  IlcSchedule schedule(solver, 0, Horizon);
  /* LOCKING THE NEEDED RESOURCES. */
  schedule.lock(4, team);
  /* DEFINING THE PROBLEM. */
  IlcActivity* act = new (solver.getHeap()) IlcActivity[4];
  for (IlcInt i = 0; i < 4; i++){
    act[i] = IlcActivity(schedule, 1);
    solver.add(act[i].requires(team[i]));
  }
  solver.add(act[0].startsAfter(releaseDate));
  solver.add(act[1].startsAfterEnd(act[0]));
  solver.add(act[2].startsAfterEnd(act[0]));
  solver.add(act[3].startsAfterEnd(act[1]));
  solver.add(act[3].startsAfterEnd(act[2]));

  /* SOLVING THE PROBLEM. */
  solver.startNewSearch(IlcSetTimes(schedule));
  IloBool solved = solver.next();

  /* UNLOCKING THE USED COMPUTATIONAL RESOURCES. */
  schedule.unlock(4, team);

  if (solved) {
  /* ACCEPTING OR NOT. */
    IlcInt acceptDate = releaseDate + 5;
    IlcInt dueDate = act[3].getEndMin();
    if (dueDate > acceptDate){
      // Not accepting :
      solver.out() << "request " << reqIndex << ": refused " << endl;
      solver.restartSearch();
    }
    else {
      // Accepting :
      solver.out() << "request " << reqIndex
              << ": promised at " << act[3].getEndMin()
              << endl;
    }
  }
  else
    solver.out() << "request " << reqIndex
            << ": not solved " << endl;
  solver.endSearch();
  return 0;
}

ILOCPGOALWRAPPER4(TreatRequest, solver,
                  IlcScheduler, dSched,
                  IloInt, reqIndex,
                  IloUnaryResource*, team,
                  IloInt, releaseDate) {

  IlcUnaryResource* workers = new (solver.getHeap()) IlcUnaryResource[4];
  for (IlcInt i = 0; i < 4; i++)
    workers[i] = dSched.getResource(team[i]);
```

```
  return TreatRequestIlc(solver, reqIndex, workers, releaseDate);
}

////////////////////////////////////////
//
// DEFINING THE REQUESTS
//
////////////////////////////////////////

const IloInt HowManyReqs = 20;

IloInt StreamOfReqs [HowManyReqs] [2] = { /* pairs type releaseDate */
  {1, 5},    // request 0
  {2, 15},   // request 1
  {2, 7} ,   // request 2
  {1, 24},   // request 3
  {2, 2},    // request 4
  {1, 10},   // request 5
  {1, 25},   // request 6
  {1, 17},   // request 7
  {2, 8},    // request 8
  {1, 15},   // request 9
  {2, 21},   // request 10
  {2, 5},    // request 11
  {1, 15},   // request 12
  {2, 17},   // request 13
  {2, 24},   // request 14
  {2, 2},    // request 15
  {1, 10},   // request 16
  {2, 28},   // request 17
  {1, 17},   // request 18
  {2, 10}    // request 19
};

int main(int argc, char* argv[]) {
  try {
    const IloInt nbOfWorkers = 8;
    IloUnaryResource workers[nbOfWorkers];
    IloUnaryResource* firstTeam = workers;
    IloUnaryResource* secondTeam = workers + 4;

    /* CREATING THE DURABLE RESOURCES. */
    IloEnv env0;
    IloSolver solver0(env0);
    IlcScheduler sched0 =
      CreateDurableResources(solver0, nbOfWorkers, workers);

    /* CREATING THE WORKER THREADS. */
    IloInt nbOfThreads = 3;
    if (argc >= 2)
      nbOfThreads = atol(argv[1]);
    IlcWorkServer server(sched0, nbOfThreads, “output”);

    /* TREATING THE REQUESTS. */
    for (IloInt j = 0; j < HowManyReqs; j++) {
      // Reading a request :
      IloInt requestType = StreamOfReqs[j][0];
      IloInt releaseDate = StreamOfReqs[j][1];
```

```
      // Getting a computation environment :
      IloEnv env = server.getIdleEnv();
      // Solving the specific problem associated with the request:
      IloUnaryResource* team = (requestType == 1 ?
                                firstTeam :
                                secondTeam);
      server.launch(TreatRequest(env, sched0, j, team, releaseDate));
    }

    /* STOPPING THE WORKER THREADS. */
    server.end();

    /* PRINTING THE SCHEDULE OF WORKERS. */
    IlcScheduler scheduler0(solver0);
    IloSchedulerSolution solution(env0);

    for (IloInt i = 0; i < nbOfWorkers; i++) {
      solver0.out() << endl << "worker" << i << ": " << endl;
      solution.store(workers[i], scheduler0);
      for (IloNumToNumStepFunctionCursor
curs(solution.getLevelMin(workers[i]));
           curs.ok();
           ++curs)
        solver0.out() << curs.getSegmentMin() << " .. "
                      << IloMin(Horizon, (curs.getSegmentMax() - 1)) << " : "
                      << (curs.getValue() == 1 ? "On" : "Off")
                      << endl;
    }

    solver0.printInformation();
    env0.end();
  } catch (IloException& exc) {
    cout << exc << endl;
  }  return 0;
}

////////////////////////////////////////////
//
// RESULTS
//
////////////////////////////////////////////

/* durablem 0
   request 0: promised at 10
   request 1: promised at 18
   request 2: promised at 10
   request 3: promised at 29
   request 4: promised at 5
   request 5: promised at 15
   request 6: promised at 30
   request 7: promised at 22
   request 8: promised at 11
   request 9: promised at 18
   request 10: promised at 24
   request 11: refused
   request 12: promised at 19
   request 13: promised at 22
```

```
request 14: promised at 29
request 15: refused
request 16: refused
request 17: not solved
request 18: refused
request 19: promised at 15

worker0:
0 .. 6 : Off
7 .. 7 : On
8 .. 9 : Off
10 .. 10 : On
11 .. 14 : Off
15 .. 17 : On
18 .. 23 : Off
24 .. 25 : On
26 .. 29 : Off

worker1:
0 .. 7 : Off
8 .. 8 : On
9 .. 10 : Off
11 .. 11 : On
12 .. 15 : Off
16 .. 18 : On
19 .. 24 : Off
25 .. 25 : On
26 .. 27 : Off
28 .. 28 : On
29 .. 29 : Off

worker2:
0 .. 7 : Off
8 .. 8 : On
9 .. 10 : Off
11 .. 11 : On
12 .. 15 : Off
16 .. 18 : On
19 .. 24 : Off
25 .. 25 : On
26 .. 27 : Off
28 .. 28 : On
29 .. 29 : Off

worker3:
0 .. 8 : Off
9 .. 9 : On
10 .. 13 : Off
14 .. 14 : On
15 .. 16 : Off
17 .. 18 : On
19 .. 20 : Off
21 .. 21 : On
22 .. 27 : Off
28 .. 29 : On

worker4:
0 .. 1 : Off
```

```
2 .. 2 : On
3 .. 6 : Off
7 .. 8 : On
9 .. 9 : Off
10 .. 10 : On
11 .. 14 : Off
15 .. 15 : On
16 .. 16 : Off
17 .. 17 : On
18 .. 20 : Off
21 .. 21 : On
22 .. 23 : Off
24 .. 24 : On
25 .. 29 : Off

worker5:
0 .. 2 : Off
3 .. 3 : On
4 .. 7 : Off
8 .. 9 : On
10 .. 10 : Off
11 .. 11 : On
12 .. 15 : Off
16 .. 16 : On
17 .. 17 : Off
18 .. 18 : On
19 .. 21 : Off
22 .. 22 : On
23 .. 24 : Off
25 .. 25 : On
26 .. 29 : Off

worker6:
0 .. 2 : Off
3 .. 3 : On
4 .. 7 : Off
8 .. 9 : On
10 .. 10 : Off
11 .. 11 : On
12 .. 15 : Off
16 .. 16 : On
17 .. 17 : Off
18 .. 18 : On
19 .. 21 : Off
22 .. 22 : On
23 .. 24 : Off
25 .. 25 : On
26 .. 29 : Off

worker7:
0 .. 3 : Off
4 .. 4 : On
5 .. 8 : Off
9 .. 10 : On
11 .. 13 : Off
14 .. 14 : On
15 .. 16 : Off
17 .. 17 : On
```

```
   18 .. 20 : Off
   21 .. 21 : On
   22 .. 22 : Off
   23 .. 23 : On
   24 .. 27 : Off
   28 .. 28 : On
   29 .. 29 : Off
*/
```

Index

C

R

S

T

U

V