IBM ILOG CP Optimizer V2.3

Getting Started with IBM ILOG CP Optimizer

Table of contents

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.

Welcome to IBM ILOG CP Optimizer

This is the *Getting Started with IBM ILOG CP Optimizer* manual.

In this section

Overview
Describes CP Optimizer.

About this manual
Describes this manual.

Prerequisites
Describes the prerequisites for using this manual.

Related documentation
Lists the documentation related to this manual.

Installing IBM ILOG CP Optimizer
Describes where to find the instructions to install CP Optimizer.

Location of examples
Lists the location of the examples used in this manual.

Typographic and naming conventions
Describes the typographic and naming conventions used in this manual.

Overview

IBM® ILOG® CP Optimizer is a software library which provides a constraint programming engine targeting both constraint satisfaction problems and optimization problems. This engine, designed to be used in a “model & run” development process, contains powerful methods for finding feasible solutions and improving them. The strength of the optimizer removes the need for you to write and maintain a search strategy.

IBM ILOG CP Optimizer is based on IBM ILOG Concert Technology. Concert Technology offers a library of classes and functions that enable you to define models for optimization problems. Likewise, CP Optimizer offers a library of classes and functions that enable you to find solutions to the models. Though the CP Optimizer defaults will prove sufficient to solve most problems, CP Optimizer offers a variety of tuning classes and parameters to control various algorithmic choices.

IBM ILOG CP Optimizer and Concert Technology provide application programming interfaces (APIs) for Microsoft® .NET languages, C++ and Java™. The CP Optimizer part of an application can be completely integrated with the rest of that application (for example, the graphical user interface, connections to databases and so on) because it can share the same objects.

About this manual

This is the *Getting Started with IBM ILOG CP Optimizer* manual. It is composed of lessons that use a procedural-based learning strategy. Each procedural-based lesson is built around a sample problem, and you work on a partially completed code example. As you follow the steps in the lesson, you complete the code and learn about the concepts. Then, you compile and run the code and analyze the results. At the end of each lesson, there are review exercises.

This manual is designed to be used by programmers who may or may not have any knowledge of constraint programming. The ideal usage context for this manual is sitting in front of your computer, with IBM® ILOG® Concert Technology and IBM ILOG CP Optimizer installed. You work through the lessons and exercises.

If you are a novice CP Optimizer user, start at the beginning of this manual, since the lessons build on each other.

Prerequisites

IBM® ILOG® CP Optimizer requires a working knowledge of the Microsoft® .NET Framework, C++ or Java™. However, it does not require you to learn a new language since it does not impose any syntactic extensions on your programming language of choice.

If you are experienced in constraint programming or operations research, you are probably already familiar with many concepts used in this manual. However, no experience in constraint programming or operations research is required to use this manual.

You should have IBM ILOG CP Optimizer and IBM ILOG Concert Technology installed in your development environment before starting to use this manual. Moreover, you should be able to compile, link and execute a sample program provided with IBM ILOG CP Optimizer.

Related documentation

The following documentation ships with IBM® ILOG® CP Optimizer and will be useful for you to refer to as you complete the lessons and exercises.

- The *IBM ILOG CP Optimizer Reference Manuals* document the IBM ILOG CP Optimizer and IBM ILOG Concert Technology classes and functions used in the *Getting Started with IBM ILOG CP Optimizer Manual*. The reference manuals also explain certain concepts more formally. There are three reference manuals; one for each of the available APIs.
- The *IBM ILOG CP Optimizer User's Manual* explains the topics covered in this *Getting Started with IBM ILOG CP Optimizer Manual* in greater depth, with individual sections about modeling, solving and tuning the CP Optimizer search process.
- The *IBM ILOG CP Optimizer Release Notes* list new and improved features, changes in the library and documentation and issues addressed for each release.

Installing IBM ILOG CP Optimizer

In this manual, it is assumed that you have already successfully installed the IBM® ILOG® Concert Technology and CP Optimizer libraries on your platform (that is, the combination of hardware and software you are using). If this is not the case, you will find installation instructions in your IBM ILOG Electronic Product Delivery package. The instructions cover all the details you need to know to install Concert Technology and CP Optimizer on your system.

Location of examples

In each lesson you work to fill in the missing parts of a partially completed code. These partial codes are available for the C++ API and are located in the directory `YourCPHome/examples/tutorial/cpp`. The completed examples are available for C++, C# and Java™. These examples are in the directories `YourCPHome/examples/src/cpp`, `YourCPHome/examples/src/csharp` and `YourCPHome/examples/src/java`, respectively. Each example has the appropriate suffix. Thus the Java equivalent of `color.cpp` is `Color.java`, and the C# version is `Color.cs`.

Typographic and naming conventions

Important ideas are *italicized* the first time they appear.

In this manual, the examples are given in C++. In C++, the names of types, classes and functions defined in the IBM® ILOG® CP Optimizer and Concert Technology libraries begin with `Ilo`.

The name of a class is written as concatenated words with the first letter of each word in upper case (that is, capital). For example,

```
IloIntVar
```

A lower case letter begins the first word in names of arguments, instances and member functions. Other words in the identifier begin with an upper case letter. For example,

```
IloIntVar aVar;
IloIntVarArray::add;
```

Names of data members begin with an underscore, like this:

```
class Bin {
public:
  IloIntVar       _type;
  IloIntVar       _capacity;
  IloIntVarArray  _contents;
  Bin (IloModel    mod,
       IloIntArray capacity,
       IloInt      nTypes,
       IloInt      nComponents);
  void display(const IloCP cp);
};
```

Generally, accessors begin with the key word `get`. Accessors for Boolean members begin with `is`. Modifiers begin with `set`.

Names of classes, methods and symbolic constants in the C# and the Java™ APIs correspond very closely to those in the C++ API with these systematic exceptions:

- In the C# API and the Java API, namespaces are used. For Java, the namespaces are `ilog.cp` and `ilog.concert`. For C#, the namespaces are `ILOG.CP` and `ILOG.Concert`.
- In the C++ API and Java API, the names of classes begin with the prefix `Ilo` whereas in in the C# API they do not.
- In the C++ API and Java API, the names of methods conventionally begin with a lowercase letter, for example, `startNewSearch`, whereas in the C# API, the names of methods conventionally begin with an uppercase letter, for example, `StartNewSearch` according to Microsoft® practice.

To make porting easier from platform to platform, IBM ILOG CP Optimizer and Concert Technology isolate characteristics that vary from system to system.

For that reason, you are encouraged to use the following identifiers for basic types in C++:

- `IloInt` stands for signed long integers;
- `IloNum` stands for double precision floating-point values ;
- `IloBool` stands for Boolean values: `IloTrue` and `IloFalse`.

You are not obliged to use these identifiers, but it is highly recommended if you plan to port your application to other platforms.

Constraint programming with IBM ILOG CP Optimizer

This section describes the basic concepts of constraint programming.

In this section

Overview
Describes CP Optimizer.

The three-stage method
Describes the three-stage method for solving problems with CP Optimizer.

Describe
Describes the first stage in solving a problem with CP Optimizer.

Model
Describes the second stage in solving a problem with CP Optimizer.

Solve
Describes the third stage in solving a problem with CP Optimizer.

Compile and test a simple application
Describes how to compile and test an application in CP Optimizer.

Scheduling in CP Optimizer
Describes the scheduling capability of CP Optimizer.

Scheduling building blocks
Describes the basic scheduling building blocks available in CP Optimizer.

Compile and test
Describes how to compile and test a scheduling application.

Review exercises
Includes the review exercises and the suggested answers.

Overview

In this lesson, you will learn how to:

- use the three-stage method to describe, model and solve problems;
- identify decision variables and constraints;
- understand basic constraint propagation and search;
- understand the basic building blocks of a scheduling model;
- compile a sample program to ensure your installation is working correctly.

IBM® ILOG® CP Optimizer is a software library which provides a constraint programming engine targeting both constraint satisfaction problems and optimization problems. CP Optimizer provides a library of re-usable and maintainable classes that can be used just as they are, or extended to meet special needs. Those 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.

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. CP Optimizer embodies this advantage of modeling and solving in a ready-to-use library of tools.

CP Optimizer offers features specially adapted to solving problems in scheduling with continuous time and resource allocation. There are, for example, classes of objects particularly designed to represent such aspects as tasks and temporal constraints. CP Optimizer offers you a workbench of modeling features that intuitively and naturally tackle the issues inherent in scheduling and allocation problems.

IBM ILOG CP Optimizer and Concert Technology provide application programming interfaces (APIs) for Microsoft® Framework .NET languages, C++ and Java™. The CP Optimizer part of an application can be completely integrated with the rest of that application (for example, the graphical user interface, connections to databases and so on) because it can share the same objects.

The three-stage method

To find a solution to a problem using IBM® ILOG® CP Optimizer, you use a three-stage method: describe, model and solve.

The first stage is to *describe* the problem in natural language. For more information, see the section *Describe*.

The second stage is to use IBM ILOG Concert Technology classes to *model* the problem. The model is composed of decision variables and constraints. *Decision variables* are the unknown information in a problem. Each decision variable has a *domain* of possible *values*. The *constraints* are limits or restrictions on combinations of values for these decision variables. The model may also contain an *objective*, an expression that can be maximized or minimized. For more information, see the section *Model*.

The third stage is to use IBM ILOG CP Optimizer classes to *solve* the problem. Solving the problem consists of finding a value for each decision variable while simultaneously satisfying the constraints and maximizing or minimizing an objective, if one is included in the model. The IBM ILOG CP Optimizer *engine* (also called "the optimizer") uses two techniques for solving optimization problems: *constructive search* and *constraint propagation*. For more information, see the section *Solve*.

In this lesson, you will describe, model and solve a simple problem to understand the basic concepts in constraint programming. The problem is to find values for *x* and *y* from the following information:

- $x + y = 17$
- $x - y = 5$
- *x* can be any integer from 5 through 12
- *y* can be any integer from 2 through 17

Describe

The first stage is to *describe* the problem in natural language.

What is the unknown information, represented by the decision variables, in this problem?

- The values of x and y, where x is an integer between 5 and 12 inclusive and y is as integer between 2 and 17 inclusive.

What are the limits or restrictions on combinations of these values, represented by the constraints, in this problem?

- $x + y = 17$
- $x - y = 5$

Though the Describe stage of the process may seem trivial in a simple problem like this one, you will find that taking the time to fully describe a more complex problem is vital for creating a successful program. You will be able to code your program more quickly and effectively if you take the time to describe the model, isolating the decision variables and constraints.

Model

Describes the second stage in solving a problem with CP Optimizer.

In this section

Overview
Describes the second stage in solving a problem with CP Optimizer.

Decision variables
Describes the decision variables.

Constraints
Describes the constraints.

Overview

The second stage is to use IBM® ILOG® Concert Technology classes to *model* the problem. The model is composed of decision variables and constraints. The model may also contain an *objective*, although in this case it does not. For more information on modeling with an objective, see *Using arrays and objectives: warehouse location*.

Decision variables

Decision variables represent the unknown information in a problem. Decision variables differ from standard programming variables in that they have domains of possible values and may have constraints placed on the allowed combinations of theses values. For this reason, decision variables are also known as *constrained variables*. In this example, the decision variables are x and y.

Each decision variable has a domain of possible values. In this example, the domain of decision variable x is [5..12], or all integers from 5 to 12. The domain of decision variable y is [2..17], or all integers from 2 to 17.

Note: In IBM® ILOG® *CP Optimizer* and Concert Technology, square brackets denote the *domain* of decision variables. For example, [5 12] denotes a domain as a set consisting of precisely two integers, 5 and 12. In contrast, [5..12] denotes a domain as a range of integers, that is, the interval of integers from 5 to 12, so it consists of 5, 6, 7, 8, 9, 10, 11 and 12.

Constraints

Constraints are limits on the combinations of values for variables. There are two constraints on the decision variables in this example: $x + y = 17$ and $x - y = 5$.

Solve

Describes the third stage in solving a problem with CP Optimizer.

In this section

Overview
Describes the third stage in solving a problem with CP Optimizer.

Search space
Describes the search space.

Initial constraint propagation
Describes the initial constraint propagation.

Constructive search
Describes constructive search.

Constraint propagation during search
Describes constraint propagation during search.

Overview

The third stage of the process is to use IBM® ILOG® CP Optimizer classes to search for a solution and *solve* the problem. A *solution* is a set of value assignments to the constrained variables such that each variable is assigned exactly one value from its domain and such that together these values satisfy the constraints. If there is an objective in the model, then an *optimal solution* is a solution that optimizes the objective function. Solving the problem consists of finding a solution for the problem or an optimal solution, if an objective is included in the model. The IBM ILOG CP Optimizer engine utilizes efficient algorithms for finding solutions to Constraint satisfaction and optimization problems.

Search space

The IBM® ILOG® CP Optimizer engine explores the *search space* to find a solution. The search space is all combinations of values. One way to find a solution would be to explicitly study each combination of values until a solution was found. Even for this simple problem, this approach is obviously time-consuming and inefficient. For a more complicated problem with many variables, the approach would be unrealistic.

The optimizer uses two techniques to find a solution: constructive search and constraint propagation. Additionally, the optimizer performs two types of constraint propagation: *initial constraint propagation* and *constraint propagation during search*.

Initial constraint propagation

First, the IBM® ILOG® CP Optimizer engine performs an initial constraint propagation. The initial constraint propagation removes values from domains that will not take part in *any* solution. Before propagation, the domains are:

```
D(x) = [5 6 7 8 9 10 11 12]
D(y) = [2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17]
```

To get an idea of how initial constraint propagation works, consider the constraint $x + y = 17$. If you take the smallest number in the domain of x, which is 5, and add it to the largest number in the domain of y, which is 17, the answer is 22. This combination of values ($x = 5$, $y = 17$) violates the constraint $x + y = 17$. The only value of x that would work with $y = 17$ is $x = 0$. However, there is no value of 0 in the domain of x, so y cannot be equal to 17. The value $y = 17$ cannot take part in any solution. The domain reduction algorithm employed by the constraint propagation engine removes the value $y = 17$ from the domain of y. Similarly, the propagation engine removes the following values from the domain of y: 13, 14, 15 and 16.

Likewise, if you take the largest number in the domain of x, which is 12, and add it to the smallest number in the domain of y, which is 2, the answer is 14. This combination of values ($x = 12$, $y = 2$) violates the constraint $x + y = 17$. The only value of x that would work with $y = 2$ is $x = 15$. However, there is no value of 15 in the domain of x, so y cannot be equal to 2. The value of $y = 2$ cannot take part in any solution. the propagation engine removes the value $y = 2$ from the domain of y. For the same reason, the domain reduction algorithm employed by the propagation engine removes the following values from the domain of y: 2, 3 and 4.

After initial propagation for the constraint $x + y = 17$, the domains are:

```
D(x) = [5 6 7 8 9 10 11 12]
D(y) = [5 6 7 8 9 10 11 12]
```

Now, examine the constraint $x - y = 5$. If you take the value 5 in the domain of x, you can see that the only value of y that would work with $x = 5$ is $y = 0$. However, there is no value of 0 in the domain of y, so x cannot equal 5. The value $x = 5$ cannot take part in any solution. The propagation engine removes the value $x = 5$ from the domain of x. Using similar logic, the propagation engine removes the following values from the domain of x: 6, 7, 8 and 9. Likewise, the domain reduction algorithm employed by the propagation engine removes the following values from the domain of y: 8, 9, 10, 11 and 12.

Returning to the other constraint, there are no further values that can be removed from the variables. After initial propagation, the search space has been reduced in size. The domains are now:

```
D(x) = [10 11 12]
D(y) = [5 6 7]
```

Constructive search

After initial constraint propagation, the search space is reduced. IBM® ILOG® CP Optimizer will use a constructive search strategy to guide the search for a solution in the remaining part of the search space. It may help to think of the strategy as one that traverses a *search tree*. The *root* of the tree is the starting point in the search for a solution; each *branch* descending from the root represents an alternative in the search. Each combination of values in the search space can be seen as a *leaf node* of the search tree.

The IBM ILOG CP Optimizer engine executes a search strategy that guides the search for a solution. The optimizer "tries" a value for a variable to see if this will lead to a solution. To demonstrate how the optimizer uses search strategies to find a solution, consider a search strategy that specifies that the optimizer should select variable *x* and assign it the lowest value in the domain of *x*. For the first *search move* in this strategy, the optimizer assigns the value 10 to the variable *x*. This move, or search tree branch, is not permanent. If a solution is not found with $x = 10$, then the optimizer can undo this move and try a different value of *x*.

Constraint propagation during search

The IBM® ILOG® CP Optimizer engine performs constraint propagation during search. This constraint propagation differs from the initial constraint propagation. The initial constraint propagation removes all values from domains that will not take part in *any* solution. Constraint propagation during search removes all values from the *current* domains that violate the constraints. You can think of constraint propagation during search in the following way. In order to “try” a value for a variable, the optimizer creates “test” or *current* domains. When constraint propagation removes values from domains during search, values are only removed from these “test” domains.

To continue the same example, suppose that, based on the search strategy, the optimizer has assigned the value 10 to the decision variable *x*. Working with the constraint $x + y = 17$, constraint propagation reduces the domain of *y* to [7]. However, this combination of values ($x = 10$, $y = 7$) violates the constraint $x - y = 5$. The optimizer removes the value $y = 7$ from the current domain of *y*. At this point, the domain of *y* is empty, and the optimizer encounters a *failure*. The optimizer can then conclude that there is no possible solution with the value of 10 assigned to *x*.

When the optimizer decides to try a different value for the decision variable *x*, the domain of *y* is at first restored to the values [5 6 7]. It then reduces the domain of *y* based on the new value assigned to *x*.

This simple example demonstrates the basic concepts of constructive search and constraint propagation. To summarize, solving a problem consists of finding a value for each decision variable while simultaneously satisfying the constraints. The IBM ILOG CP Optimizer engine uses two techniques to find a solution: constructive search with search strategies and constraint propagation. Additionally, the optimizer performs two types of constraint propagation: initial constraint propagation and constraint propagation during search.

The initial constraint propagation removes values from domains that will not take part in any solution. After initial constraint propagation, the search space is reduced. This remaining part of the search space, where the IBM ILOG CP Optimizer engine will use constructive search with a search strategy to search for a solution, is called the search tree. Constructive search is a way to “try” a value for a variable to see if this will lead to a solution. The optimizer performs constraint propagation during search. Constraint propagation during search removes all values from the current or “test” domains that violate the constraints. If the optimizer cannot find a solution after a series of choices, these can be reversed and alternatives can be tried. The CP Optimizer engine continues to search using constructive search and constraint propagation during search until a solution is found.

Compile and test a simple application

This first lesson is designed to help you understand the basic concepts in constraint programming. In future lessons, you will work through problems by describing, modeling and solving problems using IBM® ILOG® Concert Technology and CP Optimizer classes. In this lesson, you are provided with the completed example code so that you can test your installation of IBM ILOG CP Optimizer.

In *Modeling and solving a simple problem with integer variables: map coloring*, you will learn about the classes and member functions used in this program.

The following code using the C++ API models and solves the problem introduced in this lesson:

```
#include <ilcp/cp.h>

int main(int argc, const char * argv[]){
  IloEnv env;
  try {
    IloModel model(env);
    IloIntVar x(env, 5, 12, "x");
    IloIntVar y(env, 2, 17, "y");
    model.add(x + y == 17);
    model.add(x - y == 5);
    IloCP cp(model);
    if (cp.solve()){
      cp.out() << std::endl << "Solution:" << std::endl;
      cp.out() << "x = " << cp.getValue(x) << std::endl;
      cp.out() << "y = " << cp.getValue(y) << std::endl;
    }
  }
  catch (IloException& ex) {
    env.out() << "Error: " << ex << std::endl;
  }
  env.end();
  return 0;
}
```

Open the example file `YourCPHome/examples/src/cpp/intro.cpp` in your development environment. To test your installation of IBM ILOG CP Optimizer, build and run the program. When you run the program, you should get results similar to this output:

```
 ! ------------------------------------------------------------------------
--
 ! Satisfiability problem - 2 variables, 2 constraints
 ! Initial process time : 0.00s (0.00s extraction + 0.00s propagation)
 !  . Log search space  : 7.0 (before), 3.2 (after)
 !  . Memory usage      : 315.4 Kb (before), 315.4 Kb (after)
 ! ------------------------------------------------------------------------
--
 !   Branches  Non-fixed               Branch decision
```

```
 *           2      0.03s                        y !=    5
 ! ----------------------------------------------------------------------------
--
 ! Solution status        : Terminated normally, solution found
 ! Number of branches     : 2
 ! Number of fails         : 1
 ! Total memory usage     : 328.3 Kb (315.4 Kb CP Optimizer + 12.9 Kb Concert)

 ! Time spent in solve    : 0.03s (0.03s engine + 0.00s extraction)
 ! Search speed (br. / s) : 64.0
 ! ----------------------------------------------------------------------------
--

Solution:
x = 11
y = 6
```

The solution found by the IBM ILOG CP Optimizer engine is $x = 11$ and $y = 6$. In addition to the solution, information regarding the progress of the optimizer is displayed; this information is called the search log, or the log.

The first line of the log indicates the type of problem, along with the number of decision variables and constraints in the model. In this case, there is no objective included in the model, so the problem is reported to be a “Satisfiability problem”. When the model includes an objective, the problem type is reported as a “Minimization problem” or a “Maximization problem” depending on the type of objective. The next three lines of the log provide information regarding the initial constraint propagation. The “Initial process time” is the time in seconds spent at the root node of the search tree where the initial propagation occurs. This time encompasses the time used by the optimizer to load the model, called *extraction*, and the time spent in initial propagation. The value for “Log search space” provides an estimate on the size of the depth-first search tree; this value is the log (base 2) of the products of the domains sizes of all the decision variables of the problem. Typically, the estimate of the size of the search tree should be smaller after the initial propagation, as choices will have been eliminated. However, this value is always an overestimate of the log of the number of remaining leaf nodes of the tree because it does not take into account the action of propagation of constraints at each node. The memory used by the optimizer during the initial propagation is reported.

In order to interpret the remainder of the log file, you may want to think about the search as a binary tree. The root of the tree is the starting point in the search for a solution; each branch descending from the root represents an alternative choice or decision in the search. Each of these branches leads to a node where constraint propagation during search will occur. If the branch does not lead to a failure and a solution is not found at a node, the node is called a *choice point*. The optimizer can make an additional decision and create two new alternative branches from the current node, or it can jump in the tree and search from another node.

The lines in the next section of the progress log, are displayed periodically during the search and describe the state of the search. The display frequency of the progress log can be controlled with parameters of the optimizer. Since the problem in the lesson is a simple one, only one update is displayed.

The progress information given in a progress log update includes:

- Branches: the number of branches explored in the binary search tree;

- Non-fixed: the number of uninstantiated (not fixed) model variables;
- Branch decision: the decision made at the branch currently under consideration by the optimizer.

The final lines of the log provide information about the entire search, after the search has terminated. This information about the search includes:

- Solution status: the conditions under which the search terminated;
- Number of branches: the number of branches explored in the binary search tree;
- Number of fails: the number of branches that did not lead to a solution;
- Total memory usage: the memory used by IBM ILOG Concert Technology and the IBM ILOG CP Optimizer engine;
- Time spent in solve: the elapsed time from start to the end of the search displayed in hh:mm:ss.ff format;
- Search speed: average time spent per branch.

In subsequent lessons in this manual, the log output is not included in the sections which report the results. You can view the log information when you build and run the completed programs.

Note: The CP Optimizer search log is meant for visual inspection only, not for mechanized parsing. In particular, the log may change from version to version of CP Optimizer in order to improve the quality of information displayed in the log. Any code based on the log output may have to be updated when a new version of CP Optimizer is released.

Scheduling in CP Optimizer

IBM® ILOG® CP Optimizer offers features specially adapted to solving problems in scheduling with continuous time and resource allocation. There are, for example, classes of objects particularly designed to represent such aspects as tasks and temporal constraints. CP Optimizer offers you a workbench of modeling features that intuitively and naturally tackle the issues inherent in scheduling and allocation problems.

Scheduling with fine-grained time can be seen as the process of assigning start and end times to intervals. Scheduling problems also require the management of minimal or maximal capacity constraints for resources over time.

CP Optimizer provides features for modeling and solving scheduling problems where time intervals (activities, operations, tasks) need to be placed in time and may have a resource allocation aspect.

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

- combined scheduling and planning problems,
- combined scheduling and configuration problems and
- combined 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 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. 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.

IBM ILOG CP Optimizer can be used to create a variety of scheduling applications, including:

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

Scheduling building blocks

Describes the basic scheduling building blocks available in CP Optimizer.

In this section

Overview
Describes the basic scheduling building blocks available in CP Optimizer.

Interval variables
Describes interval decision variables.

Scheduling constraints
Describes the scheduling constraints.

Overview

Scheduling is the act of creating a schedule, which is a timetable for planned occurrences. Scheduling may also involve allocating resources to activities over time.

A scheduling problem is defined by:

- a set of time intervals--definitions of activities, operations or tasks to be completed;
- a set of temporal constraints--definitions of possible relationships between the start and end times of the intervals;
- a set of specialized constraints--definitions of the complex relationships on a set of intervals due to the state and finite capacity of resources.

Many of the lessons in this manual provide deeper explanations of how CP Optimizer facilitates the representation of scheduling problems.

Interval variables

In IBM® ILOG® CP Optimizer, activities, operations and tasks are represented as interval decision variables.

An interval has a start, an end, a length and a size. An interval variable allows for these values to be variable within the model. The start is the lower endpoint of the interval and the end is the upper endpoint of the interval. By default, the size is equal to the length, which is the difference between the end and the start of the interval. In general, the size is a lower bound on the length.

Also, an interval variable may be optional, and whether or not an interval is present in the solution is represented by a decision variable. If an interval is not present in the solution, this means that any constraints on this interval act like the interval is “not there.” Exact semantics will depend on the specific constraint.

In CP Optimizer, the optionality of an interval is captured by the notion of a Boolean presence status associated with each interval variable. Logical relations can be expressed between the presence statuses of interval variables, allowing, for instance, to state that whenever interval variable `a` is present then interval variable `b` must also be present.

Scheduling constraints

Several types of constraints that can be placed on interval variables:

- precedence constraints, which ensure that the relative positions of intervals in the solution (For example a precedence constraint can model a requirement that an interval `a` must end before interval `b` starts, optionally with some minimum delay `z`.);
- no overlap constraints, which ensure that positions of intervals in the solution to be disjoint in time;
- span constraints, which ensure that one interval covers the intervals in a set of intervals;
- alternative constraints, which ensure that exactly one of a set of intervals be present in the solution;
- synchronize constraints, which ensure that a set of intervals start and end at the same time as a given interval variable if it is present in the solution;
- cumulative expression constraints, which restrict the bounds of the domains of cumulative expressions;
- state constraints, which ensure that an interval be positioned based on the value of a state function.

Compile and test

In this lesson, you are provided with a completed example code so that you can test your installation of IBM® ILOG® CP Optimizer. In *Using expressions on interval variables: house building with earliness and tardiness costs*, you will learn about the classes and member functions used in this program.

The problem is a house building problem in which there are ten tasks of fixed size, each of which needs to be assigned a starting time. Using the **C++ API**, the code for creating the environment, model and interval variables that represent the tasks is:

```
#include <ilcp/cp.h>

int main(int argc, const char * argv[]) {
  IloEnv env;
  try {
    IloModel model(env);

    /* CREATE THE TIME-INTERVALS. */
    IloIntervalVar masonry  (env, 35, "masonry   ");
    IloIntervalVar carpentry(env, 15, "carpentry ");
    IloIntervalVar plumbing (env, 40, "plumbing  ");
    IloIntervalVar ceiling  (env, 15, "ceiling   ");
    IloIntervalVar roofing  (env, 5,  "roofing   ");
    IloIntervalVar painting (env, 10, "painting  ");
    IloIntervalVar windows  (env, 5,  "windows   ");
    IloIntervalVar facade   (env, 10, "facade    ");
    IloIntervalVar garden   (env, 5,  "garden    ");
    IloIntervalVar moving   (env, 5,  "moving    ");
```

The constraints in this problem are precedence constraints; some tasks cannot start until other tasks have ended. For example, the ceilings must be completed before painting can begin. The set of precedence constraints for this problem can be added to the model with the code:

```
    model.add(IloEndBeforeStart(env, masonry,   carpentry));
    model.add(IloEndBeforeStart(env, masonry,   plumbing));
    model.add(IloEndBeforeStart(env, masonry,   ceiling));
    model.add(IloEndBeforeStart(env, carpentry, roofing));
    model.add(IloEndBeforeStart(env, ceiling,   painting));
    model.add(IloEndBeforeStart(env, roofing,   windows));
    model.add(IloEndBeforeStart(env, roofing,   facade));
    model.add(IloEndBeforeStart(env, plumbing,  facade));
    model.add(IloEndBeforeStart(env, roofing,   garden));
    model.add(IloEndBeforeStart(env, plumbing,  garden));
    model.add(IloEndBeforeStart(env, windows,   moving));
    model.add(IloEndBeforeStart(env, facade,    moving));
    model.add(IloEndBeforeStart(env, garden,    moving));
```

```
    model.add(IloEndBeforeStart(env, painting,  moving));
```

Here there is a special constraint, `IloEndBeforeStart`, which ensures that one interval variable ends before the other starts. This constraint is handled specially by the engine. One reason is to correctly treat the presence of intervals so that if one of the interval variables is not present, the constraint is automatically satisfied, and another reason is for stronger inference in constraint propagation.

The interval variables and precedence constraints completely describe this simple problem. An optimizer object (an instance of the class `IloCP`) is used to find a solution to the model, assigning values to the start and end of each of the interval variables in the model. The last part of the code for this example is:

```
    IloCP cp(model);
    if (cp.solve()) {
      cp.out() << cp.domain(masonry)   << std::endl;
      cp.out() << cp.domain(carpentry) << std::endl;
      cp.out() << cp.domain(plumbing)  << std::endl;
      cp.out() << cp.domain(ceiling)   << std::endl;
      cp.out() << cp.domain(roofing)   << std::endl;
      cp.out() << cp.domain(painting)  << std::endl;
      cp.out() << cp.domain(windows)   << std::endl;
      cp.out() << cp.domain(facade)    << std::endl;
      cp.out() << cp.domain(garden)    << std::endl;
      cp.out() << cp.domain(moving)    << std::endl;
    } else {
      cp.out() << "No solution found. " << std::endl;
    }
    cp.printInformation();
  } catch (IloException& ex) {
    env.out() << "Error: " << ex << std::endl;
  }
  env.end();
  return 0;
}
```

Open the example file `YourCPHome/examples/src/cpp/sched_intro.cpp` in your development environment. To test your installation of IBM ILOG CP Optimizer, build and run the program. When you run the program, you should get results similar to this search log output:

```
 ! ----------------------------------------------------------------------------
--
 ! Satisfiability problem - 10 variables, 14 constraints
 ! Initial process time : 0.00s (0.00s extraction + 0.00s propagation)
 !  . Log search space  : 33.2 (before), 33.2 (after)
 !  . Memory usage      : 315.4 Kb (before), 315.4 Kb (after)
 ! ----------------------------------------------------------------------------
--
 !   Branches  Non-fixed               Branch decision
 *         12      0.00s                 on moving
 ! ----------------------------------------------------------------------------
```

```
--
 ! Solution status        : Terminated normally, solution found
 ! Number of branches     : 12
 ! Number of fails        : 0
 ! Total memory usage     : 347.2 Kb (331.4 Kb CP Optimizer + 15.8 Kb Concert)

 ! Time spent in solve    : 0.00s (0.00s engine + 0.00s extraction)
 ! Search speed (br. / s) : 1200.0
 ! ----------------------------------------------------------------------------
--
masonry   [1: 0 -- 35 --> 35]
carpentry [1: 35 -- 15 --> 50]
plumbing  [1: 35 -- 40 --> 75]
ceiling   [1: 35 -- 15 --> 50]
roofing   [1: 50 -- 5 --> 55]
painting  [1: 50 -- 10 --> 60]
windows   [1: 55 -- 5 --> 60]
facade    [1: 75 -- 10 --> 85]
garden    [1: 75 -- 5 --> 80]
moving    [1: 85 -- 5 --> 90]
Number of branches      : 12
Number of fails         : 0
Number of choice points : 13
Number of variables     : 10
Number of constraints   : 2
Total memory usage      : 347.2 Kb (331.4 Kb CP + 15.8 Kb Concert)
Time in last solve      : 0.00s (0.00s engine + 0.00s extraction)
Total time spent in CP  : 0.02s
```

To understand the solution found by CP Optimizer to this satisfiability scheduling problem, consider the line:

```
masonry [1: 0 -- 35 --> 35]
```

The interval variable representing the masonry task, which has size 35, has been assigned the interval [0,35). Masonry starts at time 0 and ends at the time point 35.

Note: **Displaying interval variables**

After a time interval has been assigned a start value (say `s`) and an end value (say `e`), the interval is written as `[s,e)`. The time interval does not include the endpoint `e`. If another interval variable is constrained to be placed after this interval, it can start at the time `e`.

In subsequent lessons in this manual, the log output is not included in the sections which report the results. You can view the log information when you build and run the completed programs.

Review exercises

Includes the review exercises and the suggested answers.

In this section

Exercises
Lists the review exercises.

Suggested answers
Provides the suggested answers to the review exercises.

Exercises

For answers, see *Suggested answers*.

1. To find a solution to a problem using CP Optimizer, you use the three-stage method: describe, model and solve. What does each of these stages involve?
2. What are decision variables?
3. What are constraints?
4. What are the two techniques that the CP Optimizer uses to find a solution?
5. What are interval decision variables?
6. What are some types of constraints on interval decision variables?

Suggested answers

Exercise 1

To find a solution to a problem using CP Optimizer, you use the three-stage method: describe, model and solve. What does each of these stages involve?

Suggested answer

1. Describe

   Write a natural language description of problem.

2. Model

   Use Concert Technology classes and functions to model the problem: declare the variables (unknowns) in the problem and add constraints on the variables to the model.

3. Solve

   Use CP Optimizer classes and functions to search for a solution.

Exercise 2

What are decision variables?

Suggested answer

Decision variables are the unknown information in a problem. Each decision variable has a domain of possible values.

Exercise 3

What are constraints?

Suggested answer

Constraints are limits on the combinations of values for decision variables.

Exercise 4

What are the two techniques that the CP Optimizer engine uses to find a solution?

Suggested answer

The CP Optimizer engine uses two techniques to find a solution: constructive search and constraint propagation.

Exercise 5

What are interval decision variables?

Suggested answer

Interval decision variables represent part of the unknown information in a scheduling problem. Each interval decision variable represents the start, the end, the size and the length of a task, activity or operation. An interval decision variable also represents whether or not the interval is present in the solution.

Exercise 6

What are some types of constraints on interval decision variables?

Suggested answer

Some types of constraints on interval decision variables are precedence, no overlap, span, alternative, synchronize, cumulative expression and state constraints.

Modeling and solving a simple problem with integer variables: map coloring

This section describes how to model and solve a simple problem using integer decision variables.

In this section

Overview
Describes how to model and solve a simple problem with integer decision variables.

Describe
Describes the first stage in finding a solution to the map coloring problem.

Model
Describes the second stage in finding a solution to the map coloring problem.

Solve
Describes the third stage in finding a solution to the map coloring problem.

Review exercises
Includes the review exercises and suggested answers.

Complete program
Lists the location of the complete map coloring program and the results.

Overview

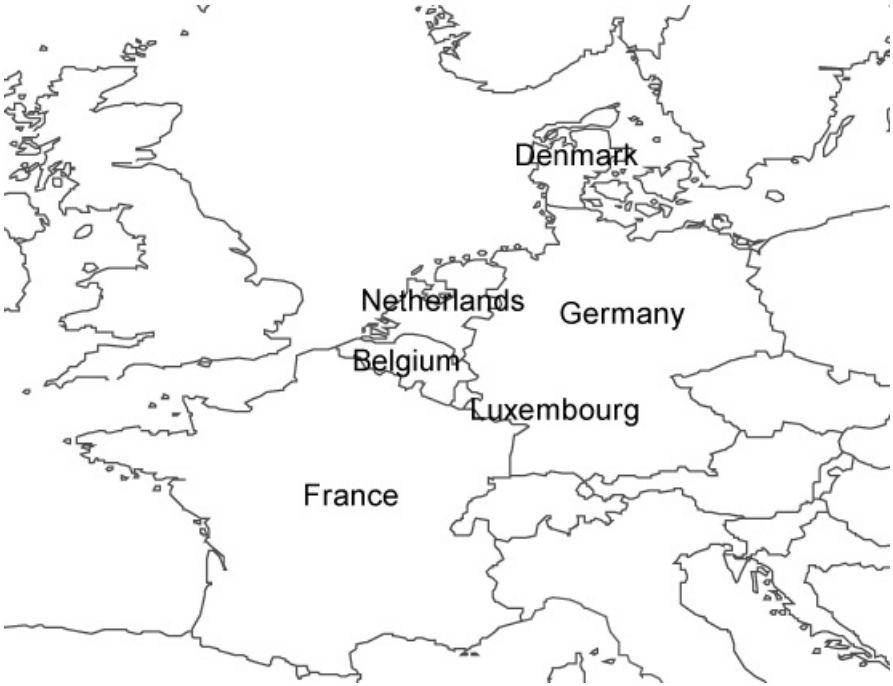
In this lesson, you will learn how to:

- use the classes `IloEnv` and `IloModel` to create an environment and a model;
- use the class `IloIntVar` to declare decision variables;
- use the class `IloConstraint` to place simple arithmetic constraints on decision variables;
- use the class `IloCP` to search for solutions and output the results.

You will learn how to model and solve a simple problem, a map coloring problem. To find a solution to this problem using IBM® ILOG® CP Optimizer, you will use the three-stage method: describe, model and solve.

Describe

The problem involves choosing colors for the countries on a map in such a way that at most four colors (blue, white, yellow, green) are used, and no pair of neighboring countries are the same color. In this lesson, you will find a solution for a map coloring problem with six countries: Belgium, Denmark, France, Germany, Luxembourg and the Netherlands. Map coloring problems are closely related to graph coloring problems and have many real world applications.



Map for coloring

Step 1: Describe the problem

The first step in modeling and solving a problem is to write a natural language description of the problem, identifying the decision variables and the constraints on these variables.

Write a natural language description of this problem.

Answer these questions:

- What is the known information in this problem?
- What are the decision variables or unknowns in this problem?
- What are the constraints on these variables?

Note: **Decision variables and constraints**

Decision variables are the unknown information in a problem. Each decision variable has a *domain* of possible *values*.

Constraints are limits or restrictions on combinations of values for decision variables.

For more information on decision variables and constraints, see the section *Model*.

Discussion

What is the known information in this problem?

- There are six known countries on a map.
- There are four known colors.

What are the decision variables or unknowns in this problem?

- The unknown is the connection between the country and the color: what color is each country? In other words, there are six variables; each variable represents the color of a specific country on the map. Each decision variable has a domain of four possible values: blue, white, yellow and green.

What are the constraints on these variables?

- In this case, each constraint specifies that a country cannot be assigned the same color as any of its neighbors. In other words, if the value of the decision variable representing the color of Belgium is blue, then the value of the variable representing the color of Germany cannot also be blue, since they are neighboring countries.

Model

After you have written a description of your problem, you can use IBM® ILOG® Concert Technology classes to model it. After you create a model of your problem, you can use IBM ILOG CP Optimizer classes and functions to search for a solution.

Step 2: Open the example file

Open the example file `YourCPHome/examples/tutorial/cpp/color_partial.cpp` in your development environment. This file is a program that is only partially completed. You will fill in the blanks in each step in this lesson. At the end, you will have completed the program code, and you can compile and run the program.

In this lesson, you include the header file `<ilcp/cp.h>`. To catch exceptions that may be thrown, you use a `try/catch` block. The code for creating the array of names for the color values and for printing out the solution found by the CP Optimizer engine is provided.

The first step in converting your natural language description of the problem into code using Concert Technology classes is to create an *environment* and a *model*.

Note: **Environment**

An instance of the class `IloEnv` manages the internal modeling issues, which include handling output, memory management for modeling objects and termination of search algorithms.

This instance is typically referred to as the *environment*. Normally an application needs only one environment, but you can create as many environments as you wish.

In the C# and Java™ APIs, the environment object is not public. To free memory used by a model in C# API, you use the method `CP.End`. To free memory used by a model in the Java API, you use the method `IloCP.end`.

Step 3: Create the environment

Add the following code after the comment `//Create the environment`

```
IloEnv env;
```

The initialization of the environment creates internal data structures to be used in the rest of the code. After this initialization is performed, you can create a *model*.

Note: **Model**

A model is a container for modeling objects such as decision variables, objectives and constraints.

The first argument passed to the constructor of the class `IloModel` is the environment. The second argument is an optional name used for debug and trace purposes. Here is a constructor:

```
IloModel(const IloEnv env, const char * name=0);
```

Step 4: Create the model

Add the following code after the comment `//Create the model`

```
  IloModel model(env);
```

After solving your problem, you can reclaim memory for all modeling objects and clean up internal data structures by calling `IloEnv::end` for every environment you have created. This clean-up should always be done before you exit your application. The call to end the environment is already included in the lesson code.

In the Microsoft® .NET Framework languages and Java™ APIs, all memory associated with the solving of the problem is reclaimed automatically when the optimizer object is garbaged.

IBM ILOG Concert Technology gives you the means to represent the unknowns in this problem, the color of each country on the map, as constrained integer, or decision, variables.

Note: **Decision Variable**

Integer decision variables are represented by `IloIntVar` in the C++ and Java APIs and by `IIntVar` in the C# API of Concert Technology.

A constrained integer variable is a decision variable that has integers as possible values. Typically, the possible values are represented as a domain of integers with an upper bound and a lower bound. These variables are said to be constrained because constraints can be placed on them.

The first argument passed to the constructor of the class `IloIntVar` is always the environment. The second argument is the lower bound of the domain of possible values, which defaults to 0. The third argument is the upper bound of the domain of possible values. The upper bound defaults to `IloIntMax`, which represents the largest possible positive integer on a given platform. The fourth argument is an optional name used for debug and trace purposes. Here is a constructor:

```
IloIntVar(const IloEnv env,
          IloInt lb = 0,
          IloInt ub = IloIntMax
          const char* name = 0);
```

After you create an environment and a model, you declare the decision variables, one for each country. Each variable represents the unknown information, the color of the country. The domain of each decision variable is [0..3] or 0 to 3 inclusive.

These values represent the possible colors:

- the value 0 represents the color blue;
- the value 1 represents the color white;
- the value 2 represents the color yellow;
- the value 3 represents the color green.

After the problem is solved, the values assigned to these decision variables will represent the solution to the problem.

Step 5: Declare the decision variables

Add the following code after the comment `//Declare the decision variables`

```
IloIntVar Belgium(env, 0, 3), Denmark(env, 0, 3),
  France(env, 0, 3), Germany(env, 0, 3),
  Luxembourg(env, 0, 3), Netherlands(env, 0, 3);
```

IBM ILOG Concert Technology allows you to express constraints involving decision variables using the following operators:

- equality (==),
- less than or equal to (<=),
- less than (<),
- greater than or equal to (>=),
- greater than (>) and
- not equal to (!=).

In this example, you use a constraint to require that if two countries are neighbors, they cannot be the same color. For example, the statement `Belgium != France` indicates that the neighbors France and Belgium should not share the same color. Explicitly, it means that the value the CP Optimizer engine assigns to the decision variable `Belgium` cannot equal the value the optimizer assigns to the variable `France`.

Note: **Constraint**

Constraints specify the restrictions on the values that may be assigned to decision variables. To create a constraint for a model, you can:

- use an arithmetic operator between decision variables and expressions to return a constraint,
- use a function that returns a constraint,
- use a specialized constraint or
- use a logical operator between constraints which returns a constraint.

Expressions, logical constraints and specialized constraints will all be introduced in later lessons.

After you create a constraint, you must explicitly add it to the model in order for it to be taken into account by the CP Optimizer engine.

You use the member function `IloModel::add` to add constraints to the model. You must explicitly add a constraint to the model object or the CP Optimizer engine will not use it in the search for a solution.

Step 6: Add the constraints

Add the following code after the comment `//Add the constraints`

```
model.add(Belgium != France);
model.add(Belgium != Germany);
model.add(Belgium != Netherlands);
model.add(Belgium != Luxembourg);
model.add(Denmark != Germany );
model.add(France != Germany);
model.add(France != Luxembourg);
model.add(Germany != Luxembourg);
model.add(Germany != Netherlands);
```

Solve

Solving a problem consists of finding a value for each decision variable so that all constraints are satisfied. You may not always know beforehand whether there is a solution that satisfies all the constraints of the problem. In some cases, there may be no solution. In other cases, there may be many solutions to a problem.

You use an instance of the class `IloCP` to solve a problem expressed in a model.

Note: **Optimizer**

The class `IloCP` in the C++ API and the Java™ API and the class `CP` in the C# API can be used to employ different algorithms for solving problems modeled with Concert Technology modeling classes.

An object of this class is sometimes referred to as the optimizer.

The constructor for `IloCP` takes an `IloModel` as its argument.

Step 7: Create an instance of IloCP

Add the following code after the comment `//Create an instance of IloCP`

```
IloCP cp(model);
```

You now use the member function `IloCP::solve`, which solves the problem contained in the model by using constructive search and constraint propagation. It is possible to modify the search using *tuning objects* and *search parameters* (more on this in *Using search parameters: team building*).

Step 8: Search for a solution

Add the following code after the comment `//Search for a solution`

```
if (cp.solve())
```

The member function `IloCP::solve` returns a Boolean value of type `IloBool`. If a solution is found, a value of `IloTrue` is returned.

After a solution has been found, you can use member functions of `IloAlgorithm`, the base class for algorithms in IBM® ILOG® Concert Technology, to examine that solution. The member function `IloAlgorithm::getValue` takes a decision variable as an argument and returns the value the IBM ILOG CP Optimizer engine has assigned to that variable. When you print the solution, you associate each value with the name of a color using the array `Names[]`. The value `0` is associated with `Names[0]`, which is the first array element “blue”;

the value `1` is associated with `Names[1]`, which is the second array element “white”; and so on. The array `Names[]` is already created for you in the lesson code.

The member function `IloAlgorithm::getStatus` returns a status code which provides information about the solution that the optimizer has found. The stream `IloAlgorithm::out` is the communication stream for general output.

The code for displaying the solution has been provided for you:

```
     cp.out() << std::endl << cp.getStatus() << " Solution" << std::endl;
     cp.out() << "Belgium:     " << Names[cp.getValue(Belgium)] << std::endl;

     cp.out() << "Denmark:     " << Names[cp.getValue(Denmark)] << std::endl;

     cp.out() << "France:      " << Names[cp.getValue(France)] << std::endl;

     cp.out() << "Germany:     " << Names[cp.getValue(Germany)] << std::endl;

     cp.out() << "Luxembourg:  " << Names[cp.getValue(Luxembourg)]  <<
std::endl;
     cp.out() << "Netherlands: " << Names[cp.getValue(Netherlands)] <<
std::endl;
```

Note: If there is more than one set of values for the variables that satisfy the constraints of the problem, there is more than one solution. Within your problem, there may be certain criteria that make one such set of values more appropriate than another as a solution. This appropriateness is usually measured in terms of a cost function that can be optimized. You can read more on cost functions, also called objectives, in *Using arrays and objectives: warehouse location*

Step 9: Compile and run the program

Compile and run the program. You should get the following results:

```
Feasible Solution
Belgium:     yellow
Denmark:     blue
France:      blue
Germany:     white
Luxembourg:  green
Netherlands: blue
```

As you can see, all four colors are used.

The complete program can be viewed online in the `YourCPHome/examples/src/cpp/color.cpp` file.

Review exercises

Includes the review exercises and suggested answers.

In this section

Exercises
Lists the review exercises.

Suggested answers
Provides the suggested answers to the review exercises.

Exercises

For answers, see *Suggested answers*.

1. What is a decision variable?
2. How do you place a constraint on decision variables?
3. Change the code so that Germany and Denmark are always the same color. Except for Germany and Denmark, no two neighboring countries are the same color.
4. Change the original code to include Switzerland in the map. Switzerland shares borders with France and Germany. No two neighboring countries are the same color.

Suggested answers

Exercise 1

What is a decision variable?

Suggested answer

A decision variable is a constrained integer variable that takes integers as possible values. A constrained integer variable is represented in Concert Technology by the class `IloIntVar`. The possible values are represented as a domain of integers with an upper bound and a lower bound.

Exercise 2

How do you place a constraint on decision variables?

Suggested answer

You create an instance of the class `IloConstraint` and explicitly add that constraint to the model using `IloModel::add`. If the constraint is not added to the model object, it will not be considered in the search.

Exercise 3

Change the code so that Germany and Denmark are always the same color. Except for Germany and Denmark, no two neighboring countries are the same color.

Suggested answer

The code that has changed from `color.cpp` follows. You can view the complete program online in the file `YourCPHome/examples/src/cpp/color_ex3.cpp`.

This constraint between Denmark and Germany is changed to be:

```
   model.add(Denmark == Germany);
```

You should obtain the following results:

```
Feasible Solution
Belgium:     green
Denmark:     white
France:      yellow
Germany:     white
Luxembourg:  blue
Netherlands: blue
```

Exercise 4

Change the original code to include Switzerland in the map. Switzerland shares borders with France and Germany. No neighboring countries are the same color.

Suggested answer

The code that has changed from `color.cpp` follows. You can view the complete program online in the file `YourCPHome/examples/src/cpp/color_ex4.cpp`.

The declaration of the variables is changed as follows:

```
   IloIntVar Belgium(env, 0, 3), Denmark(env, 0, 3), France(env, 0, 3),
     Germany(env, 0, 3), Luxembourg(env, 0, 3), Netherlands(env, 0, 3),
     Switzerland(env, 0, 3);
```

These additional constraints are added to the model:

```
   model.add(France != Switzerland);
   model.add(Germany != Switzerland);
```

This code is added to the solution display:

```
      cp.out() << "Switzerland: " << Names[cp.getValue(Switzerland)] <<
std::endl;
```

You should obtain the following results:

```
Feasible Solution
Belgium:     green
Denmark:     blue
France:      yellow
Germany:     white
Luxembourg:  blue
Netherlands: blue
Switzerland: blue
```

Complete program

You can view the complete map coloring program online in the file `YourCPHome/examples/src/cpp/color.cpp`.

Results

```
Feasible Solution
Belgium:     yellow
Denmark:     blue
France:      blue
Germany:     white
Luxembourg:  green
Netherlands: blue
```

Using arrays and objectives: warehouse location

This section describes how to model and solve a problem with arrays and an objective.

In this section

Overview
Describes how to model and solve a problem with arrays and an objective.

Describe
Describes the first stage in finding a solution to a warehouse location problem.

Model
Describes the second stage in finding a solution to a warehouse location problem.

Solve
Describes the third stage in finding a solution to a warehouse location problem.

Review exercises
Includes the review exercises and suggested answers.

Complete program
Lists the location of the complete warehouse location program and the results.

Overview

In this lesson, you will learn how to:

- use the template `IloArray`;
- use the class `IloIntVarArray` to declare arrays of decision variables;
- use the functions `IloCount` and `IloScalProd`;
- use arithmetic and element expressions;
- use objectives.

You will learn to model and solve a logistics problem, a facility location problem. To find an optimal solution to this problem using IBM® ILOG® CP Optimizer, you will use the three-stage method: describe, model and solve.

Describe

In this lesson, you will solve a facility location problem for a company which has eight stores. Each store must be supplied by one supplier warehouse. The company has five possible locations where it can open a supplier warehouse: Bonn, Bordeaux, London, Paris and Rome. The warehouse locations have varied supply capacities. A warehouse in Bordeaux or Rome could supply only one store. A warehouse in London could supply two stores; a warehouse in Bonn could supply three stores; and a warehouse in Paris could supply four stores. The supply costs vary for each store and depend on which warehouse is the supplier. For example, a store that is located in Paris would have low supply costs if it were supplied by a warehouse also in Paris. That same store would have much higher supply costs if it were supplied by the other warehouses. The cost of opening a warehouse depends on its location.

The table *Supply costs for stores* gives the cost of opening each warehouse and of supplying each existing store from each of the potential supplier warehouse sites.

Supply costs for stores

	Bonn	Bordeaux	London	Paris	Rome
cost	480	200	320	340	300
store 0	24	74	31	51	84
store 1	57	54	86	61	68
store 2	57	67	29	91	71
store 3	54	54	65	82	94
store 4	98	81	16	61	27
store 5	13	92	34	94	87
store 6	54	72	41	12	78
store 7	54	64	65	89	89

The problem is to find the most cost-effective solution to this problem, while ensuring that each store is supplied by a warehouse. An *objective* is used to express the cost of each potential solution

Note: **Objective**

An *objective* expresses the cost of possible solutions. The optimal solution to an optimization problem is the feasible solution that, depending on the problem type, minimizes or maximizes the cost.

Step 1: Describe the problem

The first step in modeling and solving a problem is to write a natural language description of the problem, identifying the decision variables and the constraints on these variables.

Write a natural language description of this problem. Answer these questions:

- What is the known information in this problem?
- What are the decision variables or unknowns in this problem?
- What are the constraints on these variables?
- What is the objective?

Discussion

What is the known information in this problem?

- There are eight stores.
- There are five potential supplier warehouse sites.
- The cost of opening each warehouse is known.
- The cost of supplying each existing store from each of the potential warehouse sites is known.

What are the decision variables or the unknowns in this problem?

- The unknown is which warehouse should supply each store. In other words, there is a decision variable for each store which indicates which warehouse will serve it. Each of these variables has a domain of [0..4]. Each element of the domain represents one of the potential warehouses.
- In addition, it is unknown whether or not a particular warehouse will be opened. There is a decision variable for each warehouse which indicates whether it is to be opened. Each of these variables has a domain of [0..1]. A value of 1 indicates that the warehouse is to be opened; conversely, a value of 0 indicates that it is not to be opened.

What are the constraints on these variables?

- In this case, only one warehouse can supply a given store and a warehouse must be open in order to serve any store.
- The number of stores supplied by a warehouse must not exceed the capacity of that warehouse.

What is the objective?

- The objective is to find the most cost-effective solution, one that takes into account the costs of supplying individual stores and the costs of opening warehouses.

Model

After you have written a description of your problem, you can use IBM® ILOG® Concert Technology classes to model it. After you create a model of your problem, you can use IBM ILOG CP Optimizer classes and functions to search for a solution.

Step 2: Open the example file

Open the example file `YourCPHome/examples/tutorial/cpp/facility_partial.cpp` in your development environment.

In this lesson, you include the header file `<ilcp/cp.h>`. To catch exceptions that may be thrown, you use a `try/catch` block. The code to declare an environment and a model is provided for you as is the code to declare the standard programming variables `i` and `j` for use in loops:

```
int main(int argc, const char* argv[]){
  IloEnv env;
  try{
    IloModel model(env);
    IloInt i, j;
```

In this lesson, you will input data from the file `YourCPHome/examples/data/facility.dat`. The code for opening the data file is provided for you:

```
    const char* filename = "../../../examples/data/facility.dat";
    if (argc > 1)
      filename = argv[1];
    std::ifstream file(filename);
    if (!file){
      env.out() << "usage: " << argv[0] << " <file>" << std::endl;
      throw FileError();
    }
```

The data from the file `YourCPHome/examples/data/facility.dat` is:

```
[ 3, 1, 2, 4, 1]
[ 480, 200, 320, 340, 300]
[[ 24, 74, 31, 51, 84],
 [ 57, 54, 86, 61, 68],
 [ 57, 67, 29, 91, 71],
 [ 54, 54, 65, 82, 94],
 [ 98, 81, 16, 61, 27],
 [ 13, 92, 34, 94, 87],
 [ 54, 72, 41, 12, 78],
 [ 54, 64, 65, 89, 89]]
```

The first line of data is the capacity of each potential supplier warehouse: Bonn 3, Bordeaux 1, London 2, Paris 4 and Rome 1. The second line of data is the fixed cost associated with opening each warehouse. The last part of the data is a matrix representing the relative cost of supplying each existing store from each of the potential warehouse sites. Other than the capacity data, this is the information presented in the table *Supply costs for stores*.

You create standard programming variables to represent the number of stores, `nbStores`, and the number of supplier warehouses, `nbLocations`. You model the capacities and fixed costs of the potential warehouses as arrays of integer values.

Note: **Array of integer values**

Arrays of integer values are represented by the class `IloIntArray` in the C++ API of Concert Technology. These arrays are extensible.

When you use an array, you can access a value in that array by its index, and the `operator[]` is overloaded for this purpose.

In the C# and Java™ APIs, the native arrays are used.

The constructor of the class `IloIntArray` takes an environment as its first argument. The second argument is the number of integer values in the array, which is extensible. The elements of the new array take the values that are passed as the remaining arguments. Here is one constructor:

```
IloIntArray(const IloEnv env, IloInt n, const IloInt v0, ...);
```

In this lesson however, you create empty arrays which will be filled directly from the data file. To do this, you pass solely the first argument, the environment, to the constructor.

You model the matrix of relative supply costs as an array of arrays. To create this cost matrix, you use the Concert Technology template `IloArray`.

Note: **Extensible array**

In the C++ API, Concert Technology provides the template class `IloArray` which makes it easy for you to create classes of arrays for elements of any given class. In other words, you can use this template to create arrays of Concert Technology objects; you can also use this template to create arrays of arrays (that is, multidimensional arrays).

When you use an array, you can access a value in that array by its index, and the `operator[]` is overloaded for this purpose.

The classes you create in this way consist of extensible arrays. That is, you can add elements to the array as needed.

To model the supply costs, you create an `IloArray` in which each of the elements is an `IloIntArray`.

Step 3: Model the data

Add the following code after the comment `//Model the data`

```
IloIntArray capacity(env), fixedCost(env);
IloArray<IloIntArray> cost(env);
IloInt nbLocations;
IloInt nbStores;
```

Now, you input the data from the data file. The overloaded C++ `operator>>` directs input from an input stream.

Step 4: Input the data

Add the following code after the comment `//Input the data`

```
file >> capacity >> fixedCost >> cost;
```

The number of stores can be deduced from the size of the `cost` matrix. The number of supplier warehouses can be deduced from the size of the `capacity` array. Calculating these values from the data read from the file allows you to easily extend the example by using another data file. The code for determining the values of `nbLocations` and `nbStores` is provided for you:

```
nbLocations = capacity.getSize();
nbStores = cost.getSize();
```

Next, you represent the first set of unknowns in this problem, the warehouse that will serve each store. IBM® ILOG® Concert Technology gives you the means to represent these unknowns as an array of constrained integer variables. You associate one decision variable in the array with each store.

Note: **Array of decision variables**

In the C++ API, arrays of constrained integer variables are represented by the class `IloIntVarArray` in Concert Technology.

When you use an array, you can access a decision variable in that array by its index, and the `operator[]` is overloaded for this purpose.

The constructor of the class `IloIntVarArray` takes an environment as its first argument. The second argument is the number of decision variables in the array. Arrays are extensible,

so you can later increase or reduce the number of variables in the array. The third and fourth arguments are the lower and upper bounds of the domain of possible values for each variable in the array. When you use an array, you can access a variable in that array by its index, and the operator`[]` is overloaded for this purpose. Here is a constructor:

```
IloIntVarArray (const IloEnv env,
                IloInt n,
                IloInt lb,
                IloInt ub);
```

The first array of decision variables, called `supplier`, represents which supplier warehouse should supply each store. This array has `nbStores` elements or, in this example, 10. The domain of possible values for each of these variables represent the supplier warehouses. In this example, a value of 0 represents Bonn, a value of 1 represents Bordeaux, a value of 2 represents London, a value of 3 represents Paris, and a value of 4 represents Rome.

For each store, the associated decision variable in the array `supplier` can be assigned one value in a solution, so each solution will satisfy the constraint that a store must be served by exactly one warehouse. The array of decision variables `supplier` will represent the solution to the problem, after it has been solved.

Step 5: Declare the supplier decision variables

Add the following code after the comment `//Declare the supplier decision variables`

```
   IloIntVarArray supplier(env, nbStores, 0, nbLocations - 1);
```

To represent whether a potential supplier warehouse site should be open or not, you declare another array of decision variables, `open`. It has `nbLocations` elements or, in this example, 5. There are two possible values for each variable, 0 and 1. The value of 1 indicates that the warehouse is open and 0 indicates that the warehouse is not open.

Step 6: Declare the warehouse open decision variables

Add the following code after the comment `//Declare the warehouse open decision variables`

```
   IloIntVarArray open(env, nbLocations, 0, 1);
```

Now you add the constraints. The first constraint states that the supplier warehouse used by a given store must be open. In other words, the warehouse must be open if a store is to be supplied by it. For example, if store 0 is supplied by the Rome warehouse, the constraint states that the variable of the `open` array that is associated with the Rome warehouse must be assigned the value 1. The Rome warehouse must be open.

This interdependence of the decision variable arrays `supplier` and `open` is modeled through the use of an *element expression*.

Note: **Element expression**

The construction `T[i] = v` (where `T` is an instance of `IloIntVarArray`, `i` is an instance of `IloIntVar` and `v` is an integer value) constrains the `i`-th element of `T` to be equal to `v`.

In the C# and Java™ APIs, the respective methods `CP.Element` and `IloCP.element` are used to create an element expression.

For each store `i`, the warehouse that supplies that store is represented by `supplier[i]`. To indicate that `supplier[i]` must be open, you constrain `open[supplier[i]]` to be assigned the value 1.

Step 7: Add the constraints on open warehouses

Add the following code after the comment `//Add the constraint on open warehouses`

```
for (i = 0; i < nbStores; i++)
  model.add(open[supplier[i]] == 1);
```

Next, you need to compute the number of stores being supplied by a particular warehouse and ensure that this value does not exceed the capacity of the warehouse. You can use the IBM® ILOG® Concert Technology function `IloCount` to represent the number of times a warehouse is assigned to be a supplier.

Note: **Counting expression**

Using the function `IloCount`, you can create an *expression* that represents the number of times a particular value is assigned to the decision variables in an array of constrained integer variables.

Expressions are discussed in further detail in the following step.

The function `IloCount` takes two arguments, an array of decision variables and an integer value, and returns an instance of `IloIntExprArg` that represents the number of times the value appears in the array. The class `IloIntExprArg` is used internally by IBM ILOG Concert Technology to build expressions; you should not use `IloIntExprArg` directly. Here is the version of `IloCount` that you will use:

```
IloIntExprArg IloCount(const IloIntVarArray vars, IloInt value);
```

For each warehouse, you count the number of times it appears in the array `supplier` using the function `IloCount`. You constrain this expression to be no greater than the warehouse capacity using `operator<=`.

Step 8: Add the constraints on warehouse capacity

Add the following code after the comment `//Add the capacity constraints`

```
for (j = 0; j < nbLocations; j++)
  model.add(IloCount(supplier, j)  <= capacity[j]);
```

Finally, you consider the objective. The objective in a constraint programming model is an *expression* that can be maximized or minimized. Expressions in IBM® ILOG® Concert Technology are generally represented by the classes `IloExpr` and its subclasses `IloIntExpr` and `IloNumExpr`.

Note: **Expression**

Values may be combined with decision variables and other expressions to form expressions. To combine values, variables and other expressions to form an expression, you can use, among other functions, the operators:

- addition (`+`),
- subtraction (`-`),
- multiplication (`*`),
- division (`/`),
- self-assigned addition (`+=`) and
- self-assigned subtraction (`-=`).

Many Concert Technology functions, such as `IloCount`, return expressions. During search, expressions have domains of possible values like decision variables. Unlike variables, these domains are not stored but instead calculated based on the basic elements of the expression.

To build the objective expression, you need to represent the sum of the fixed costs of all the warehouses that are to be opened. This cost can be modeled as the scalar product of the `fixedCost` array and the `open` array. To express the fixed cost expression, you use the Concert Technology function `IloScalProd`.

Note: **Scalar product expression**

Using the function `IloScalProd`, you can create an expression that represents the scalar product of two arrays.

The scalar product is also called the inner product or the weighted sum.

The function `IloScalProd` takes two arguments, both arrays, and returns an instance of `IloIntExprArg` that represents the scalar product of the two arrays. Although you should not use an `IloIntExprArg` directly, it can be cast to an `IloIntExpr`, which you should use instead. Here is the signature of the function `IloScalProd` that you will use:

```
IloIntExprArg IloScalProd(const IloIntVarArray vars,
                          const IloIntArray values);
```

You declare the integer expression `obj` to be the expression returned by the `IloScalProd` function called with the arguments `open` and `fixedCost`. To complete the objective expression, you must also add the supply costs to it. To express the cost of supplying each store from the warehouse selected to supply it, you use an element expression. For each store `i`, the cost of supplying the store from the assigned warehouse is `cost[i][supplier[i]]`. You can incrementally add these costs to the expression `obj` by using the self-assigned addition operator.

Step 9: Create the objective expression

Add the following code after the comment `//Build the objective expression`

```
  IloIntExpr obj = IloScalProd(open, fixedCost);
  for (i = 0; i < nbStores; i++)
    obj += cost[i][supplier[i]];
```

To add the objective to the model, you first use the function `IloMinimize`, which creates an instance of the class `IloObjective`. The function `IloMinimize` takes the environment as its first argument and an expression as its second argument. The last argument is an optional name used for debug and trace purposes. Here is signature of the `IloMinimize` function that you will use:

```
IloObjective IloMinimize(const IloEnv env,
                         const IloExpr expr,
                         const char* name=0);
```

You then use the member function `IloModel::add` to add the objective to the model. You must explicitly add an objective to the model or the CP Optimizer engine will not use it in the search for a solution. If an objective has been added to the model, the optimizer will search for an optimal solution which minimizes the value of the objective.

Step 10: Add the objective to the model

Add the following code after the comment `//Add the objective to the model`

```
  model.add(IloMinimize(env, obj));
```

Solve

Solving the problem consists of finding values for all of the decision variables in such a way that the values simultaneously satisfy the constraints and minimize the objective representing the cost of the solution.

Step 11: Create an instance of IloCP

Add the following code after the comment `//Create an instance of IloCP`

```
IloCP cp(model);
```

You now use the member function `IloCP::solve`, which solves a problem by using constructive search and constraint propagation.

Step 12: Search for a solution

Add the following code after the comment `//Search for a solution`

```
if (cp.solve())
```

The member function `IloCP::solve` will return a value of `IloTrue` if the optimizer is able to find an optimal solution. To display the solution, you use the member functions and streams `IloAlgorithm::getStatus`, `IloAlgortihm::getValue` and `IloAlgorithm::out`. The code for displaying the solution is provided for you:

```
cp.out() << std::endl << "Optimal value: " << cp.getValue(obj) << std::endl;

 for (j = 0; j < nbLocations; j++){
   if (cp.getValue(open[j]) == 1){
     cp.out() << "Facility " << j << " is open, it serves stores ";
     for (i = 0; i < nbStores; i++){
       if (cp.getValue(supplier[i]) == j)
         cp.out() << i << " ";
     }
     cp.out() << std::endl;
   }
 }
```

Step 13: Compile and run the program

Compile and run the program. You should get the following results:

```
Optimal value: 1383
Facility 0 is open, it serves stores 2 5 7
Facility 1 is open, it serves stores 3
Facility 3 is open, it serves stores 0 1 4 6
```

As you can see, the optimal solution uses three warehouse sites: Bonn, Bordeaux, Paris. All stores are supplied by a supplier warehouse and the total cost is 1383.

The complete program can be viewed online in the `YourCPHome/examples/src/cpp/facility.cpp` file.

Review exercises

Includes the review exercises and suggested answers.

In this section

Exercises
Lists the review exercises.

Suggested answers
Provides the suggested answers to the review exercises.

Exercises

For answers, see *Suggested answers*.

1. What is an element expression?
2. What is an objective?
3. Modify the program in this lesson to add the constraint that no supplier can serve both stores 2 and 7.
4. Modify the program in this lesson to add the constraint that at most one of warehouses 0 (Bonn) and 3 (Paris) can be open.

Suggested answers

Exercise 1

What is an element expression?

Suggested answer

An element expression is an expression created by using a constrained integer decision variable as an index of an array.

Exercise 2

What is an objective?

Suggested answer

An objective is an expression which can be maximized or minimized. The cost of different solutions can be expressed in an objective.

Exercise 3

Modify the program in this lesson to add the constraint that no supplier can serve both stores 2 and 7.

Suggested answer

The code that has changed from `facility.cpp` follows. You can view the complete program online in the file `YourCPHome/examples/src/cpp/facility_ex3.cpp`.

This additional constraint is added to the model:

```
model.add(supplier[2] != supplier[7]);
```

You should obtain the following results:

```
Optimal value: 1390
Facility 0 is open, it serves stores 0 5 7
Facility 1 is open, it serves stores 3
Facility 3 is open, it serves stores 1 2 4 6
```

Exercise 4

Modify the program in this lesson to add the constraint that at most one of warehouses 0 (Bonn) and 3 (Paris) can be open.

Suggested answer

The code that has changed from `facility.cpp` follows. You can view the complete program online in the file `YourCPHome/examples/src/cpp/facility_ex4.cpp`.

This additional constraint is added to the model:

```
   model.add(open[0] + open[3] <= 1);
```

You should obtain the following results:

```
Optimal value: 1517
Facility 1 is open, it serves stores 3
Facility 2 is open, it serves stores 2 5
Facility 3 is open, it serves stores 0 1 6 7
Facility 4 is open, it serves stores 4
```

Complete program

You can view the complete facility location program online in the file `YourCPHome/examples/src/cpp/facility.cpp`.

Results

Here are the results:

```
Optimal value: 1383
Facility 0 is open, it serves stores 2 5 7
Facility 1 is open, it serves stores 3
Facility 3 is open, it serves stores 0 1 4 6
```

Using specialized constraints and tuples: scheduling teams

This section describes how to model and solve a problem using specialized constraints and tuples.

In this section

Overview
Describes how to model and solve a problem using specialized constraints and tuples.

Describe
Describes the first stage in a sports league scheduling problem.

Model
Describes the second stage in a sports league scheduling problem.

Solve
Describes the third stage in a sports league scheduling problem.

Review exercises
Includes the review exercises and suggested answers.

Complete program
Lists the location of the complete sports scheduling program and the results.

Overview

In this lesson, you will learn how to:

- use constraints defined from allowed tuples;
- use the specialized constraints `IloInverse` and `IloAllDiff`;
- use the functions `IloDiv` and `IloAbs`;
- use logical constraints;
- use additional constraints to improve the solving performance;
- use search phases to tune the search strategy.

You will learn how to model and solve an assignment problem regarding the scheduling of matches for a league of sports teams. To find an optimal solution to the problem using IBM® ILOG® CP Optimizer, you will use the three-stage method: describe, model and solve.

Describe

In this lesson, you will solve a sports league scheduling problem. The league has 10 teams that play games over a season of 18 weeks. Each team has a home arena and plays every other team in the league twice during the season, once in its home arena and once in the opposing team's home arena. For each of these games, the team playing at its home arena is referred to as the home team; the team playing at the opponent's arena is called the away team. There are 90 games altogether.

Each of the 18 weeks in the season has five identical slots to which games can be assigned. Each team plays once a week. For each pair of teams, these two teams are opponents twice in a season; these two games must be scheduled in different halves of the season. Moreover, these two games must be scheduled at least six weeks apart. A team must play at home either the first or last week but not both.

A break is a sequence of consecutive weeks in which a team plays its games either all at home or all away. No team can have a break longer than two weeks. The objective in this problem is to minimize the total number of breaks the teams play.

Step 1: Describe the problem

The first step in modeling and solving a problem is to write a natural language description of the problem, identifying the variables and the constraints on these variables.

Write a natural language description of this problem. Answer these questions:

- What is the known information in this problem?
- What are the variables or unknowns in this problem?
- What are the constraints on these variables?
- What is the objective?

Discussion

What is the known information in this problem?

- There are 10 teams.
- These teams play against one another during a season of 18 weeks. There are 5 slots for games each week.
- Each team must play every other team exactly twice, once in each team's home arena. Each of these 90 games has a unique identifier in the range 0..89.

What are the variables or unknowns in this problem?

- The unknown information is which of the 90 games is played in each of the 90 slots. In other words, there is a decision variable for each slot which indicates which game will be assigned to it. Each of these constrained variables has a domain of [0..89]; each integer of these domains is a game identifier.

What are the constraints on these variables?

- Each team must play exactly once each week.

- Each game must be played exactly once.
- For each pair of teams, the two games in which the teams are opponents must be scheduled in different halves of the season. Moreover, these games must be at least six weeks apart.
- No team may have a break longer than two weeks.
- Each team must play exactly one of its first and last games at home.

What is the objective?

- The objective is to minimize the total number of breaks that are scheduled.

Model

After you have written a description of your problem, you can use IBM® ILOG® Concert Technology classes to model it.

Step 2: Open the example file

Open the example file `YourCPHome/examples/tutorial/cpp/sports_partial.cpp` in your development environment.

In this lesson, you include the header file `<ilcp/cp.h>`. To catch exceptions that may be thrown, you use a `try/catch` block. The code for declaring an environment and a model, calculating the unique game identifiers and for printing out the solution found by the CP Optimizer engine is provided for you.

First, you represent the data of the program. The number of teams, `n`, is set to a default of 10 in this example, but can be changed with an input argument. (For this problem, the number of teams must always be even. If the number of teams is not even, it is increased by 1.) The code for processing the input argument is provided for you:

```
IloInt n = 10;
if (argc > 1)
  n = atoi(argv[1]);
if ((n % 2) == 1)
  n++;
env.out() << "Finding schedule for " << n << " teams" << std::endl;
```

The number of weeks, `nbWeeks`, is `2*(n-1)`, or 18 in this example. The number of game slots per week, `nbGamesPerWeek`, is `nbTeams/2`, or 5 in this example. The number of game slots, `nbGames`, is equal to the number of weeks times the number of game slots per week, or 90 in this example. The number of games to be scheduled is equal to the number of game slots available.

Step 3: Calculate the data

Add the following code after the comment `//Calculate the data`

```
IloInt nbWeeks = 2 * (n - 1);
IloInt nbGamesPerWeek = n / 2;
IloInt nbGames = n * (n - 1);
```

Each week has five slots; each slot must have a game assigned to it. The unknowns in this problem are which game is assigned to which slot. To represent these unknowns, you declare a matrix of decision variables, `games`, which is indexed on week and slot. To create the matrix of variables, you create an array in which each element is an array of constrained integer variables. The domain of values for each variable in `games` is the set of unique game identifiers; in this example the domain of each variable is [0..89]. The matrix of variables

games will represent the solution to the problem after a solution has been found by the CP Optimizer engine.

To help you model the constraints that involve the home and away teams, it will be useful to be able to easily determine which team is the home team and which team is the away team for each slot. You declare auxiliary variables that are not directly a part of the solution, but make it easier to model the breaks and other constraints. You declare matrices of decision variables which represent the teams that will play at home and away in each slot, called home and away, respectively. These matrices are indexed on week and slot in the same manner as the matrix games. The domain of the variables in each of these matrices are the teams, represented by integers in the range [0..9].

Step 4: Declare the game, home team and away team variables

Add the following code after the comment `//Declare the game, home team and away team variables`

```
IloIntVarArray2 games(env, nbWeeks);
IloIntVarArray2 home(env, nbWeeks);
IloIntVarArray2 away(env, nbWeeks);
for (IloInt i = 0; i < nbWeeks; i++) {
  home[i]  = IloIntVarArray(env, nbGamesPerWeek, 0, n - 1);
  away[i]  = IloIntVarArray(env, nbGamesPerWeek, 0, n - 1);
  games[i] = IloIntVarArray(env, nbGamesPerWeek, 0, nbGames - 1);
}
```

For each slot, you need to link the variables in the arrays game, home and away. In other words, you constrain that only certain configurations of home team, away team and game identifiers are allowed. These allowed configurations can be modeled as a set of integer tuples.

Note: **Set of tuples**

An *integer tuple* is an ordered set of values represented by an array. A set of integer tuples in a model is represented by an instance of `IloIntTupleSet` in the C++ and Java™ APIs and by an instance of `IIntTupleSet` in the C# API.

The number of values in a tuple is known as the arity of the tuple.

The constructor of `IloIntTupleSet` takes the environment as its first argument, and the arity as its second argument. Here is a constructor for `IloIntTupleSet`:

```
IloIntTupleSet(IloEnv env, const int arity);
```

You use the member function `IloIntTupleSet::add` to add tuples to the set.

In this example, the game identifier indicates a particular pairing of home team and away team. Using the `Game` function, you see that the game in which Team 1 plays at home against

Team 2 is Game 10. This combination is a tuple. You would write it as (1, 2, 10). The number of values in a tuple is known as the arity of the tuple. The tuple (1, 2, 10) has an arity of 3. You calculate all allowed combinations of home team, away team and game identifier using the `Game` function.

Step 5: Calculate the allowed tuples

Add the following code after the comment `//Calculate the allowed tuples`

```
IloIntTupleSet gha(env, 3);
IloIntArray tuple(env, 3);
for (IloInt i = 0; i < n; i++) {
  tuple[0] = i;
  for (IloInt j = 0; j < n; j++) {
    if (i != j) {
      tuple[1] = j;
      tuple[2] = Game(i, j, n);
      gha.add(tuple);
    }
  }
}
```

The complete set of possible combinations of home team, away team and game identifier are enumerated in the set of tuples, `gha`. To constrain the values assigned to a given slot of the matrices of `home`, `away` and `games` variables to be allowed combinations, you use the IBM ILOG Concert Technology constraint `IloAllowedAssignments`.

Note: **Compatibility constraint**

The function `IloAllowedAssignments` takes an array of decision variables and a set of tuples. It returns a constraint that specifies that the only possible combinations of allowed values for the variables are the tuples in the tupleset.

The function `IloAllowedAssignments` returns an instance of an `IloConstraint`. The first argument passed to this function is the environment. The second argument is the array of decision variables. The third argument is the tupleset which enumerates the allowed combinations of values for the array of variables. The arity of the tuples must be the same as the length of the array of the decision variables. Here is the function signature you will use:

```
IloConstraint IloAllowedAssignments(const IloEnv env,
                                    const IloIntVarArray vars,
                                    const IloIntTupleSet set);
```

For each slot, you build a small temporary array of decision variables from variables you have already created. This new array has the `home`, `away` and `games` variables associated with the given slot. Note that you are not creating new decision variables but creating an

alternate manner of referencing the previously created variables. Adding the compatibility constraint that the allowed assignments for these variables must be in the set of tuples, `gha`, ensures that these three matrices of variables are properly linked.

Step 6: Add the constraint on allowed combinations

Add the following code after the comment `//Add the constraint on allowed combinations`

```
for (IloInt i = 0; i < nbWeeks; i++) {
  for (IloInt j = 0; j < nbGamesPerWeek; j++) {
    IloIntVarArray vars(env);
    vars.add(home[i][j]);
    vars.add(away[i][j]);
    vars.add(games[i][j]);
    model.add(IloAllowedAssignments(env, vars, gha));
  }
}
```

Now that the auxiliary variables are properly linked to the `games` variables, you begin to model the constraints as outlined in the description of the problem. The first set of constraints state that each team can play in exactly one slot a week. For each week, no team can appear twice in the set of variables representing the teams playing at home and away for that week. In other words, the home and away variables associated with a given week must all be assigned different values in any solution. For each week, you create an array with all of the home and away variables for that week. To state that the value assigned to each decision variable in an array must be different from that of every other variable in that array, you use the IBM ILOG Concert Technology predefined constraint `IloAllDiff`.

Note: **All different constraint**

Using specialized constraints, such as `IloAllDiff`, makes modeling simpler and the solving more efficient.

The single constraint `IloAllDiff` on `n` variables is logically equivalent to `n(n-1)/2` instances of the “not equal” constraint, `!=`, between each pair of decision variables in the array.

The class `IloAllDiff` is a subclass of the class `IloConstraint`. The constructor of `IloAllDiff` takes the environment as its first argument. The second argument is the array of variables. The third argument is an optional name used for debug and trace purposes. Here is the function signature you will use:

```
IloAllDiff(const IloEnv env,
           const IloIntVarArray vars = 0,
           const char* name = 0);
```

To simplify creating the single array containing the home and away variables for a given week, you use the `add` method of `IloIntVarArray` whose single argument is an `IloIntVarArray`. This method appends the argument array to the invoking array.

Step 7: Add the alldiff constraint

Add the following code after the comment `//Add the alldiff constraint`

```
for (IloInt i = 0; i < nbWeeks; i++) {
  IloIntVarArray teamsThisWeek(env);
  teamsThisWeek.add(home[i]);
  teamsThisWeek.add(away[i]);
  model.add(IloAllDiff(env, teamsThisWeek));
}
```

Next, you add the constraints that the two games between a pair of teams must be in different halves of the season and also must be at least six weeks apart (if there are fewer than 6 teams, this minimum distance is reduced). To simplify modeling this constraint, you create auxiliary variables to represent the week of the season in which each game will be played.

You create the array `weekOfGame` which is indexed on game identifiers. Each element of this array is a decision variable with a domain of the possible weeks, [0..17]. The value of each variable in this array represents the week in which that game will be played.

To ensure that the `weekOfGame` variables are assigned the appropriate values, you need to be able to determine the week of each game. Assuming that each game is played exactly once, the week that game `g` is played is `w`, where there is an `i` such that `game[w][i]` is assigned the value `g`. To model this relationship, you use two steps. First, you determine which slot a game has been assigned to, then you determine the week of that slot.

To determine to which slot a game has been assigned, you create two arrays of decision variables, each of length `nbGames`. One is a "flattened" version of the matrix `games`, called `allGames`. The variables in `allGames` are not new variables, but provide an alternate view on the original variables in `games`. The other array, `allSlots`, is indexed by the games. The domain of each variable in `allSlots` is the set of values of the indices of `allGames`, [0..89]. Since `allSlots[i] == j` if and only if `allGames[j] == i`, the arrays have an inverse relationship. This relationship can be modeled using the IBM ILOG Concert Technology constraint `IloInverse`.

Note: **Inverse constraint**

Using specialized constraints, such as `IloInverse`, makes modeling simpler and the solving more efficient.

The single constraint `IloInverse` ensures that for two arrays of decision variables `x` and `y` of equal size

- for all `i` in the interval [0..`n`-1], `y[x[i]] == i`;
- for all `j` in the interval [0..`n`-1], `x[y[j]] == j`.

The class `IloInverse` is a subclass of `IloConstraint`. The constructor of `IloInverse` takes the environment as its first argument. The second argument is one array of decision variables. The third argument is the other array of variables. Here is the function signature you will use:

```
IloInverse(const IloEnv env,
           const IloIntVarArray f,
           const IloIntVarArray invf);
```

As the arrays are of the same length, this constraint also has the virtue of ensuring that in any solution, the values assigned to all the variables within each array will be unique.

Step 8: Add the inverse constraint

Add the following code after the comment `//Add the inverse constraint`

```
  IloIntVarArray weekOfGame(env, nbGames, 0, nbWeeks - 1);
  IloIntVarArray allGames(env);
  IloIntVarArray allSlots(env, nbGames, 0, nbGames - 1);
  for (IloInt i = 0; i < nbWeeks; i++)
    allGames.add(games[i]);
  model.add(IloInverse(env, allGames, allSlots));
```

After you have established the connection between the arrays `allSlots` and `allGames`, you write a constraint to represent to which week a game will be assigned. Using the array `allSlots`, you can represent to which slot a game will be assigned. To determine to which week a slot belongs, you can use integer division. For instance, slot `i` is in the week `i div nGamesPerWeek` where `div` represents the integer division operator. IBM ILOG Concert Technology provides a function `IloDiv` for use in integer expressions.

Note: **Integer division expression**

Using the function `IloDiv`, you can create an expression that represents integer division.

The function `IloDiv` takes two arguments, an integer expression (or variable) and an integer value, and returns an instance of `IloIntExprArg`. The class `IloIntExprArg` is used internally by IBM ILOG Concert Technology to build expressions. You should not use `IloIntExprArg` directly. Here is the version of `IloDiv` that you will use:

```
IloIntExprArg IloDiv(const IloIntExprArg x, IloInt y);
```

Step 9: Add the week of game

Add the following code after the comment `//Add the week of game constraint`

```
for (IloInt i = 0; i < nbGames; i++)
  model.add(weekOfGame[i] == IloDiv(allSlots[i], nbGamesPerWeek));
```

Now that you have a representation of the week in which each game will be played, you can complete the second step in writing constraints to model the limitations on season half and minimum length of time between the games. You determine the identifiers of the two games played between each pair of teams and then constrain that if one game is scheduled in the last half of the season, then the other game must be scheduled in the first half of the season and that if one game is not scheduled in the last half of the season, then the other game must not be scheduled in the first half of the season. This type of conditional constraint is called a *logical constraint*.

Note: **Logical constraint**

A logical constraint is created by combining constraints or placing constraints on other constraints. Logical constraints are based on the idea that constraints have value. IBM ILOG CP Optimizer handles constraints not only as if they have a Boolean value, such as true or false, but effectively as if the value is 0 or 1. This allows you to combine constraints into expressions or impose constraints on constraints.

Constraints can be combined using arithmetic operators. With the C++ API, you can also use the following logical operators to combine constraints:

- not (`!`),
- and (`&&`),
- or (`||`),
- equivalence (`==`),
- exclusive or (`!=`) (This overloaded C++ operator constrains its two arguments to be unequal--different from each other.) and
- implication (`<=`) (This overloaded C++ operator is also the “less than or equal to” operator. When its arguments are constraints, this operator functions as the implication operator. The argument on the left side of the operator implies the argument on the right side of the operator.).

Step 10: Create the different halves constraint

Add the following code after the comment `//Create the different halves constraint`

```
IloInt mid = nbWeeks / 2;
IloInt overlap = 0;
if (n >= 6)
  overlap = IloMin(n / 2, 6);
for (IloInt i = 0; i < n; i++) {
  for (IloInt j = i + 1; j < n; j++) {
    IloInt g1 = Game(i, j, n);
    IloInt g2 = Game(j, i, n);
    model.add((weekOfGame[g1] >= mid) == (weekOfGame[g2] < mid));
```

To constrain that six weeks is the minimum length of time between the two games played between a given pair of teams, you could write a logical constraint using a disjunction. However, IBM ILOG Concert Technology provides a function `IloAbs`, which not only eases modeling but may also improve performance in the search, since it provides more information.

Note: **Absolute value expression**

Using the function `IloAbs`, you can create an expression that represents the absolute value of an expression.

Step 11: Create the distance constraint

Add the following code after the comment `//Create the distance constraint`

```
    if (overlap != 0)
      model.add(IloAbs(weekOfGame[g1] - weekOfGame[g2]) >= overlap);
  }
}
```

You next write the constraint on the maximum size of breaks. No team can have a break of length three or greater. In other words, in any sequence of three weeks, a team must play at least one game at home and no more than two games at home. In order to count the number of games at home in a sequence of three weeks, you create a matrix of auxiliary variables indexed on team and week called `playHome`. The domain of each variable in this matrix is [0..1]. A value of 0 indicates the team plays away that week and a value of 1 indicates that the team plays at home that week.

To constrain each variable in `playHome` to be assigned the appropriate value, you use the function `IloCount`. You constrain the value of the element of `playHome` for that week and team by counting the number of times that a particular team plays at home during a particular week (it must be 0 or 1).

For each team, you represent each sequence of three games with a subarray of the array `playHome`. Using this subarray and the predefined IBM ILOG Concert Technology function `IloSum`, you can represent the number of time the team plays at home during the three weeks.

Note: **Addition expression**

Using the function `IloSum`, you can create an expression that represents the sum of the decision variables in the array passed as an argument to the function.

For each team, you constrain the sum of each sequence of three `playHome` variables for each team to take the value 1 or 2. To do this, you use a *range constraint*.

Note: **Range constraint**

A *range constraint* is useful when the value of a decision variable or expression must fall between to values. In the C++ API, instead of writing two separate constraints, one can be written in the form

`lb <= exp <= ub`, where `lb` and `ub` are the bounds and `exp` is the expression.

Step 12: Add the max break length constraint

Add the following code after the comment `//Add max break length constraints`

```
IloIntVarArray2 playHome(env, n);
for (IloInt i = 0; i < n; i++) {
  playHome[i] = IloIntVarArray(env, nbWeeks, 0, 1);
  for (IloInt j = 0; j < nbWeeks; j++)
    model.add(playHome[i][j] == IloCount(home[j], i));
  for (IloInt j = 0; j < nbWeeks -3; j++) {
    IloIntVarArray window(env);
    for (IloInt k = j; k < j + 3; k++)
      window.add(playHome[i][k]);
    model.add(1 <= IloSum(window) <= 2);
  }
}
```

The final modeling constraint you add states that the team must play either its first game or last game at home, but not both. For each team, you state that the value of its `playHome` decision variable for the first week must not equal the value of its `playHome` variable for the last week.

Step 13: Add the constraint on first and last weeks

Add the following code after the comment `//Add the constraint on first and last weeks`

```
for (IloInt i = 0; i < n; i++)
```

```
    model.add(playHome[i][0] != playHome[i][nbWeeks-1]);
```

To complete the model, you add the objective. The objective is constructed similarly to the constraint on break length. Since each break has a length of at most two, you know that there are at most `nbWeeks/2` breaks for each team. For each team, you count the number of breaks, which is now easily represented as two consecutive weeks for which the `playHome` variables are assigned the same value, using an expression comprised of constraints using a self-assigned addition operator. The variable `breaks` is set to be the total sum of all of the breaks for all the teams. The objective is to minimize the value of `breaks`.

Step 14: Add the objective

Add the following code after the comment `//Add the objective`

```
IloIntVarArray teamBreaks(env, n, 0, nbWeeks / 2);
for (IloInt i = 0; i < n; i++) {
  IloIntExpr nbreaks(env);
  for (IloInt j = 1; j < nbWeeks; j++)
    nbreaks += (playHome[i][j-1] == playHome[i][j]);
  model.add(teamBreaks[i] == nbreaks);
}
IloIntVar breaks(env, n - 2, n * (nbWeeks / 2));
model.add(breaks == IloSum(teamBreaks));
model.add(IloMinimize(env, breaks));
```

While the model has now been fully expressed, there are additional constraints that can be introduced to help improve the search. The first set of additional constraints are called *surrogate constraints*. The second type of additional constraints are constraints used to *reduce symmetry*.

Note: **Surrogate constraint**

A *surrogate constraint* makes explicit a property that satisfies a solution implicitly. In fact, in some disciplines, surrogate constraints are known as implicit constraints for that reason. Such a constraint should not change the nature of the solution, but its propagation should delimit the general shape of the solution more quickly.

In any case where an implicit property makes good sense, or derives from experience, or satisfies formal computations, its explicit implementation as a surrogate constraint can be beneficial.

Since constraint propagation decreases the search space by reducing the domains of variables, it is obviously important to express all necessary constraints. In some cases, it is even a good idea to introduce surrogate constraints to reduce the size of the search space by supplementary propagation.

Processing supplementary constraints inevitably slows down execution. However, this slowing down may be negligible in certain problems when it is compared with the efficiency gained from reducing the size of the search space.

It is known that a team must play half of its games at home. This is already implicitly expressed in the model within the set of tuples, so you are certain every solution will satisfy this. However, adding an additional constraint on the `playHome` variables will improve the constraint propagation.

Step 15: Add the surrogate constraints

Add the following code after the comment `//Add surrogate constraints`

```
for (IloInt i = 0; i < n; i++)
  model.add(IloSum(playHome[i]) == nbWeeks / 2);
```

Since only breaks of size two are allowed, each team must have an even number of breaks. This can be constrained using the IBM ILOG Concert Technology modulus operator, `operator%`.

Step 16: Add more surrogate constraints

Add the following code after the comment `//Add more surrogate constraints`

```
for (IloInt i = 0; i < n; i++)
  model.add((teamBreaks[i] % 2) == 0);
```

To help the search perform efficiently, it is also important to reduce symmetry.

Note: **Reduce symmetry**

he apparent complexity of a problem can often be reduced to a much smaller practical complexity by detecting intrinsic symmetries. One way to *reduce symmetry* in a problem is to introduce order.

There is no need to examine all the possible solutions when two or more constrained variables satisfy all of the following conditions:

- the initial domains of these constrained variables are identical;
- these variables are subject to the same constraints;
- the variables can be permuted without changing the statement of the problem.

By introducing order among these variables so that only one of these permutations is found, you minimize the size of the search space.

Since the teams are interchangeable, you can fix the games scheduled for the first week. This constraint removes only symmetric solutions; it does not eliminate any “real” solutions.

Step 17: Add constraints to reduce symmetry

Add the following code after the comment `//Add constraints to fix first week`

```
for (IloInt i = 0; i < nbGamesPerWeek; i++) {
  model.add(home[0][i] == i * 2);
  model.add(away[0][i] == i * 2 + 1);
}
```

Since the slots for a given week are identical with no distinguishing characteristics, you reduce symmetry by forcing an order on which games are assigned to which slots within a week.

Step 18: Add more constraints to reduce symmetry

Add the following code after the comment `//Add the slot order constraint`

```
for (IloInt i = 0; i < nbWeeks; i++)
  for (IloInt j = 1; j < nbGamesPerWeek; j++)
    model.add(games[i][j] > games[i][j-1]);
```

Solve

Solving the problem consists of finding values for all of the decision variables in such a way that the values simultaneously satisfy the constraints and minimize the objective representing the cost of the solution. To solve the problem expressed in the model, you create an instance of the class `IloCP`.

Step 19: Create an instance of IloCP

Add the following code after the comment `//Create an instance of IloCP`

```
IloCP cp(model);
```

The search for an optimal solution in this problem could potentiality take a long time, so you place a *time limit* on the solve process. The search will stop when the time limit is reached, even if optimality of the current best solution is not guaranteed. You also add the code to change the display frequency of the progress log of the search. Both of these are done by setting parameters on the optimizer.

Step 20: Add the time limit

Add the following code after the comment `//Add the time limit`

```
cp.setParameter(IloCP::TimeLimit, 60);
cp.setParameter(IloCP::LogPeriod, 10000);
```

Since you know information about the structure of this problem, you can tune the optimizer to help it perform better. While modeling the problem, you have introduced quite a few auxiliary variables. However, the array `allGames` contains the solution, so it is best to assign its values first. In fact, after all the decision variables in the array `allGames` have been assigned values, the values for all of the other variables in the model will have been determined through constraint propagation. A search strategy that chooses the variable in this array which currently has the smallest index and assigns it a random value works well in this problem.

To tune the optimizer, you first indicate that the search strategy should select from the variables of `allGames` the variable with the smallest index that has not been assigned a value. To do this, you pass the instance of the class `IloIntVarEval` that is returned by the function `IloVarIndex` to the selector `IloSelectSamllest` and add this to a variable selector array.

To indicate that you wish the search strategy to try to assign a random value to the variable that has been chosen, you pass the selector returned by the `IloSelectRandomValue` function to a value selector array.

Step 21: Create the search selectors

Add the following code after the comment `//Create the search selectors`

```
IloVarSelectorArray varSel(env);
varSel.add(IloSelectSmallest(IloVarIndex(env, allGames)));
IloValueSelectorArray valSel(env);
valSel.add(IloSelectRandomValue(env));
```

To inform the optimizer to use these selectors, you create an instance of the class `IloSearchPhase`, which is the container for the information regarding tuning the search strategy.

Step 22: Create the tuning object

Add the following code after the comment `//Create the tuning object`

```
IloSearchPhase phase(env, allGames, varSel, valSel);
```

As the search will likely terminate before optimality has been proven, you use a search that allows you to examine intermediate solutions instead of using `IloCP::solve`. To be able to display intermediate solutions, you use three other member functions of the class `IloCP`. You first use the member function `IloCP::startNewSearch`, whose argument is an instance of `IloSearchPhase`, to initialize the optimizer.

To look for a solution, you call `IloCP::next` in a loop. This tells the optimizer to:

- search for a solution in the manner defined by the tuning object passed to `IloCP::startNewSearch` and
- add the constraint that the following solution must be better than the solution found previously.

In other words, each time `IloCP::next` is called, the new solution will yield a better value for the objective. If no limit has been specified with a search parameter, the last solution found is the optimal solution.

In this lesson, you use a loop to find better and better solutions and print the cost for each solution found.

After the while loop terminates, you should use the member function `IloCP::end` to terminate the search and delete internal objects created by CP Optimizer to carry out the search. The code for ending the optimizer is given for you in the lesson code.

Step 23: Search for a solution

Add the following code after the comment `//Search for a solution`

```
cp.startNewSearch(phase);
while (cp.next()) {
```

The objective for each solution found is displayed. The code for displaying each solution is provided for you in the lesson code.

Step 24: Compile and run the program

Compile and run the program. Your results may vary from the ones reported here since on different systems the time limit will be reached at different points in the search.

```
Solution at 40
```

The complete program can be viewed online in the file `YourCPHome/examples/src/cpp/sports.cpp`.

Review exercises

Includes the review exercises and suggested answers.

In this section

Exercises
Lists the review exercises.

Suggested answers
Provides the suggested answers to the review exercises.

Exercises

For answers, see *Suggested answers*.

1. What is a tuple?
2. What are two types of additional constraints that can improve the process of finding a solution?
3. What is a logical constraint?

Suggested answers

Exercise 1

What is a tuple?

Suggested answer

An tuple is an ordered set of values. For example, if you state that Team 1 must play Team 2 in Game 7, this is a tuple. You would write it as (1, 2, 7). In the C++ API, the tuple would be represented by the array of values `IloIntArray(env,3,1,2,7)`.

Exercise 2

What are two types of additional constraints that can improve the process of finding a solution?

Suggested answer

Two types of constraints that can be added to improve constraint propagation and the search are surrogate constraints and constraints that reduce symmetry. Surrogate constraints are additional constraints that are valid for the model, but have not been explicitly expressed in the modeling constraints. Constraints that reduce symmetry eliminate extra solutions that can be considered as permutations of other solutions.

Exercise 3

What is a logical constraint?

Suggested answer

A logical constraint is created by combining constraints or placing constraints on other constraints. Logical constraints are based on the idea that constraints have value. IBM® ILOG® CP Optimizer handles constraints not only as if they have a Boolean value, such as true or false, but effectively as if the value is 0 or 1. This allows you to combine constraints into expressions or impose constraints on constraints

Complete program

You can view the complete sports scheduling program online in the file `YourCPHome/examples/src/cpp/sports.cpp`.

Results

Your results may likely vary from the ones here since on different systems the time limit will be reached at different points in the search.

```
Solution at 40
```

Using expressions on interval variables: house building with earliness and tardiness costs

This section describes modeling and solving a problem with expressions on interval variables.

In this section

Overview
Describes modeling and solving a problem with expressions on interval variables.

Describe
Describes the first stage in finding a solution to a house building problem.

Model
Describes the second stage in finding a solution to a house building problem.

Solve
Describes the third stage in finding a solution to a house building problem.

Review exercises
Includes the review exercises.

Complete program
Lists where to find the complete house building problem.

Overview

In this lesson, you will learn how to:

- use the class `IloIntervalVar`;
- use the specialized constraint `IloEndBeforeStart`;
- use the expressions `IloStartOf`, `IloStartEval`, `IloEndOf` and `IloEndEval`;
- use the class `IloNumToNumSegmentFunction`;
- use the class `IloCP` to search for a solution to and output the results of a scheduling problem.

You will learn how to model and solve a simple problem, a problem of scheduling the tasks involved in building a house in such a manner that minimizes an objective. Here the objective is to minimize the costs associated with preforming specific tasks before a preferred earliest start date or after a preferred latest end date. Some tasks must necessarily take place before other tasks, and each task has a given duration. To find a solution to this problem using IBM® ILOG® CP Optimizer, you will use the three-stage method: describe, model and solve.

Describe

The problem consists of assigning start dates to tasks in such a way that the resulting schedule satisfies precedence constraints and minimizes an expression. The objective for this problem is to minimize the earliness costs associated with starting certain tasks earlier than a given date and tardiness costs associated with completing certain tasks later than a given date.

For each task in the house building project, the following table shows the duration (measured in days) of the task along with the tasks that must finish before the task can start.

House construction tasks

Task	Duration	Preceding tasks
masonry	35	
carpentry	15	masonry
plumbing	40	masonry
ceiling	15	masonry
roofing	5	carpentry
painting	10	ceiling
windows	5	roofing
facade	10	roofing, plumbing
garden	5	roofing, plumbing
moving	5	windows, facade, garden, painting

Other information for the problem includes the earliness and tardiness costs associated with some tasks.

House construction task earliness costs

Task	Preferred earliest start date	Cost per day for starting early
masonry	25	200.0
carpentry	75	300.0
ceiling	75	100.0

House construction task tardiness costs

Task	Preferred latest end date	Cost per day for ending late
moving	100	400.0

Solving the problem consists of determining starting dates for the tasks such that the cost, determined by the earliness and lateness costs, is minimized.

Note: In CP Optimizer, the unit of time represented by an interval variable is not defined. As a result, the size of the masonry task in this problem could be 35 hours or 35 weeks or 35 months.

Step 1: Describe the problem

The first step in modeling and solving a problem is to write a natural language description of the problem, identifying the decision variables and the constraints on these variables.

Write a natural language description of this problem. Answer these questions:

- What is the known information in this problem?
- What are the decision variables or unknowns in this problem?
- What are the constraints on these variables?
- What is the objective?

Discussion

What is the known information in this problem?

- There are ten house building tasks, each with a given duration. For each task, there is a list of tasks that must be completed before the task can start. Some tasks also have costs associated with an early start date or late end date.

What are the decision variables or unknowns in this problem?

- The unknowns are the dates that the tasks will start. The cost is determined by the assigned start dates.

What are the constraints on these variables?

- In this case, each constraint specifies that a particular task may not begin until one or more given tasks have been completed.

What is the objective?

- The objective is to minimize the cost incurred through earliness and tardiness costs.

Model

After you have written a description of your problem, you can use IBM® ILOG® Concert Technology classes to model it.

Step 2: Open the example file

Open the example file `YourCPHome/examples/tutorial/cpp/sched_time_partial.cpp` in your development environment. This file is a program that is only partially completed. You will fill in the blanks in each step in this lesson. At the end, you will have completed the program code, and you can compile and run the program.

In this lesson, you include the header file `<ilcp/cp.h>`. To catch exceptions that may be thrown, you use a `try/catch` block. The code for creating the environment and model and for printing out the solution found by the CP Optimizer engine is provided.

IBM ILOG Concert Technology gives you the means to represent the unknowns in this problem, the interval in which each task will occur, as interval variables.

Note: **Interval decision variable**

Tasks are represented by the class `IloIntervallVar` in IBM ILOG Concert Technology.

An interval has a start time, an end time, a size and a length. An interval variable allows for these values to be variable in the model.

The length of a present interval variable is equal to the difference between its end time and its start time. The size is the actual amount of time the task takes to process. By default, the size is equal to the length, which is the difference between the end and the start of the interval. In general, the size is a lower bound on the length.

An interval variable may be optional. Whether an interval is present in the solution or not is represented by a decision variable. If an interval is not present in the solution, this means that any constraints on this interval act like the interval is “not there.” Exact semantics will depend on the specific constraint.

Logical relations can be expressed between the presence statuses of interval variables, allowing, for instance, to state that whenever the interval variable `a` is present then the interval variable `b` must also be present.

After you create an environment and a model, you declare the interval variables, one for each task. Each variable represents the unknown information, the scheduled interval for each activity. After the problem is solved, the values assigned to these interval variables will represent the solution to the problem. To improve the display of the solution, it is useful to assign a name to each of the interval variables.

Here is the constructor for the class `IloIntervalVar` that you use in this lesson:

```
IloIntervalVar(const IloEnv env,
               IloInt       sz,
               const char*  name = 0);
```

The first argument passed to this constructor of the class `IloIntervalVar` is the environment. The second argument is the size of the task. The third argument is an optional name used for debug and trace purposes.

Step 3: Declare the interval variables

Add the following code after the comment `//Declare the interval variables`

```
IloIntervalVar masonry  (env, 35, "masonry   ");
IloIntervalVar carpentry(env, 15, "carpentry ");
IloIntervalVar plumbing (env, 40, "plumbing  ");
IloIntervalVar ceiling  (env, 15, "ceiling   ");
IloIntervalVar roofing  (env, 5,  "roofing   ");
IloIntervalVar painting (env, 10, "painting  ");
IloIntervalVar windows  (env, 5,  "windows   ");
IloIntervalVar facade   (env, 10, "facade    ");
IloIntervalVar garden   (env, 5,  "garden    ");
IloIntervalVar moving   (env, 5,  "moving    ");
```

In this example, certain tasks can start only after other tasks have been completed. IBM ILOG CP Optimizer allows you to express constraints involving temporal relationships between pairs of interval variables using *precedence constraints*.

Note: **Precedence constraint**

Constraints are represented by the class `IloConstraint` in IBM ILOG Concert Technology.

Precedence constraints are used to specify when one interval variable must start or end with respect to the start or end time of another interval. The following types of precedence constraints are available; if `a` and `b` denote interval variables, both interval variables are present and `delay` is a number or integer expression (0 by default), then:

- `IloEndBeforeEnd(env, a, b, delay)` constrains that at least the given delay should elapse between the end of `a` and the end of `b`. It imposes the inequality `endTime(a) + delay <= endTime(b).`
- `IloEndBeforeStart(env, a, b, delay)` constrains that at least the given delay should elapse between the end of `a` and the start of `b`. It imposes the inequality `endTime(a) + delay <= startTime(b).`

- `IloEndAtEnd(env, a, b, delay)` constrains the given delay to separate the end of `a` and the end of `b`. It imposes the equality `endTime(a) + delay == endTime(b)`.
- `IloEndAtStart(env, a, b, delay)` constrains the given delay to separate the end of `a` and the start of `b`. It imposes the equality `endTime(a) + delay == startTime(b)`.
- `IloStartBeforeEnd(env, a, b, delay)` constrains that at least the given delay should elapse between the start of `a` and the end of `b`. It imposes the inequality `startTime(a) + delay <= endTime(b)`.
- `IloStartBeforeStart(env, a, b, delay)` constrains that at least the given delay should elapse between the start of `a` and the start of `b`. It imposes the inequality `startTime(a) + delay <= startTime(b)`.
- `IloStartAtEnd(env, a, b, delay)` constrains the given delay to separate the start of `a` and the end of `b`. It imposes the equality `startTime(a) + delay == endTime(b)`.
- `IloStartAtStart(env, a, b, delay)` constrains the given delay to separate the start of `a` and the start of `b`. It imposes the equality `startTime(a) + delay == startTime(b)`.

If either interval `a` or `b` is not present in the solution, the constraint is automatically satisfied, and it is as if the constraint was never imposed.

You use the member function `IloModel::add` to add constraints to the model. You must explicitly add a constraint to the model object or the CP Optimizer engine will not use it in the search for a solution.

Step 4: Add the precedence constraints

Add the following code after the comment `//Add the precedence constraints`

```
model.add(IloEndBeforeStart(env, masonry,   carpentry));
model.add(IloEndBeforeStart(env, masonry,   plumbing));
model.add(IloEndBeforeStart(env, masonry,   ceiling));
model.add(IloEndBeforeStart(env, carpentry, roofing));
model.add(IloEndBeforeStart(env, ceiling,   painting));
model.add(IloEndBeforeStart(env, roofing,   windows));
model.add(IloEndBeforeStart(env, roofing,   facade));
model.add(IloEndBeforeStart(env, plumbing,  facade));
model.add(IloEndBeforeStart(env, roofing,   garden));
model.add(IloEndBeforeStart(env, plumbing,  garden));
model.add(IloEndBeforeStart(env, windows,   moving));
model.add(IloEndBeforeStart(env, facade,    moving));
model.add(IloEndBeforeStart(env, garden,    moving));
```

```
   model.add(IloEndBeforeStart(env, painting,  moving));
```

To model the cost for starting a task earlier than the preferred starting date, you write a function based on the expression that represents the starting date of the given task. This lesson illustrates two methods for creating this function. Concert Technology provides the expression, `IloStartOf`, to represent the start time of an interval variable as an integer expression. The single argument passed to the function `IloStartOf` is the appropriate interval variable. Here is the function:

```
IloIntExprArg IloStartOf (const IloIntervalVar a);
```

To model the cost for starting a task earlier than the preferred starting date, you use the expression `IloStartOf` that represents the start time of an interval variable as an integer expression.

For each task that has an earliest preferred start date, you determine how many days before the preferred date it is scheduled to start using the expression `IloStartOf`; this expression can be negative if the task starts after the preferred date. By taking the maximum of this value and 0 using `IloMax`, you determine how many days early the task is scheduled to start. Weighting this value with the cost per day of starting early, you determine the cost associated with the task.

Alternatively, you can represent the earliness cost as a piecewise linear function. Concert Technology provides the class `IloNumToNumSegmentFunction` to model a piecewise linear function.

Note: **Segment function**

Piecewise linear functions are represented by the class `IloNumToNumSegmentFunction` in IBM ILOG Concert Technology.

Each interval *[x1*, *x2)* on which the function is linear is called a *segment*.

When two consecutive segments of the function are co-linear, these segments are merged so that the function is always represented with the minimal number of segments.

In this case, to create an instance of the `IloNumToNumSegmentFunction`, you use the function `IloPiecewiseLinear`. The first argument passed to the function is the environment. The second argument is an array of points that are the endpoints of the segments. The third argument is an array of slopes. The fourth argument is a point, and the fifth argument is the value of the function at that point. The sixth argument is an optional name used for debug and trace purposes. Here is the function:

```
IloNumToNumSegmentFunction
              IloPiecewiseLinearFunction(const IloEnv env,
                                         const IloNumArray point,
                                         const IloNumArray slope,
                                         IloNum a,
                                         IloNum fa,
```

```
                                                const char* name = 0);
```

The piecewise linear function for the earliness function in this problem has two segments, one from `-IloInfinity` to the preferred earliest start date and then one from the preferred earliest start date to `IloInfinity`. The cost function on the leftmost interval is `-weight`; on the rightmost interval, the function value is 0.

The function `IloStartEval`, when called with an interval variable and a segment function, returns an expression that represents the value of the function at the start of the interval.

Step 5: Add the earliness function

Add the following code after the comment `//Add the earliness function`

```
IloNumExpr EarlinessCost(IloIntervalVar task, IloInt rd, IloNum weight, IloBool
 useFunction) {
  IloEnv env = task.getEnv();
  if (useFunction) {
    IloNumToNumSegmentFunction f =
      IloPiecewiseLinearFunction(env,
         IloNumArray(env, 1, rd),
         IloNumArray(env, 2, -weight, 0.0),
         rd, 0.0);
    return IloStartEval(task, f);
  }
  else
    return weight * IloMax(0, rd-IloStartOf(task));
}
```

The cost for ending a task later than the preferred date is modeled in a similar manner, using the `IloEndOf` function or the `IloNumToNumSegmentFunction` class and `IloEndEval` function provided by Concert Technology.

Step 5: Add the tardiness function

Add the following code after the comment `//Add the tardiness function`

```
IloNumExpr TardinessCost(IloIntervalVar task, IloInt dd, IloNum weight, IloBool
 useFunction) {
  IloEnv env = task.getEnv();
  if (useFunction) {
   IloNumToNumSegmentFunction f =
      IloPiecewiseLinearFunction(env,
         IloNumArray(env, 1, dd),
         IloNumArray(env, 2, 0.0, weight),
         dd, 0.0);
    return IloEndEval(task, f);
  }
  else
```

```
    return weight * IloMax(0, IloEndOf(task)-dd);
}
```

These two cost functions are used to create an expression that models the overall cost. The resulting `IloNumExpr` is passed to an `IloMinimize` function, and the objective added to the model.

Step 6: Add the objective

Add the following code after the comment `//Add the objective`

```
    IloBool useFunction = IloTrue;
    IloNumExpr cost(env);
    cost += EarlinessCost(masonry,   25, 200.0, useFunction);
    cost += EarlinessCost(carpentry, 75, 300.0, useFunction);
    cost += EarlinessCost(ceiling,   75, 100.0, useFunction);
    cost += TardinessCost(moving,   100, 400.0, useFunction);
    model.add(IloMinimize(env, cost));
```

Solve

Solving a problem consists of finding a value for each decision variable so that all constraints are satisfied. You may not always know beforehand whether there is a solution that satisfies all the constraints of the problem. In some cases, there may be no solution. In other cases, there may be many solutions to a problem.

You use an instance of the class `IloCP` to solve a problem expressed in a model. The constructor for `IloCP` takes an `IloModel` as its argument.

Step 7: Create an instance of IloCP

Add the following code after the comment `//Create an instance of IloCP`

```
IloCP cp(model);
```

You now use the member function `IloCP::solve`, which solves the problem contained in the model by using constructive search and constraint propagation.

Step 8: Search for a solution

Add the following code after the comment `//Search for a solution`

```
if (cp.solve()) {
```

The member function `IloCP::solve` returns a Boolean value of type `IloBool`. If a solution is found, the value `IloTrue` is returned.

After a solution has been found, you can use the member functions `IloCP::getObjValue` and `IloCP::domain` to examine the solution. The stream `IloAlgorithm::out` is the communication stream for general output.

The code for displaying the solution has been provided for you:

```
  cp.out() << "Cost Value: " << cp.getObjValue() << std::endl;
  cp.out() << cp.domain(masonry)   << std::endl;
  cp.out() << cp.domain(carpentry) << std::endl;
  cp.out() << cp.domain(plumbing)  << std::endl;
  cp.out() << cp.domain(ceiling)   << std::endl;
  cp.out() << cp.domain(roofing)   << std::endl;
  cp.out() << cp.domain(painting)  << std::endl;
  cp.out() << cp.domain(windows)   << std::endl;
  cp.out() << cp.domain(facade)    << std::endl;
  cp.out() << cp.domain(garden)    << std::endl;
  cp.out() << cp.domain(moving)    << std::endl;
} else {
  cp.out() << "No solution found. " << std::endl;
```

```
    }
    cp.printInformation();
```

Step 9: Compile and run the program

Compile and run the program. You should get the following results:

```
Cost Value: 5000
masonry   [1: 20 -- 35 --> 55]
carpentry [1: 75 -- 15 --> 90]
plumbing  [1: 55 -- 40 --> 95]
ceiling   [1: 75 -- 15 --> 90]
roofing   [1: 90 -- 5 --> 95]
painting  [1: 90 -- 10 --> 100]
windows   [1: 95 -- 5 --> 100]
facade    [1: 95 -- 10 --> 105]
garden    [1: 95 -- 5 --> 100]
moving    [1: 105 -- 5 --> 110]
```

As you can see, the overall cost is 5000 and moving will be completed by day 110.

The complete program can be viewed online in the `YourCPHome/examples/src/cpp/sched_time.cpp` file.

Review exercises

Includes the review exercises.

In this section

Exercises
Lists the review exercises.

Suggested answers
Provides the answers to the review exercises.

Exercises

For answers, see *Suggested answers*.

1. What is an interval variable?
2. What is a precedence constraint?

Suggested answers

Exercise 1

What is an interval variable?

Suggested answer

An interval has a start, an end, a size and a length. An interval variable allows for these values to be variable in the model.

The length of a present interval variable is equal to the difference between its end time and its start time. The size is the actual amount of time the task takes to process. By default, the size is equal to the length, which is the difference between the end and the start of the interval. In general, the size is a lower bound on the length.

An interval variable may be optional. Whether an interval is present in the solution or not is represented by a decision variable. If an interval is not present in the solution, this means that any constraints on this interval acts like the interval is “not there.” Exact semantics will depend on the specific constraint.

Logical relations can be expressed between the presence statuses of interval variables, allowing, for instance, to state that whenever the interval variable `a` is present then the interval variable `b` must also be present.

Exercise 2

What is a precedence constraint?

Suggested answer

A precedence constraint is a constraint that specifies when one interval variable must start or end with respect to the start or end time of another interval variable.

Complete program

You can view the complete house building program in the file `YourCPHome/examples/src/cpp/sched_time.cpp`.

Results

```
Cost Value: 5000
masonry   [1: 20 -- 35 --> 55]
carpentry [1: 75 -- 15 --> 90]
plumbing  [1: 55 -- 40 --> 95]
ceiling   [1: 75 -- 15 --> 90]
roofing   [1: 90 -- 5 --> 95]
painting  [1: 90 -- 10 --> 100]
windows   [1: 95 -- 5 --> 100]
facade    [1: 95 -- 10 --> 105]
garden    [1: 95 -- 5 --> 100]
moving    [1: 105 -- 5 --> 110]
```

Using no overlap constraints on interval variables: house building with workers

This section describes how to model and solve a problem using no overlap constraints.

In this section

Overview
Describes how to model and solve a problem with no overlap constraints.

Describe
Describes the first stage in modeling and solving a house building with workers problem.

Model
Describes the second stage in modeling and solving a house building with workers problem.

Solve
Describes the third stage in modeling and solving a house building with workers problem.

Review exercises
Includes the review exercises.

Complete program
Lists the complete house building with workers program.

Overview

In this lesson, you will learn how to:

- use the classes `IloIntervalSequenceVar`;
- use the specialized constraint `IloNoOverlap`;
- use the classes `IloTransitionDistance` and `IloSpan`;
- use the expression `IloLengthOf`.

You will learn how to model and solve a problem of scheduling the tasks involved in building multiple houses in a manner that minimizes the costs associated with completing each house after a given due date and with the length of time it takes to build each house. Some tasks must necessarily take place before other tasks, and each task has a predefined size. Each house has an earliest starting date. Moreover, there are two workers, each of whom must perform a given subset of the necessary tasks, and there is a transition time associated with a worker transferring from one house to another house. A task, once started, cannot be interrupted. The objective is to minimize the cost, which is composed of tardiness costs for certain tasks as well as a cost associated with the length of time it takes to complete each house. To find a solution to this problem using IBM® ILOG® CP Optimizer, you will use the three-stage method: describe, model and solve.

Describe

The problem consists of assigning start dates to a set of tasks in such a way that the schedule satisfies temporal constraints and minimizes an expression. The objective for this problem is to minimize the tardiness costs associated with completing each house later than its specified due date and the cost associated with the length of time it takes to complete each house.

For each task type in the house building project, the following table shows the duration of the task in days along with the tasks that must be finished before the task can start. In addition, each type of task must be performed by a specific worker, Jim or Joe. A worker can only work on one task at a time; each task, once started, may not be interrupted. The time required to transfer from one house to another house is determined by a function based on the location of the two houses.

House construction tasks

Task	Duration	Worker	Preceding tasks
masonry	35	Joe	
carpentry	15	Joe	masonry
plumbing	40	Jim	masonry
ceiling	15	Jim	masonry
roofing	5	Joe	carpentry
painting	10	Jim	ceiling
windows	5	Jim	roofing
facade	10	Joe	roofing, plumbing
garden	5	Joe	roofing, plumbing
moving	5	Jim	windows, facade, garden, painting

For each of the five houses that must be built, there is an earliest starting date, a due date and a cost per day of completing the house later than the preferred due date.

House construction tardiness costs

House	Earliest starting date	Preferred latest end date	Cost per day for ending late
0	0	120	100.0
1	0	212	100.0
2	151	304	100.0
3	59	181	200.0
4	243	425	100.0

Solving the problem consists of determining starting dates for the tasks such that the cost, where the cost is determined by the lateness costs and length costs, is minimized.

Step 1: Describe the problem

The first step in modeling and solving a problem is to write a natural language description of the problem, identifying the decision variables and the constraints on these variables.

Write a natural language description of this problem. Answer these questions:

- What is the known information in this problem?
- What are the decision variables or unknowns in this problem?
- What are the constraints on these variables?
- What is the objective?

Discussion

What is the known information in this problem?

- There are five houses to be built by two workers. For each house, there are ten house building tasks, each with a given size. Each house also has a given earliest starting date. For each task, there is a list of tasks that must be completed before the task can start. Each task must be performed by a given worker, and there is a transition time associated with a worker transferring from one house to another house. There are costs associated with completing each house after its preferred due date and with the length of time it takes to complete each house.

What are the decision variables or unknowns in this problem?

- The unknowns are the start and end dates of the interval variables associated with the tasks. Once fixed, these interval variables also determine the cost of the solution. For some of the interval variables, there is a fixed minimum start date.

What are the constraints on these variables?

- There are constraints that specify a particular task may not begin until one or more given tasks have been completed. In addition, there are constraints that specify that a worker can be assigned to only one task at a time and that it takes time for a worker to travel from one house to the other.

What is the objective?

- The objective is to minimize the cost incurred through tardiness and length costs.

Model

After you have written a description of your problem, you can use IBM® ILOG® Concert Technology classes to model it.

Step 2: Open the example file

Open the example file `YourCPHome/examples/tutorial/cpp/sched_sequence_partial.cpp` in your development environment. This file is a program that is only partially completed. You will fill in the blanks in each step in this lesson. At the end, you will have completed the program code, and you can compile and run the program.

In this lesson, you include the header file `<ilcp/cp.h>`. To catch exceptions that may be thrown, you use a `try/catch` block. The code for creating the environment and model and for printing out the solution found by the CP Optimizer engine is provided.

In addition, the data related to the tasks, such as the tasks (`Tasks`), the number of tasks (`NbTasks`), the names of the tasks (`TaskNames`) and sizes of the tasks (`TaskDurations`), are provided.

After you create an environment and a model, you need to define the decision variables and add the constraints and objective to the model. Since the requirements for each of the five houses are similar, you use a function, `MakeHouse`, to create the decision variables, constraints and costs associated with a house. The information about individual houses that must be shared with the main function includes the cost expression and the set of tasks associated with each worker. This set of tasks is needed in order to create the constraints for each worker that involve tasks of different houses. In order to display the results of the optimization, it is also useful to maintain an array of all the interval variables.

To access this global information, you create objects to be updated in the `MakeHouse` function. An array of task interval variables, `allTasks`, stores all the interval variables that are created. The overall cost is represented by a numerical expression called `cost`. For the worker Joe, you create two arrays to be filled by the `MakeHouse` function: `joeTasks`, an array of interval variables, and `joeLocations`, an array of integers. These two corresponding arrays are indexed the same, so that for each index `i`, `joeTasks[i]` is performed at `joeLocation[i]`. A similar pair of arrays are created for the worker Jim. These four arrays are passed to the `MakeHouse` function which adds elements to these arrays as the interval variables and constraints are created

Step 3: Declare the objects needed for MakeHouse

Add the following code after the comment `//Declare the objects needed for MakeHouse`

```
IloNumExpr cost(env);
IloIntervalVarArray allTasks(env);
IloIntervalVarArray joeTasks(env);
IloIntervalVarArray jimTasks(env);
IloIntArray joeLocations(env);
IloIntArray jimLocations(env);
```

You pass the model, the cost expression, the array of all tasks, the arrays `joeTasks`, `jimTasks`, `joeLocations` and `jimLocations`, the identifier of the current house, the earliest date the house can be started, the preferred due date of the house and the cost per day of completing the house late as arguments to the `MakeHouse` function.

Step 4: Create the MakeHouse function

Add the following code after the comment `//Create the MakeHouse function`

```
void MakeHouse(IloModel model,
      IloNumExpr cost,
      IloIntervalVarArray allTasks,
      IloIntervalVarArray joeTasks,
      IloIntervalVarArray jimTasks,
      IloIntArray joeLocations,
      IloIntArray jimLocations,
      IloInt loc,
      IloInt rd,
      IloInt dd,
      IloNum weight) {
```

Each house has a list of `NbTasks` that must be scheduled. Task `i`, where `i` is in `0..NbTasks-1`, has a size of `TaskDurations[i]` and the name `TaskNames[i]`. Using these, you build an array `tasks` of interval variables. As you create each interval variable, you also add it to the array `allTasks` that will be used to display the solution once the schedule has been determined.

Step 5: Create the interval variables

Add the following code after the comment `//Create the interval variables`

```
  char name[128];

  IloIntervalVarArray tasks(env, NbTasks);
  for (IloInt i=0; i<NbTasks; ++i) {
    sprintf(name, "H%ld-%s", loc, TaskNames[i]);
    tasks[i] = IloIntervalVar(env, TaskDurations[i], name);
    allTasks.add(tasks[i]);
  }
```

To model the cost associated with the length of time it takes to build a single house, you create an interval variable that starts at the start of the first task of the house and ends at the end of the last task. This interval variable must *span* the tasks.

Note: **Span constraint**

With the specialized constraint `IloSpan`, you can create a constraint that specifies that one interval variable must exactly cover a set of interval variables.

In other words, the spanning interval `a` is present in the solution if and only if at least one of the spanned interval variables is present and, in this case, the spanning interval variable starts at the start of the interval variable scheduled earliest in the set and ends at the end of the interval variable scheduled latest in the set.

The first argument passed to the constructor of the class `IloSpan` is the environment. The second argument is the interval variable that will be constrained to cover the set of interval variables. The third argument is the array of interval variables that are to be covered. The final argument is an optional name used for debug and trace purposes. Here is a constructor:

```
IloSpan (const IloEnv env,
         const IloIntervalVar a,
         const IloIntervalVarArray bs,
         const char *name =0);
```

You create an interval variable called `house` and constrain the variable to cover the array of tasks, `tasks`.

Step 6: Add the house interval variable and span constraint

Add the following code after the comment `//Add the house interval variable and span constraint`

```
sprintf(name, "H%ld", loc);
IloIntervalVar house(env, name);
model.add(IloSpan(env, house, tasks));
```

The tasks of the house building project have precedence constraints that are added to the model. Moreover, each house has an earliest starting date which can be modeled with a *time bound modifier*.

Note: **Interval variable modifier**

Properties of interval variables can be modified.

Time bound modifiers are used to limit the possible values that may be assigned to the start, length, size, or end of an interval variable. These modifiers include `setStartMin`, `setStartMax`, `setEndMin`, `setEndMax`, `setLengthMin`, `setLengthMax`., `setSizeMin` and `setSizeMax`. For example, if `a` denotes an interval variable and `val` is a number, then:

- `a.setStartMin(val)` constrains that `a` must not start before `val` or `a` is not present in the solution. It imposes the inequality `startTime(a) >= val`.

Other modifiers of interval variables include `setPresent`, `setAbsent` and `setOptional` which indicate whether or not an interval variable must be present.

Step 7: Add the precedence and time bound constraints

Add the following code after the comment `//Add the precendence and time bound constraints`

```
house.setStartMin(rd);
model.add(IloEndBeforeStart(env, tasks[masonry],   tasks[carpentry]));
model.add(IloEndBeforeStart(env, tasks[masonry],   tasks[plumbing]));
model.add(IloEndBeforeStart(env, tasks[masonry],   tasks[ceiling]));
model.add(IloEndBeforeStart(env, tasks[carpentry], tasks[roofing]));
model.add(IloEndBeforeStart(env, tasks[ceiling],   tasks[painting]));
model.add(IloEndBeforeStart(env, tasks[roofing],   tasks[windows]));
model.add(IloEndBeforeStart(env, tasks[roofing],   tasks[facade]));
model.add(IloEndBeforeStart(env, tasks[plumbing],  tasks[facade]));
model.add(IloEndBeforeStart(env, tasks[roofing],   tasks[garden]));
model.add(IloEndBeforeStart(env, tasks[plumbing],  tasks[garden]));
model.add(IloEndBeforeStart(env, tasks[windows],   tasks[moving]));
model.add(IloEndBeforeStart(env, tasks[facade],    tasks[moving]));
model.add(IloEndBeforeStart(env, tasks[garden],    tasks[moving]));
model.add(IloEndBeforeStart(env, tasks[painting],  tasks[moving]));
```

Each of the tasks requires a particular worker. As a worker can perform only one task at a time, it is necessary to know all of the tasks that a worker must perform and then constrain that these intervals not overlap. Also, as there are transition times between houses that must be taken into account, it is necessary to know where each task is to be performed. To create the no overlap and transition time constraints in the main function, you add the appropriate tasks to the arrays `joeTasks` and `jimTasks`. To indicate at which house the task is performed, whenever a task is added to a worker's task array, a location is added to that worker's location array.Thus the two corresponding arrays are indexed the same, so that for each index `i`, `joeTasks[i]` is performed at `joeLocation[i]`.

Step 8: Add the tasks to workers

Add the following code after the comment `//Add the tasks to workers`

```
joeTasks.add(tasks[masonry]);
joeLocations.add(loc);
joeTasks.add(tasks[carpentry]);
joeLocations.add(loc);
jimTasks.add(tasks[plumbing]);
```

```
  jimLocations.add(loc);
  jimTasks.add(tasks[ceiling]);
  jimLocations.add(loc);
  joeTasks.add(tasks[roofing]);
  joeLocations.add(loc);
  jimTasks.add(tasks[painting]);
  jimLocations.add(loc);
  jimTasks.add(tasks[windows]);
  jimLocations.add(loc);
  joeTasks.add(tasks[facade]);
  joeLocations.add(loc);
  joeTasks.add(tasks[garden]);
  joeLocations.add(loc);
  jimTasks.add(tasks[moving]);
  jimLocations.add(loc);
```

To model the cost of building the house, you create a function that uses the function `IloEndOf` to model the cost associated with a task being completed later than its preferred latest end date.

Step 9: Add the tardiness cost function

Add the following code after the comment `//Add the tardiness cost function`

```
IloNumExpr TardinessCost(IloIntervalVar task, IloInt dd, IloNum weight) {
  IloEnv env = task.getEnv();
  return weight * IloMax(0, IloEndOf(task)-dd);
}
```

This cost function returns an expression that models the tardiness cost for the end date of the house interval variable. The cost for building a house is the sum of the tardiness cost and the number of days it takes from start to finish building the house. To model the cost of the length of time it takes to build the house, you use the function `IloLengthOf`, which returns an expression representing the length of an interval variable.

Step 10: Add the cost expression

Add the following code after the comment `//Add the cost expression`

```
  cost += TardinessCost(house, dd, weight);
  cost += IloLengthOf(house);
```

This completes the `MakeHouse` function. In the main function, you now call the `MakeHouse` function, once for each house. At each call, the cost expression is incremented by the cost associated with that house and additional elements are appended to the arrays `allTasks`, `joeTasks`, `jimTasks`, `joeLocations` and `jimLocations`.

Step 11: Create the houses

Add the following code after the comment `//Create the houses`

```
MakeHouse(model, cost, allTasks, joeTasks, jimTasks, joeLocations,
  jimLocations, 0, 0,   120, 100.0);
MakeHouse(model, cost, allTasks, joeTasks, jimTasks, joeLocations,
  jimLocations, 1, 0,   212, 100.0);
MakeHouse(model, cost, allTasks, joeTasks, jimTasks, joeLocations,
  jimLocations, 2, 151, 304, 100.0);
MakeHouse(model, cost, allTasks, joeTasks, jimTasks, joeLocations,
  jimLocations, 3, 59,  181, 200.0);
MakeHouse(model, cost, allTasks, joeTasks, jimTasks, joeLocations,
  jimLocations, 4, 243, 425, 100.0);
```

You now model the transition times associated with the workers transferring between houses.

Note: **Transition time object**

The class `IloTransitionDistance` in IBM ILOG Concert Technology lets you build a table of transition times to apply to a sequence of non-overlapping interval variables. An instance of this class is a table of non-negative numbers, indexed by an integer interval variable type associated with each interval variable.

Given an interval variable `a1` that precedes (not necessarily directly) an interval variable `a2` in a sequence of non-overlapping interval variables, the transition time between `a1` and `a2` is an amount of time that must elapse between the end of `a1` and the beginning of `a2`.

The first argument passed to the constructor of the class `IloTransitionDistance` is the environment. The second argument is the number of transition types. The final argument is an optional name used for debug and trace purposes. Here is a constructor:

```
IloTransitionDistance(const IloEnv env,
                      IloInt size,
                      const char* name = 0);
```

In this problem, there are five houses, thus the number of types of interval variables is also five. The transition time from one house to another is the absolute difference of the associated house identifiers.

Step 12: Create the transition times

Add the following code after the comment `//Create the transition times`

```
IloTransitionDistance tt(env, 5);
for (i=0; i<5; ++i)
  for (j=0; j<5; ++j)
    tt.setValue(i, j, IloAbs(i-j));
```

To add the constraints that Jim and Joe can only perform one task at a time and must respect transition times, you create a sequence variable that represents the order in which the workers perform the tasks. Note that the sequence variable does not force the tasks to not overlap or the order of tasks; in a future step you create a constraint that enforces these relations on the sequence of interval variables.

Note: **Interval sequence decision variable**

Using the class `IloIntervalSequenceVar` in Concert Technology, you can create a variable that represents a sequence of interval variables. The sequence can contain a subset of the variables or be empty. In a solution, the sequence will represent a total order over all the intervals in the set that are present in the solution.

The assigned order of interval variables in the sequence does not necessarily determine their relative positions in time in the schedule.

The first argument passed to the constructor of the class `IloIntervalSequenceVar` is always the environment. The second argument is the array of interval variables to be sequenced. The third argument is the array of integer types of the interval variables in the second argument. The final argument is an optional name used for debug and trace purposes. Here is a constructor:

```
IloIntervalSequenceVar(const IloEnv env,
                       const IloIntervalVarArray a,
                       const IloIntArray types,
                       const char* name=0);
```

You create interval sequence variables for Jim and Joe, using the arrays of their tasks and task locations as the arguments.

Step 13: Create the sequence variables

Add the following code after the comment `//Create the sequence variables`

```
IloIntervalSequenceVar joe(env, joeTasks, joeLocations, "Joe");
IloIntervalSequenceVar jim(env, jimTasks, jimLocations, "Jim");
```

Now that you have created the sequence variables, you must constrain each sequence such that the interval variables do not overlap in the solution, that the transition times are

respected and that the sequence represents the relations of the interval variables in time. To do this, you use the specialized constraint `IloNoOverlap`.

Note: **No overlap constraint**

Using the class `IloNoOverlap` in Concert Technology, you can constrain that an interval sequence variable defines a chain of non-overlapping intervals that are present in the solution. If a transition matrix is specified, it defines the minimal time that must elapse between two intervals in the chain.

Note that intervals which are not present in the solution are automatically removed from the sequence.

In this case, the first argument passed to the constructor of the class `IloNoOverlap` is the environment. The second argument is the sequence of interval variables. The third argument is the transition object. The final argument is an optional name used for debug and trace purposes. Here is a constructor:

```
IloNoOverlap(const IloEnv env,
             const IloIntervalSequenceVar seq,
             const IloTransitionDistance ttime =0,
             const char* name=0);
```

You add one no overlap constraint for the sequence interval variable for each worker.

Step 14: Add the no overlap constraints

Add the following code after the comment `//Add the no overlap constraints`

```
  model.add(IloNoOverlap(env, joe, tt));
  model.add(IloNoOverlap(env, jim, tt));
```

The objective of this problem is to minimize the cost as represented by the cost expression.

Step 15: Add the objective

Add the following code after the comment `//Add the objective`

```
  model.add(IloMinimize(env, cost));
```

Solve

You use an instance of the class `IloCP` to solve a problem expressed in a model. The constructor for `IloCP` takes an `IloModel` as its argument.

Step 16: Create an instance of IloCP

Add the following code after the comment `//Create an instance of IloCP`

```
   IloCP cp(model);
```

You now use the member function `IloCP::solve`, which solves the problem contained in the model by using constructive search and constraint propagation. The search for an optimal solution in this problem could potentiality take a long time, so you place a fail limit on the solve process. The search will stop when the fail limit is reached, even if optimality of the current best solution is not guaranteed.

Step 17: Search for a solution

Add the following code after the comment `//Search for a solution`

```
   cp.setParameter(IloCP::FailLimit, 30000);
   if (cp.solve()) {
```

The member function `IloCP::solve` returns a Boolean value of type `IloBool`. If a solution is found, the value `IloTrue` is returned.

After a solution has been found, you can use the member functions `IloCP::getObjValue` and `IloCP::domain` to examine the solution. The stream `IloAlgorithm::out` is the communication stream for general output. The code for displaying the solution has been provided for you:

```
      cp.out() << "Solution with objective " << cp.getObjValue() << ":" <<
std::endl;
      for (i=0; i<allTasks.getSize(); ++i) {
        cp.out() << cp.domain(allTasks[i]) << std::endl;
      }
```

Step 18: Compile and run the program

Compile and run the program. You should get the following results:

```
Solution with objective 13852:
H0-masonry  [1: 1 -- 35 --> 36]
```

```
H0-carpentry[1: 36 -- 15 --> 51]
H0-plumbing [1: 36 -- 40 --> 76]
H0-ceiling  [1: 76 -- 15 --> 91]
H0-roofing  [1: 51 -- 5 --> 56]
H0-painting [1: 91 -- 10 --> 101]
H0-windows  [1: 101 -- 5 --> 106]
H0-facade   [1: 97 -- 10 --> 107]
H0-garden   [1: 107 -- 5 --> 112]
H0-moving   [1: 112 -- 5 --> 117]
H1-masonry  [1: 138 -- 35 --> 173]
H1-carpentry[1: 192 -- 15 --> 207]
H1-plumbing [1: 197 -- 40 --> 237]
H1-ceiling  [1: 237 -- 15 --> 252]
H1-roofing  [1: 207 -- 5 --> 212]
H1-painting [1: 252 -- 10 --> 262]
H1-windows  [1: 262 -- 5 --> 267]
H1-facade   [1: 252 -- 10 --> 262]
H1-garden   [1: 262 -- 5 --> 267]
H1-moving   [1: 267 -- 5 --> 272]
H2-masonry  [1: 216 -- 35 --> 251]
H2-carpentry[1: 268 -- 15 --> 283]
H2-plumbing [1: 273 -- 40 --> 313]
H2-ceiling  [1: 313 -- 15 --> 328]
H2-roofing  [1: 283 -- 5 --> 288]
H2-painting [1: 328 -- 10 --> 338]
H2-windows  [1: 338 -- 5 --> 343]
H2-facade   [1: 333 -- 10 --> 343]
H2-garden   [1: 328 -- 5 --> 333]
H2-moving   [1: 343 -- 5 --> 348]
H3-masonry  [1: 59 -- 35 --> 94]
H3-carpentry[1: 115 -- 15 --> 130]
H3-plumbing [1: 120 -- 40 --> 160]
H3-ceiling  [1: 160 -- 15 --> 175]
H3-roofing  [1: 130 -- 5 --> 135]
H3-painting [1: 175 -- 10 --> 185]
H3-windows  [1: 185 -- 5 --> 190]
H3-facade   [1: 180 -- 10 --> 190]
H3-garden   [1: 175 -- 5 --> 180]
H3-moving   [1: 190 -- 5 --> 195]
H4-masonry  [1: 291 -- 35 --> 326]
H4-carpentry[1: 345 -- 15 --> 360]
H4-plumbing [1: 350 -- 40 --> 390]
H4-ceiling  [1: 390 -- 15 --> 405]
H4-roofing  [1: 360 -- 5 --> 365]
H4-painting [1: 405 -- 10 --> 415]
H4-windows  [1: 415 -- 5 --> 420]
H4-facade   [1: 390 -- 10 --> 400]
H4-garden   [1: 400 -- 5 --> 405]
H4-moving   [1: 420 -- 5 --> 425]
```

The complete program can be viewed online in the `YourCPHome/examples/src/cpp/sched_sequence.cpp` file.

Review exercises

Includes the review exercises.

In this section

Exercises
Lists the review exercises.

Suggested answers
Provides the answers to the review exercises.

Exercises

For answers, see *Suggested answers*.

1. What is transition time?
2. What is a no overlap constraint?

Suggested answers

Exercise 1

What is transition time?

Suggested answer

Given an interval variable `a1` that precedes an interval variable `a2` in a sequence of non-overlapping interval variables, the transition time between `a1` and `a2` is the amount of time that must elapse between the end of `a1` and the beginning of `a2`.

Exercise 2

What is a no overlap constraint?

Suggested answer

A no overlap constraint, represented by the class `IloNoOverlap`, constrains a sequence of interval variables to not overlap in time. This constraint also forces the scheduling of the interval variables to respect a transition time table. If the interval variables are specified as a `IloIntervalSequenceVar`, then the order of the intervals in sequence variable will be forced to respect the temporal relations of the interval variables.

Complete program

The complete house building program can be viewed online in the file `YourCPHome/examples/src/cpp/sched_sequence.cpp`.

Results

```
Solution with objective 13852:
H0-masonry  [1: 1 -- 35 --> 36]
H0-carpentry[1: 36 -- 15 --> 51]
H0-plumbing [1: 36 -- 40 --> 76]
H0-ceiling  [1: 76 -- 15 --> 91]
H0-roofing  [1: 51 -- 5 --> 56]
H0-painting [1: 91 -- 10 --> 101]
H0-windows  [1: 101 -- 5 --> 106]
H0-facade   [1: 97 -- 10 --> 107]
H0-garden   [1: 107 -- 5 --> 112]
H0-moving   [1: 112 -- 5 --> 117]
H1-masonry  [1: 138 -- 35 --> 173]
H1-carpentry[1: 192 -- 15 --> 207]
H1-plumbing [1: 197 -- 40 --> 237]
H1-ceiling  [1: 237 -- 15 --> 252]
H1-roofing  [1: 207 -- 5 --> 212]
H1-painting [1: 252 -- 10 --> 262]
H1-windows  [1: 262 -- 5 --> 267]
H1-facade   [1: 252 -- 10 --> 262]
H1-garden   [1: 262 -- 5 --> 267]
H1-moving   [1: 267 -- 5 --> 272]
H2-masonry  [1: 216 -- 35 --> 251]
H2-carpentry[1: 268 -- 15 --> 283]
H2-plumbing [1: 273 -- 40 --> 313]
H2-ceiling  [1: 313 -- 15 --> 328]
H2-roofing  [1: 283 -- 5 --> 288]
H2-painting [1: 328 -- 10 --> 338]
H2-windows  [1: 338 -- 5 --> 343]
H2-facade   [1: 333 -- 10 --> 343]
H2-garden   [1: 328 -- 5 --> 333]
H2-moving   [1: 343 -- 5 --> 348]
H3-masonry  [1: 59 -- 35 --> 94]
H3-carpentry[1: 115 -- 15 --> 130]
H3-plumbing [1: 120 -- 40 --> 160]
H3-ceiling  [1: 160 -- 15 --> 175]
H3-roofing  [1: 130 -- 5 --> 135]
H3-painting [1: 175 -- 10 --> 185]
H3-windows  [1: 185 -- 5 --> 190]
H3-facade   [1: 180 -- 10 --> 190]
H3-garden   [1: 175 -- 5 --> 180]
H3-moving   [1: 190 -- 5 --> 195]
H4-masonry  [1: 291 -- 35 --> 326]
H4-carpentry[1: 345 -- 15 --> 360]
H4-plumbing [1: 350 -- 40 --> 390]
```

```
H4-ceiling  [1: 390 -- 15 --> 405]
H4-roofing  [1: 360 -- 5 --> 365]
H4-painting [1: 405 -- 10 --> 415]
H4-windows  [1: 415 -- 5 --> 420]
H4-facade   [1: 390 -- 10 --> 400]
H4-garden   [1: 400 -- 5 --> 405]
H4-moving   [1: 420 -- 5 --> 425]
```

Using interval variables with intensities: house building with resource calendars

This section describes how to model and solve a problem using interval variables with intensities.

In this section

Overview
Describes how to model and solve a problem using interval variables with intensities.

Describe
Describes the first stage in finding a solution to a house building problem with resource calendars.

Model
Describes the second stage in finding a solution to a house building problem with resource calendars.

Solve
Describes the third stage in finding a solution to a house building problem with resource calendars.

Review exercises
Includes the review exercises.

Complete program
Lists the location of the complete house building with resource calenders program.

Overview

In this lesson, you will learn how to:

- use the class `IloNumToNumStepFunction`;
- use an alternative constructor of the specialized constraint class `IloNoOverlap`;
- use intensity expressions;
- use the functions `IloForbidStart` and `IloForbidEnd`.

You will learn how to model and solve a house building problem, a problem of scheduling the tasks involved in building multiple houses in such a manner that minimizes the overall completion date of the houses. Some tasks must necessarily take place before other tasks, and each task has a predefined size. Moreover, there are two workers, each of whom must perform a given subset of the necessary tasks. Each worker has a calendar detailing the days on which he does not work, such as weekends and holidays. On a worker's day off, he does no work on his tasks. His tasks may not be scheduled to start or end on these days. Tasks that are in process by the worker are suspended during his days off. To find a solution to this problem using IBM® ILOG® CP Optimizer, you will use the three-stage method: describe, model and solve.

Describe

The problem consists of assigning start dates to a set of tasks in such a way that the schedule satisfies temporal constraints and minimizes an expression. The objective for this problem is to minimize the overall completion date.

For each task type in the house building project, the following table shows the duration of the task in days along with the tasks that must be finished before the task can start. In addition, each type of task can be performed by a given one of the two workers, Jim and Joe. A worker can only work on one task at a time; a task may be suspended during a worker's days off, but may not be interrupted by another task.

House construction tasks

Task	Duration	Worker	Preceding tasks
masonry	35	Joe	
carpentry	15	Joe	masonry
plumbing	40	Jim	masonry
ceiling	15	Jim	masonry
roofing	5	Joe	carpentry
painting	10	Jim	ceiling
windows	5	Jim	roofing
facade	10	Joe	roofing, plumbing
garden	5	Joe	roofing, plumbing
moving	5	Jim	windows, facade, garden, painting

Solving the problem consists of determining starting dates for the tasks such that the overall completion date is minimized.

Step 1: Describe the problem

The first step in modeling and solving a problem is to write a natural language description of the problem, identifying the decision variables and the constraints on these variables.

Write a natural language description of this problem. Answer these questions:

- What is the known information in this problem?
- What are the decision variables or unknowns in this problem?
- What are the constraints on these variables?
- What is the objective?

Discussion

What is the known information in this problem?

- There are five houses to be built by two workers. For each house, there are ten house building tasks, each with a given size. For each task, there is a list of tasks that must be completed before the task can start. Each task must be performed by a given worker, and each worker has a calendar listing his days off.

What are the decision variables or unknowns in this problem?

- The unknowns are the start and end times of tasks which also determine the overall completion time. The actual length of a task depends on its position in time and on the calendar of the associated worker.

What are the constraints on these variables?

- There are constraints that specify that a particular task may not begin until one or more given tasks have been completed. In addition, there are constraints that specify that a worker can be assigned to only one task at a time. A task cannot start or end during the associated worker's days off.

What is the objective?

- The objective is to minimize the overall completion date.

Model

After you have written a description of your problem, you can use IBM® ILOG® Concert Technology classes to model it.

Step 2: Open the example file

Open the example file `YourCPHome/examples/tutorial/cpp/sched_calendar_partial.cpp` in your development environment. This file is a program that is only partially completed. You will fill in the blanks in each step in this lesson. At the end, you will have completed the program code, and you can compile and run the program.

In this lesson, you include the header file `<ilcp/cp.h>`. To catch exceptions that may be thrown, you use a `try/catch` block. The code for creating the environment and model and for printing out the solution found by the CP Optimizer engine is provided.

In addition, the data related to the tasks, such as the tasks (`Tasks`), the number of tasks (`NbTasks`), the names of the tasks (`TaskNames`) and sizes of the tasks (`TaskDurations`), are provided.

After you create an environment and a model, you need to define the decision variables and add the constraints and objective to the model. Since the requirements for each of the five houses are similar, you use a function, `MakeHouse`, to create the decision variables, constraints and costs associated with a house. Information about individual houses that must be shared with the main function includes the expressions needed to create the objective function and the set of tasks associated with each worker. This set of tasks is needed in order to post the no overlap constraints on a worker's tasks. In order to display the results of the optimization, it is also useful to maintain an array of all the interval variables.

To access this information, you create objects to be updated in the `MakeHouse` function. An array of task interval variables, `allTasks`, stores all the interval variables that are created. The cost expression involves the date at which moving is completed for each house; the integer expression array `ends` is used to store this information. You create an array of interval variables, `joeTasks`, to be filled by the `MakeHouse` function. A similar array is created for Jim. These two arrays are passed to the `MakeHouse` function which adds elements to these arrays as the variables are created.

Step 3: Declare the objects needed for MakeHouse

Add the following code after the comment `//Declare the objects needed for MakeHouse`

```
IloInt nbHouses = 5;
IloModel model(env);
IloIntExprArray ends(env);
IloIntervalVarArray allTasks(env);
IloIntervalVarArray joeTasks(env);
IloIntervalVarArray jimTasks(env);
```

You pass the model, the house identifier (a number in `0..nbHouses-1`), the array of expressions representing the completion dates of the houses, the array of all tasks and the arrays `joeTasks` and `jimTasks` as arguments to the `MakeHouse` function.

Step 4: Create the MakeHouse function

Add the following code after the comment `//Create the MakeHouse function`

```
void MakeHouse(IloModel model,
       IloInt id,
       IloIntExprArray ends,
       IloIntervalVarArray allTasks,
       IloIntervalVarArray joeTasks,
       IloIntervalVarArray jimTasks) {
```

Each house has a list of `NbTasks` that must be scheduled. Task `i`, where `i` is in `0..NbTasks -1`, has a size of `TaskDurations[i]` and the name `TaskNames[i]`. Using these, you build an array `tasks` of interval variables. As you create each interval variable, you also add it to the array `allTasks` that will be used to display the solution once the schedule has been determined.

Step 5: Create the interval variables

Add the following code after the comment `//Create the interval variables`

```
  char name[128];
  IloIntervalVarArray tasks(env, NbTasks);
  for (IloInt i=0; i<NbTasks; ++i) {
    sprintf(name, "H%ld-%s", id, TaskNames[i]);
    tasks[i] = IloIntervalVar(env, TaskDurations[i], name);
    allTasks.add(tasks[i]);
  }
```

The tasks of the house building project have precedence constraints that are added to the model.

Step 6: Add the precedence constraints

Add the following code after the comment `//Add the precendence constraints`

```
  model.add(IloEndBeforeStart(env, tasks[masonry],   tasks[carpentry]));
  model.add(IloEndBeforeStart(env, tasks[masonry],   tasks[plumbing]));
  model.add(IloEndBeforeStart(env, tasks[masonry],   tasks[ceiling]));
  model.add(IloEndBeforeStart(env, tasks[carpentry], tasks[roofing]));
  model.add(IloEndBeforeStart(env, tasks[ceiling],   tasks[painting]));
  model.add(IloEndBeforeStart(env, tasks[roofing],   tasks[windows]));
  model.add(IloEndBeforeStart(env, tasks[roofing],   tasks[facade]));
  model.add(IloEndBeforeStart(env, tasks[plumbing],  tasks[facade]));
  model.add(IloEndBeforeStart(env, tasks[roofing],   tasks[garden]));
  model.add(IloEndBeforeStart(env, tasks[plumbing],  tasks[garden]));
  model.add(IloEndBeforeStart(env, tasks[windows],   tasks[moving]));
```

```
model.add(IloEndBeforeStart(env, tasks[facade],    tasks[moving]));
model.add(IloEndBeforeStart(env, tasks[garden],    tasks[moving]));
model.add(IloEndBeforeStart(env, tasks[painting],  tasks[moving]));
```

Each of the tasks requires a particular worker. For expressing constraints related to workers, such as no overlap and calendar constraints, it is necessary to know the set of tasks a particular worker must perform. To maintain this information, you add the appropriate tasks to the arrays `joeTasks` and `jimTasks`.

Step 7: Add the tasks to workers

Add the following code after the comment `//Add the tasks to workers`

```
joeTasks.add(tasks[masonry]);
joeTasks.add(tasks[carpentry]);
jimTasks.add(tasks[plumbing]);
jimTasks.add(tasks[ceiling]);
joeTasks.add(tasks[roofing]);
jimTasks.add(tasks[painting]);
jimTasks.add(tasks[windows]);
joeTasks.add(tasks[facade]);
joeTasks.add(tasks[garden]);
jimTasks.add(tasks[moving]);
```

To model the cost of building the houses, you will need to determine the maximum completion date among the individual house projects. To access the expression representing the completion date of the house currently in consideration, you use the function `IloEndOf` on the last task in building a house (here, it is the moving task) and store this expression in the array `ends`.

Step 8: Add the cost expression

Add the following code after the comment `//Add the cost expression`

```
ends.add(IloEndOf(tasks[moving]));
```

This completes the `MakeHouse` function. In the main function, you now call the `MakeHouse` function, once for each house. At each call, additional elements are appended to the arrays `ends`, `allTasks`, `joeTasks` and `jimTasks`.

Step 9: Create the houses

Add the following code after the comment `//Create the houses`

```
  for (IloInt h=0; h<nbHouses; ++h) {
    MakeHouse(model, h, ends, allTasks, joeTasks, jimTasks);
```

```
  }
```

To add the constraint that a worker can perform only one task at a time, you constrain that the interval variables associated with that worker do not overlap in the solution. To do this, you can use the specialized constraint `IloNoOverlap`, but with a slightly different constructor than was used in *Using no overlap constraints on interval variables: house building with workers*.

This constructor is a shortcut that avoids the need to explicitly define the interval sequence variable when no additional constraints are required on the sequence variable. The first argument passed to this constructor of the class `IloNoOverlap` is the environment. The second argument is the array of interval variables that should not overlap. The third argument is an optional transition parameter. The final argument is an optional name used for debug and trace purposes. Here is a constructor:

```
IloNoOverlap(const IloEnv env,
             const IloIntervalVarArray a,
             const IloTransitionDistance ttime =0,
             const char* name=0);
```

You add to the model one no overlap constraint on the array of interval variables for each worker.

Step 10: Add the no overlap constraints

Add the following code after the comment `//Add the no overlap constraints`

```
  model.add(IloNoOverlap(env, joeTasks));
  model.add(IloNoOverlap(env, jimTasks));
```

To model the availability of a worker in regard to his days off, you first create a function that represents his intensity over time. You specify that this function has a range of [0..100], where the value 0 represents that the worker is not available and the value 100 represents that the worker is available in regard to his calendar.

Concert Technology provides the class `IloNumToNumStepFunction` to represent a step function that is defined everywhere on a given interval and can be used to model the intensity of a worker.

Note: **Step function**

Step functions are represented by the class `IloNumToNumStepFunction` in IBM ILOG Concert Technology.

Each interval *[x1, x2)* on which the function has the same value is called a *step*.

When two consecutive steps of the function have the same value, these steps are merged so that the function is always represented with the minimal number of steps.

The first argument passed to the constructor of the class `IloNumToNumStepFunction` is the environment. The second and third arguments define the interval on which the function is defined. The fourth argument is the default value of the function on the defined interval. The fifth argument is an optional name used for debug and trace purposes. Here is a constructor:

```
IloNumToNumStepFunction(const IloEnv env,
                        IloNum xmin = -IloInfinity,
                        IloNum xmax =  IloInfinity,
                        IloNum dval =  0.0,
                        const char* name = 0);
```

This constructor creates a step defined everywhere on the interval `[xmin,xmax)` whose single step takes the value `dval`.

In this case, a worker's calendar is simplest to describe in terms of noting exceptions to being available to work, so you set the default value for his intensity to 100. Later you modify the subintervals associated with his days off to take the value 0. Based on knowledge of the problem, you can assume that it should take less than two years to build all five houses and thus define the function's interval as [0..2*365).

Step 11: Add the intensity step functions

Add the following code after the comment `//Add the intensity step functions`

```
IloNumToNumStepFunction joeCalendar(env, 0, 2*365, 100);
IloNumToNumStepFunction jimCalendar(env, 0, 2*365, 100);
```

To modify the step function to reflect a worker's days off, you set the step value of the subintervals associated with those days to the value 0. Since both workers have the weekends off, you iterate through the weeks and set the step value for day 5 and 6 of each week to 0 (For this problem, day 0 is the start of a week.). For the holidays, you also set the values of the associated intervals to zero.

Step 12: Add the weekends and holidays

Add the following code after the comment `//Add the weekends and holidays`

```
// Week-ends
for (IloInt w=0; w<2*52; ++w) {
  joeCalendar.setValue(5+(7*w), 7+(7*w), 0);
  jimCalendar.setValue(5+(7*w), 7+(7*w), 0);
```

```
}

// Holidays
joeCalendar.setValue(  5,  12, 0);
joeCalendar.setValue(124, 131, 0);
joeCalendar.setValue(215, 236, 0);
joeCalendar.setValue(369, 376, 0);
joeCalendar.setValue(495, 502, 0);
joeCalendar.setValue(579, 600, 0);
jimCalendar.setValue( 26,  40, 0);
jimCalendar.setValue(201, 225, 0);
jimCalendar.setValue(306, 313, 0);
jimCalendar.setValue(397, 411, 0);
jimCalendar.setValue(565, 579, 0);
```

To apply a worker's intensity function to all of his tasks, you use the function `IloIntervalVar::setIntensity`.

```
void IloIntervalVar::setIntensity(IloNumToNumStepFunction intensity,
                                  IloInt granularity =100);
```

This method sets the step function `intensity` as the intensity function of the invoking interval variable. Full intensity is represented by a value `granularity`. The argument `intensity` should be an integral `IloNumToNumStepFunction` with values in the range `[0..granularity]`. In this example, the intensity values are only 0 and 100, and the granularity is 100.

When an intensity function is set on an interval variable, the interval will automatically be prolonged if it overlaps any weekends and/or holidays. The size of the interval variable is the time spent at the house to process the task, not including the worker's days off. The length is the difference between the start and the end of the interval.

A task could still be scheduled to start or end on a day off. In this problem, a worker's tasks cannot start or end during the worker's days off. Concert Technology provides the constraints `IloForbidStart` and `IloForbidEnd` to model these types of restrictions.

Note: **Forbidden interval placement constraint**

With the specialized constraint `IloForbidStart`, you can create a constraint that specifies that an interval variable must not be scheduled to start at certain times.

The constraint takes an interval variable and a step function. If the interval variable is present in the solution, then it is constrained to not start at a time when the value of the step function is zero.

Concert Technology also provides `IloForbidEnd` and `IloForbidExtent`, which respectively constrain an interval variable to not end and not overlap where the associated step function is valued zero.

The first argument passed to the function `IloForbidStart` is the environment. The second argument is the interval variable on which you want to place the constraint. The third

argument is the step function that defines a set of forbidden values for the start of the interval variable: the interval variable cannot start at a point where the step function is 0.

Here is a function signature:

```
IloConstraint IloForbidStart (const IloEnv env,
                              const IloIntervalVar a
                              const IloNumToNumStepFunction func);
```

Step 13: Add the forbidden start and end constraints

Add the following code after the comment `//Add the forbidden start and end constraints`

```
  IloInt i;
  for (i=0; i<joeTasks.getSize(); ++i) {
    joeTasks[i].setIntensity(joeCalendar);
    model.add(IloForbidStart(env, joeTasks[i], joeCalendar));
    model.add(IloForbidEnd(env, joeTasks[i], joeCalendar));
  }
  for (i=0; i<jimTasks.getSize(); ++i) {
    jimTasks[i].setIntensity(jimCalendar);
    model.add(IloForbidStart(env, jimTasks[i], jimCalendar));
    model.add(IloForbidEnd(env, jimTasks[i], jimCalendar));
  }
```

The objective of this problem is to minimize the overall completion date (the completion date of the house that is completed last). To do this, you minimize the maximal expression in the array `ends`.

Step 14: Add the objective

Add the following code after the comment `//Add the objective`

```
  model.add(IloMinimize(env, IloMax(ends)));
```

Solve

You use an instance of the class `IloCP` to solve a problem expressed in a model. The constructor for `IloCP` takes an `IloModel` as its argument.

Step 15: Create an instance of IloCP

Add the following code after the comment `//Create an instance of IloCP`

```
   IloCP cp(model);
```

You now use the member function `IloCP::solve`, which solves the problem contained in the model by using constructive search and constraint propagation. The search for an optimal solution in this problem could potentiality take a long time, so you place a fail limit on the solve process. The search will stop when the fail limit is reached, even if optimality of the current best solution is not guaranteed.

Step 16: Search for a solution

Add the following code after the comment `//Search for a solution`

```
   cp.setParameter(IloCP::FailLimit, 10000);
   if (cp.solve()) {
```

The member function `IloCP::solve` returns a Boolean value of type `IloBool`. If a solution is found, the value `IloTrue` is returned.

After a solution has been found, you can use the member functions `IloCP::getObjValue` and `IloCP::domain` to examine the solution. The stream `IloAlgorithm::out` is the communication stream for general output.The code for displaying the solution has been provided for you:

```
      cp.out() << "Solution with objective " << cp.getObjValue() << ":" <<
std::endl;
      for (i=0; i<allTasks.getSize(); ++i) {
        cp.out() << cp.domain(allTasks[i]) << std::endl;
      }
```

Step 17: Compile and run the program

Compile and run the program. You should get the following results:

```
Solution with objective 638:
H0-masonry  [1: 0 -- (35)54 --> 54]
```

```
H0-carpentry[1: 301 -- (15)19 --> 320]
H0-plumbing [1: 77 -- (40)54 --> 131]
H0-ceiling  [1: 56 -- (15)19 --> 75]
H0-roofing  [1: 399 -- (5)5 --> 404]
H0-painting [1: 589 -- (10)14 --> 603]
H0-windows  [1: 498 -- (5)7 --> 505]
H0-facade   [1: 483 -- (10)12 --> 495]
H0-garden   [1: 441 -- (5)5 --> 446]
H0-moving   [1: 603 -- (5)7 --> 610]
H1-masonry  [1: 210 -- (35)68 --> 278]
H1-carpentry[1: 280 -- (15)19 --> 299]
H1-plumbing [1: 428 -- (40)56 --> 484]
H1-ceiling  [1: 316 -- (15)21 --> 337]
H1-roofing  [1: 392 -- (5)5 --> 397]
H1-painting [1: 526 -- (10)14 --> 540]
H1-windows  [1: 484 -- (5)7 --> 491]
H1-facade   [1: 511 -- (10)12 --> 523]
H1-garden   [1: 504 -- (5)5 --> 509]
H1-moving   [1: 631 -- (5)7 --> 638]
H2-masonry  [1: 105 -- (35)54 --> 159]
H2-carpentry[1: 364 -- (15)26 --> 390]
H2-plumbing [1: 337 -- (40)56 --> 393]
H2-ceiling  [1: 393 -- (15)35 --> 428]
H2-roofing  [1: 413 -- (5)5 --> 418]
H2-painting [1: 554 -- (10)28 --> 582]
H2-windows  [1: 582 -- (5)7 --> 589]
H2-facade   [1: 448 -- (10)12 --> 460]
H2-garden   [1: 462 -- (5)5 --> 467]
H2-moving   [1: 610 -- (5)7 --> 617]
H3-masonry  [1: 161 -- (35)47 --> 208]
H3-carpentry[1: 343 -- (15)19 --> 362]
H3-plumbing [1: 253 -- (40)63 --> 316]
H3-ceiling  [1: 232 -- (15)21 --> 253]
H3-roofing  [1: 420 -- (5)5 --> 425]
H3-painting [1: 540 -- (10)14 --> 554]
H3-windows  [1: 519 -- (5)7 --> 526]
H3-facade   [1: 469 -- (10)12 --> 481]
H3-garden   [1: 525 -- (5)5 --> 530]
H3-moving   [1: 624 -- (5)7 --> 631]
H4-masonry  [1: 56 -- (35)47 --> 103]
H4-carpentry[1: 322 -- (15)19 --> 341]
H4-plumbing [1: 133 -- (40)54 --> 187]
H4-ceiling  [1: 189 -- (15)43 --> 232]
H4-roofing  [1: 406 -- (5)5 --> 411]
H4-painting [1: 505 -- (10)14 --> 519]
H4-windows  [1: 491 -- (5)7 --> 498]
H4-facade   [1: 427 -- (10)12 --> 439]
H4-garden   [1: 532 -- (5)5 --> 537]
H4-moving   [1: 617 -- (5)7 --> 624]
```

You may notice that the results are displayed a little differently than in the previous lessons. Here both the size and length of an interval variable are reported as these are not equal in the solution to this problem.

The complete house building program can be viewed online in the `YourCPHome/examples/src/cpp/sched_calendar.cpp` file.

Review exercises

Includes the review exercises.

In this section

Exercises
Lists the review exercises.

Suggested answers
Provides the answers to the review exercises.

Exercises

For answers, see *Suggested answers*.

1. What is an intensity function?
2. What is the constraint `IloForbidStart` used to model?

Suggested answers

Exercise 1

What is an intensity function?

Suggested answer

An intensity function is an integer step function defined by the Concert Technology class `IloNumToNumStepFunction`. Using this class, you can set the intensity of a resource on intervals in time, where intensity is measured as a percentage of full intensity, valued at `granularity`, a specified integer granularity.

Exercise 2

What is the constraint `IloForbidStart` used to model?

Suggested answer

An `IloForbidStart` constraint is used to restrict an interval variable from starting in certain intervals. These intervals can be modeled using a step function, where the forbidden intervals are steps at which the function value is zero.

Complete program

The complete house building program can be viewed online in the file `YourCPHome/examples/src/cpp/sched_calendar.cpp`.

Results

```
Solution with objective 638:
H0-masonry  [1: 0 -- (35)54 --> 54]
H0-carpentry[1: 301 -- (15)19 --> 320]
H0-plumbing [1: 77 -- (40)54 --> 131]
H0-ceiling  [1: 56 -- (15)19 --> 75]
H0-roofing  [1: 399 -- (5)5 --> 404]
H0-painting [1: 589 -- (10)14 --> 603]
H0-windows  [1: 498 -- (5)7 --> 505]
H0-facade   [1: 483 -- (10)12 --> 495]
H0-garden   [1: 441 -- (5)5 --> 446]
H0-moving   [1: 603 -- (5)7 --> 610]
H1-masonry  [1: 210 -- (35)68 --> 278]
H1-carpentry[1: 280 -- (15)19 --> 299]
H1-plumbing [1: 428 -- (40)56 --> 484]
H1-ceiling  [1: 316 -- (15)21 --> 337]
H1-roofing  [1: 392 -- (5)5 --> 397]
H1-painting [1: 526 -- (10)14 --> 540]
H1-windows  [1: 484 -- (5)7 --> 491]
H1-facade   [1: 511 -- (10)12 --> 523]
H1-garden   [1: 504 -- (5)5 --> 509]
H1-moving   [1: 631 -- (5)7 --> 638]
H2-masonry  [1: 105 -- (35)54 --> 159]
H2-carpentry[1: 364 -- (15)26 --> 390]
H2-plumbing [1: 337 -- (40)56 --> 393]
H2-ceiling  [1: 393 -- (15)35 --> 428]
H2-roofing  [1: 413 -- (5)5 --> 418]
H2-painting [1: 554 -- (10)28 --> 582]
H2-windows  [1: 582 -- (5)7 --> 589]
H2-facade   [1: 448 -- (10)12 --> 460]
H2-garden   [1: 462 -- (5)5 --> 467]
H2-moving   [1: 610 -- (5)7 --> 617]
H3-masonry  [1: 161 -- (35)47 --> 208]
H3-carpentry[1: 343 -- (15)19 --> 362]
H3-plumbing [1: 253 -- (40)63 --> 316]
H3-ceiling  [1: 232 -- (15)21 --> 253]
H3-roofing  [1: 420 -- (5)5 --> 425]
H3-painting [1: 540 -- (10)14 --> 554]
H3-windows  [1: 519 -- (5)7 --> 526]
H3-facade   [1: 469 -- (10)12 --> 481]
H3-garden   [1: 525 -- (5)5 --> 530]
H3-moving   [1: 624 -- (5)7 --> 631]
H4-masonry  [1: 56 -- (35)47 --> 103]
H4-carpentry[1: 322 -- (15)19 --> 341]
H4-plumbing [1: 133 -- (40)54 --> 187]
```

```
H4-ceiling  [1: 189 -- (15)43 --> 232]
H4-roofing  [1: 406 -- (5)5 --> 411]
H4-painting [1: 505 -- (10)14 --> 519]
H4-windows  [1: 491 -- (5)7 --> 498]
H4-facade   [1: 427 -- (10)12 --> 439]
H4-garden   [1: 532 -- (5)5 --> 537]
H4-moving   [1: 617 -- (5)7 --> 624]
```

Using cumulative functions: house building with budget and resource pools

This section describes how to model and solve a problem using cumulative functions.

In this section

Overview
Describes how to model and solve a problem using cumulative functions.

Describe
Describes the first stage in finding a solution to a house building problem with budget and resource pools.

Model
Describes the second stage in finding a solution to a house building problem with budget and resource pools.

Solve
Describes the third stage in finding a solution to a house building problem with budget and resource pools.

Review exercises
Includes the review exercises.

Complete program
Lists the location of the complete house building with budget and resource pools program.

Overview

In this lesson, you will learn how to:

- use the class `IloCumulFunctionExpr`;
- use the functions `IloPulse`, `IloStepAtStart` and `IloStepAtEnd`.

You will learn how to model and solve a house building problem, a problem of scheduling the tasks involved in building multiple houses in such a manner that minimizes the overall completion date of the houses. Some tasks must necessarily take place before other tasks, and each task has a predefined size. Moreover, there are three workers, and each task requires any one of the three workers. A worker can be assigned to at most one task at a time. In addition, there is a cash budget with a starting balance. Each task requires a certain amount of the cash budget at the start of the task, and the cash balance is increased every sixty days. To find a solution to this problem using IBM® ILOG® CP Optimizer, you will use the three-stage method: describe, model and solve.

Describe

The problem consists of assigning start dates to a set of tasks in such a way that the schedule satisfies temporal constraints and minimizes an expression. The objective for this problem is to minimize the overall completion date. Each task requires 200 dollars per day of the task, payable at the start of the task. Every sixty days, starting at day 0, the amount of 30,000 dollars is added to the cash balance.

For each task type in the house building project, the following table shows the duration of the task in days along with the tasks that must be finished before the task can start. Each task consumes any one of the three workers. A worker can only work on one task at a time; each task, once started, may not be interrupted.

House construction tasks

Task	Duration	Preceding Tasks
masonry	35	
carpentry	15	masonry
plumbing	40	masonry
ceiling	15	masonry
roofing	5	carpentry
painting	10	ceiling
windows	5	roofing
facade	10	roofing, plumbing
garden	5	roofing, plumbing
moving	5	windows, facade, garden, painting

There is an earliest starting date for each of the five houses that must be built.

House construction earliest starting date

House	Earliest starting date
0	31
1	0
2	90
3	120
4	90

Solving the problem consists of determining starting dates for the tasks such that the overall completion date is minimized.

Step 1: Describe the problem

The first step in modeling and solving a problem is to write a natural language description of the problem, identifying the decision variables and the constraints on these variables.

Write a natural language description of this problem. Answer these questions:

- What is the known information in this problem?
- What are the decision variables or unknowns in this problem?
- What are the constraints on these variables?
- What is the objective?

Discussion

What is the known information in this problem?

- There are five houses to be built by three workers. For each house, there are ten house building tasks, each with a given size and cost. For each task, there is a list of tasks that must be completed before the task can start. There is a starting cash balance of a given amount, and each sixty days the cash balance is increased by a given amount.

What are the decision variables or unknowns in this problem?

- The unknowns are the dates that the tasks will start. Once starting dates have been fixed, the overall completion date will also be fixed.

What are the constraints on these variables?

- There are constraints that specify that a particular task may not begin until one or more given tasks have been completed. Each task requires any one of the three workers. In addition, there are constraints that specify that a worker can be assigned to only one task at a time. Before a task can start, the cash balance must be large enough to pay the cost of the task.

What is the objective?

- The objective is to minimize the overall completion date.

Model

After you have written a description of your problem, you can use IBM® ILOG® Concert Technology classes to model it.

Step 2: Open the example file

Open the example file `YourCPHome/examples/tutorial/cpp/sched_cumul_partial.cpp` in your development environment. This file is a program that is only partially completed. You will fill in the blanks in each step in this lesson. At the end, you will have completed the program code, and you can compile and run the program.

In this lesson, you include the header file `<ilcp/cp.h>`. To catch exceptions that may be thrown, you use a `try/catch` block. The code for creating the environment and model and for printing out the solution found by the CP Optimizer engine is provided.

In addition, the data related to the tasks, such as the tasks (`Tasks`), the number of tasks (`NbTasks`), the names of the tasks (`TaskNames`) and sizes of the tasks (`TaskDurations`), are provided. The number of workers (`NbWorkers`) is also provided.

After you create an environment and a model, you need to define the decision variables and add the constraints and objective to the model. Since the requirements for each of the five houses are similar, you use a function, `MakeHouse`, to create the decision variables, constraints and costs associated with each house. Information about individual houses that must be shared with the main function includes the expressions needed to create the objective function and information about worker usage and the cash balance. In order to display the results of the optimization, it is also useful to maintain an array of all the interval variables.

To access this information, you create objects to be updated in the `MakeHouse` function. An array of task interval variables, `allTasks`, stores all the interval variables that are created. The cost expression involves the date at which moving is completed for each house, and the integer expression array `ends` is used to store this information. In addition, the expressions used to represent worker usage and the cash balance are included in the global information that is updated in each call to the `MakeHouse` function.

Since the workers are equivalent in this problem, it is better to represent them as one pool of workers instead of as individual workers with no overlap constraints as was done in the earlier examples. This representation removes symmetry. The expression representing usage of the pool of workers can be modified by the interval variables that require a worker.

To model both the limited number of workers and the limited budget, you need to represent the sum of individual contributions of interval variables. In the case of the cash budget, some tasks consume some of the budget at the start. In the case of the workers, a task requires a worker only for the duration of the task. Concert Technology provides the class `IloCumulFunctionExpr` to represent the sum of individual contributions of interval variables.

Note: **Cumulative function expression**

A cumulative function, represented in IBM ILOG Concert Technology by `IloCumulFunctionExpr`, can be used to model a resource usage function over

time. This function can be computed as a sum of interval variable demands on a resource over time.

An interval usually increases the cumulated resource usage function at its start time and decreases it when it releases the resource at its end time (pulse function).

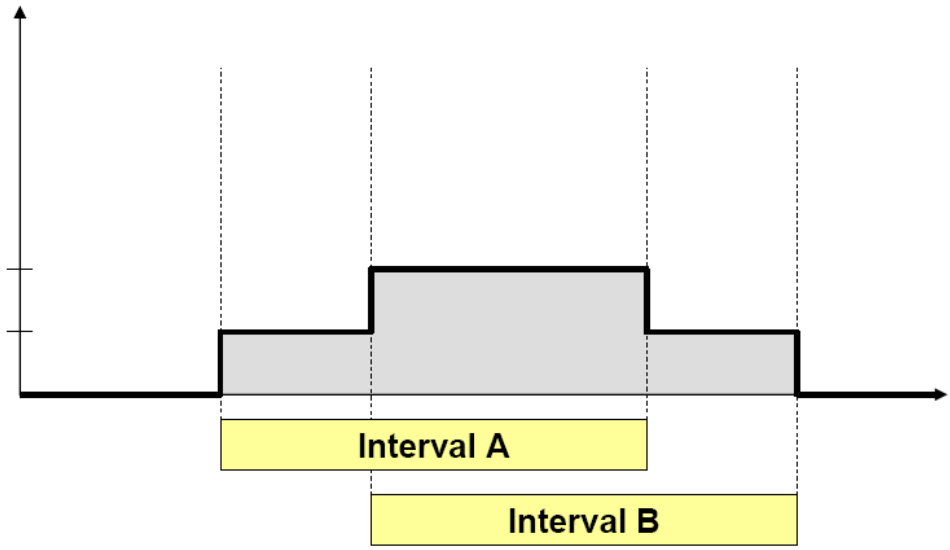
For resources that can be produced and consumed by activities (for instance the contents of an inventory or a tank), the resource level can also be described as a function of time. A production activity will increase the resource level at the start or end time of the activity whereas a consuming activity will decrease it. The cumulated contribution of activities on the resource can be represented by a function of time, and constraints can be posted on this function (for instance, a maximal or a safety level).

A cumulative function expression can be computed as a sum of the following types of elementary demands:

- `IloStep`, which increases or decreases the level of the function by a given amount at a given time;
- `IloPulse`, which increases or decreases the level of the function by a given amount for the length of a given interval variable or fixed interval;
- `IloStepAtStart`, which increases or decreases the level of the function by a given amount at the start of a given interval variable;
- `IloStepAtEnd`, which increases or decreases the level of the function by a given amount at the end of a given interval variable.

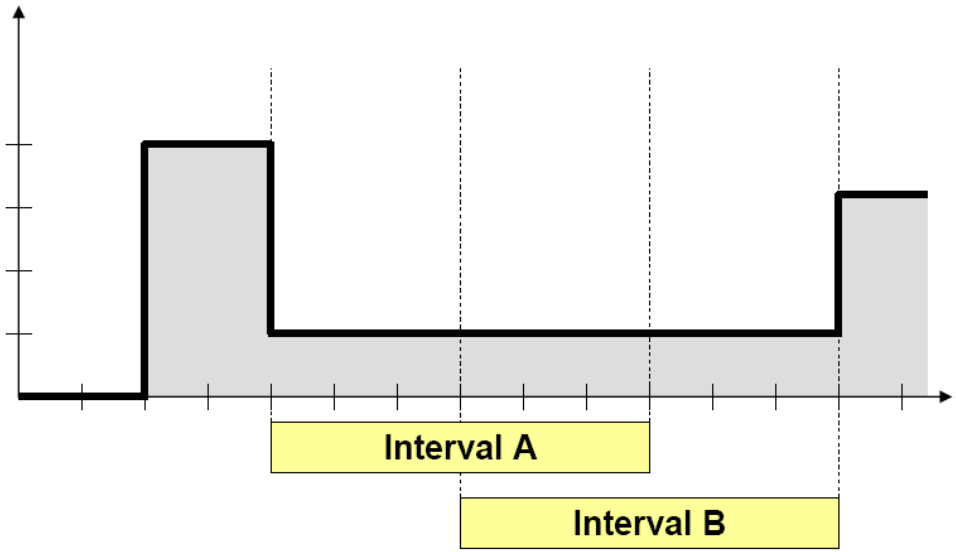
A cumulative function expression can be constrained to model limited resource capacity by constraining that the function be <= the capacity.

To illustrate, consider a cumulative resource usage function that measures how much of a resource is being used. There are two intervals, A and B, bound in time, and each interval increases the cumulative function expression by one unit over its duration. For each interval, this modification to the cumulative resource usage function can be made by incrementing the cumulative function with the elementary function `IloPulse`, created with the interval and the given amount. Given this, the function would take the profile as in the figure “Pulse on Cumulative Function Expression.”



Pulse on Cumulative Function Expression

As another example, consider a function measuring a consumable resource, similar to the budget resource. Consider that the level of the resource is zero, until time 2 when the value is increased to 4; modeled by modifying the cumulative function with the elementary cumul function `IloStep` at time 2. There are two intervals, A and B, fixed in time. Interval A decreases the level of the resource by 3 at the start of the interval, modeled by applying `IloStepAtStart`, created with Interval A and the value (3), to the cumulative function. Interval B increases the level of the resource by 2 at the end of the interval, modeled by applying `IloStepAtEnd`, created with Interval B and the value (2), to the cumulative function for the interval. Given this, the function would take the profile as in the figure “Step on Cumulative Function Expression.”.



Step on Cumulative Function Expression

For the house building problem, you create two cumulative function expression objects, one to represent the usage of the workers and the other to represent the cash balance. The constructor of the class `IloCumulFunctionExpr` takes one argument, the environment.

Step 3: Declare the objects needed for MakeHouse

Add the following code after the comment `//Declare the objects needed for MakeHouse`

```
IloCumulFunctionExpr workersUsage(env);
IloCumulFunctionExpr cash(env);
IloIntExprArray ends(env);
IloIntervalVarArray allTasks(env);
```

To set the increases to the cash balance, you use the function `IloStep`, which can be used to increment or decrement the cumulative function expression by a fixed amount on a given date. The first argument passed to this function is the environment. The second argument is the date at which the function should be modified. The third argument is the amount by which the function should be changed; this must be a non-negative value.

```
IloCumulFunctionExpr IloStep(const IloEnv env, IloInt t, IloInt v);
```

For setting the increases to the cash balance, the cumulative function expression for the cash balance should be incremented by the appropriate amount, 30,000 dollars, every 60 days, starting at day 0.

Step 4: Add the cash payment expression

Add the following code after the comment `//Add the cash payment expression`

```
   for (IloInt p=0; p<5; ++p)
     cash += IloStep(env, 60*p, 30000);
```

You need to pass the model, the house identifier, the earliest start date of the house, the cumulative function expressions for the worker usage and cash balance, the array of expressions representing the completion dates of the houses and the array of all the tasks as arguments to the `MakeHouse` function.

Step 5: Create the MakeHouse function

Add the following code after the comment `//Create the MakeHouse function`

```
void MakeHouse(IloModel model,
               IloInt id,
               IloInt rd,
               IloCumulFunctionExpr& workersUsage,
               IloCumulFunctionExpr& cash,
               IloIntExprArray ends,
               IloIntervalVarArray allTasks) {
```

Each house has a list of `NbTasks` that must be scheduled. Task `i`, where `i` is in `0..NbTasks-1`, has a size of `TaskDurations[i]` and the name `TaskNames[i]`. Using these, you build an array `tasks` of interval variables. As you create each interval variable, you also add it to the array `allTasks` that will be used to display the solution once the schedule has been computed.

Each task also requires one worker from the start to the end of the task interval. To represent the fact that a worker is required for the task, you modify the cumulative function expression, `workerUsage`. Since it is not known when a task will begin or end, you use the function `IloPulse`. The first argument passed to this function is the interval during which the function should be modified. The second argument is the amount by which the function should be changed; this must be a non-negative value.

```
 IloCumulFunctionExpr IloPulse(const IloIntervalVar a, IloInt v);
```

Moreover, each task requires a payment equal to 200 dollars a day for the length of the task, payable at the start of the task. For each task, you use the function `IloStepAtStart` to adjust the cash balance cumulative function expression.

Step 6: Create the interval variables

Add the following code after the comment `//Create the interval variables`

```
char name[128];
IloIntervalVarArray tasks(env, NbTasks);
for (IloInt i=0; i<NbTasks; ++i) {
  sprintf(name, "H%ld-%s", id, TaskNames[i]);
  IloIntervalVar task(env, TaskDurations[i], name);
  tasks[i] = task;
  allTasks.add(task);
  workersUsage += IloPulse(task, 1);
  cash -= IloStepAtStart(task, 200 * TaskDurations[i]);
}
```

The tasks have precedence constraints that are added to the model. Moreover, each house has an earliest starting date.

Step 7: Add the temporal constraints

Add the following code after the comment `//Add the temporal constraints`

```
tasks[masonry].setStartMin(rd);
model.add(IloEndBeforeStart(env, tasks[masonry], tasks[carpentry]));
model.add(IloEndBeforeStart(env, tasks[masonry], tasks[plumbing]));
model.add(IloEndBeforeStart(env, tasks[masonry], tasks[ceiling]));
model.add(IloEndBeforeStart(env, tasks[carpentry], tasks[roofing]));
model.add(IloEndBeforeStart(env, tasks[ceiling], tasks[painting]));
model.add(IloEndBeforeStart(env, tasks[roofing], tasks[windows]));
model.add(IloEndBeforeStart(env, tasks[roofing], tasks[facade]));
model.add(IloEndBeforeStart(env, tasks[plumbing], tasks[facade]));
model.add(IloEndBeforeStart(env, tasks[roofing], tasks[garden]));
model.add(IloEndBeforeStart(env, tasks[plumbing], tasks[garden]));
model.add(IloEndBeforeStart(env, tasks[windows], tasks[moving]));
model.add(IloEndBeforeStart(env, tasks[facade], tasks[moving]));
model.add(IloEndBeforeStart(env, tasks[garden], tasks[moving]));
model.add(IloEndBeforeStart(env, tasks[painting], tasks[moving]));
```

To model the cost of building the houses, you will need to determine the maximum completion date among the individual house projects. To determine the expression representing the completion date of the house currently in consideration, you use the function `IloEndOf` on the last task in building a house (here, it is the moving task) and store this expression in the array `ends`.

Step 8: Add the objective expression

Add the following code after the comment `//Add the objective expression`

```
ends.add(IloEndOf(tasks[moving]));
```

This completes the `MakeHouse` function. In the main function, you now call the `MakeHouse` function, once for each house. At each call, additional elements are appended to the arrays `ends` and `allTasks`, and the expressions for worker usage and the cash balance are modified.

Step 9: Create the houses

Add the following code after the comment `//Create the houses`

```
MakeHouse(model, 0,  31, workersUsage, cash, ends, allTasks);
MakeHouse(model, 1,   0, workersUsage, cash, ends, allTasks);
MakeHouse(model, 2,  90, workersUsage, cash, ends, allTasks);
MakeHouse(model, 3, 120, workersUsage, cash, ends, allTasks);
MakeHouse(model, 4,  90, workersUsage, cash, ends, allTasks);
```

To add the constraint that the cash balance must remain positive, you constrain the cumulative function expression representing the cash to be non-negative. (Note that by default the function is constrained to be non-negative, but the cumulative expression function must be added to the model in order for the optimizer to take it into account during the search.)

Step 10: Add the cash balance constraint

Add the following code after the comment `//Add the cash balance constraint`

```
model.add(0 <= cash);
```

To add the constraint that there can be only three workers working at a given time, you constrain the cumulative function expression representing worker usage to be no greater than the value `NbWorkers`.

Step 11: Add the worker usage constraint

Add the following code after the comment `//Add the worker usage constraint`

```
model.add(workersUsage <= NbWorkers);
```

The objective of this problem is to minimize the overall completion date (the completion date of the house that is completed last). To do this, you minimize the maximal expression in the array `ends`.

Step 12: Add the objective

Add the following code after the comment `//Add the objective`

```
model.add(IloMinimize(env, IloMax(ends)));
```

Solve

You use an instance of the class `IloCP` to solve a problem expressed in a model. The constructor for `IloCP` takes an `IloModel` as its argument.

Step 13: Create an instance of IloCP

Add the following code after the comment `//Create an instance of IloCP`

```
   IloCP cp(model);
```

You now use the member function `IloCP::solve`, which solves the problem contained in the model by using constructive search and constraint propagation. The search for an optimal solution in this problem could potentiality take a long time, so you place a fail limit on the solve process. The search will stop when the fail limit is reached, even if optimality of the current best solution is not guaranteed.

Step 14: Search for a solution

Add the following code after the comment `//Search for a solution`

```
   cp.setParameter(IloCP::FailLimit, 10000);
   if (cp.solve()) {
```

The member function `IloCP::solve` returns a Boolean value of type `IloBool`. If a solution is found, the value `IloTrue` is returned.

After a solution has been found, you can use the member functions `IloCP::getObjValue` and `IloCP::domain` to examine the solution. The stream `IloAlgorithm::out` is the communication stream for general output. The code for displaying the solution has been provided for you:

```
      cp.out() << "Solution with objective " << cp.getObjValue() << ":" <<
std::endl;
      for (IloInt i=0; i<allTasks.getSize(); ++i) {
        cp.out() << cp.domain(allTasks[i]) << std::endl;
      }
```

Step 15: Compile and run the program

Compile and run the program. You should get the following results:

```
Solution with objective 285:
H0-masonry  [1: 31 -- 35 --> 66]
```

```
H0-carpentry[1: 66 -- 15 --> 81]
H0-plumbing [1: 81 -- 40 --> 121]
H0-ceiling  [1: 70 -- 15 --> 85]
H0-roofing  [1: 85 -- 5 --> 90]
H0-painting [1: 110 -- 10 --> 120]
H0-windows  [1: 95 -- 5 --> 100]
H0-facade   [1: 255 -- 10 --> 265]
H0-garden   [1: 240 -- 5 --> 245]
H0-moving   [1: 270 -- 5 --> 275]
H1-masonry  [1: 0 -- 35 --> 35]
H1-carpentry[1: 35 -- 15 --> 50]
H1-plumbing [1: 50 -- 40 --> 90]
H1-ceiling  [1: 35 -- 15 --> 50]
H1-roofing  [1: 50 -- 5 --> 55]
H1-painting [1: 60 -- 10 --> 70]
H1-windows  [1: 55 -- 5 --> 60]
H1-facade   [1: 100 -- 10 --> 110]
H1-garden   [1: 90 -- 5 --> 95]
H1-moving   [1: 280 -- 5 --> 285]
H2-masonry  [1: 120 -- 35 --> 155]
H2-carpentry[1: 155 -- 15 --> 170]
H2-plumbing [1: 195 -- 40 --> 235]
H2-ceiling  [1: 205 -- 15 --> 220]
H2-roofing  [1: 195 -- 5 --> 200]
H2-painting [1: 265 -- 10 --> 275]
H2-windows  [1: 270 -- 5 --> 275]
H2-facade   [1: 240 -- 10 --> 250]
H2-garden   [1: 250 -- 5 --> 255]
H2-moving   [1: 275 -- 5 --> 280]
H3-masonry  [1: 121 -- 35 --> 156]
H3-carpentry[1: 180 -- 15 --> 195]
H3-plumbing [1: 195 -- 40 --> 235]
H3-ceiling  [1: 180 -- 15 --> 195]
H3-roofing  [1: 200 -- 5 --> 205]
H3-painting [1: 255 -- 10 --> 265]
H3-windows  [1: 250 -- 5 --> 255]
H3-facade   [1: 255 -- 10 --> 265]
H3-garden   [1: 265 -- 5 --> 270]
H3-moving   [1: 275 -- 5 --> 280]
H4-masonry  [1: 90 -- 35 --> 125]
H4-carpentry[1: 125 -- 15 --> 140]
H4-plumbing [1: 140 -- 40 --> 180]
H4-ceiling  [1: 180 -- 15 --> 195]
H4-roofing  [1: 156 -- 5 --> 161]
H4-painting [1: 245 -- 10 --> 255]
H4-windows  [1: 161 -- 5 --> 166]
H4-facade   [1: 240 -- 10 --> 250]
H4-garden   [1: 265 -- 5 --> 270]
H4-moving   [1: 275 -- 5 --> 280]
```

The complete program can be viewed online in the `YourCPHome/examples/src/cpp/sched_cumul.cpp` file.

Review exercises

Includes the review exercises.

In this section

Exercises
Lists the review exercises.

Suggested answers
Provides the answers to the review exercises.

Exercises

For answers, see *Suggested answers*.

1. What is a cumulative function?
2. From what elements can a cumulative function be computed?

Suggested answers

Exercise 1

What is a cumulative function?

Suggested answer

An instance of the class `IloCumulFunctionExpr` represents the sum of individual contributions of interval variables. An interval usually increases the cumulated resource usage function at its start time and decreases it when it releases the resource at its end time (pulse function).

For resources that can be produced and consumed by activities (for instance the contents of an inventory or a tank), the resource level can also be described as a function of time. Production activities will increase the resource level whereas consuming activities will decrease it. The cumulated contribution of activities on the resource can be represented by a function of time, and constraints can be posted on this function (for instance, a maximal or a safety level).

Exercise 2

From what elements can a cumulative function be computed?

Suggested answer

A cumulative function expression can be computed as a sum of the following types of elementary demands:

- `IloStep`, which modifies the level of the function by a given amount at a given time;
- `IloPulse`, which modifies the level of the function by a given amount for the length of a given interval variable;
- `IloStepAtStart`, which modifies the level of the function by a given amount at the start of a given interval variable;
- `IloStepAtEnd`, which modifies the level of the function by a given amount at the end of a given interval variable.

Complete program

The complete house building can be viewed online in the file `YourCPHome/examples/src/cpp/sched_cumul.cpp`.

Results

```
Solution with objective 285:
H0-masonry  [1: 31 -- 35 --> 66]
H0-carpentry[1: 66 -- 15 --> 81]
H0-plumbing [1: 81 -- 40 --> 121]
H0-ceiling  [1: 70 -- 15 --> 85]
H0-roofing  [1: 85 -- 5 --> 90]
H0-painting [1: 110 -- 10 --> 120]
H0-windows  [1: 95 -- 5 --> 100]
H0-facade   [1: 255 -- 10 --> 265]
H0-garden   [1: 240 -- 5 --> 245]
H0-moving   [1: 270 -- 5 --> 275]
H1-masonry  [1: 0 -- 35 --> 35]
H1-carpentry[1: 35 -- 15 --> 50]
H1-plumbing [1: 50 -- 40 --> 90]
H1-ceiling  [1: 35 -- 15 --> 50]
H1-roofing  [1: 50 -- 5 --> 55]
H1-painting [1: 60 -- 10 --> 70]
H1-windows  [1: 55 -- 5 --> 60]
H1-facade   [1: 100 -- 10 --> 110]
H1-garden   [1: 90 -- 5 --> 95]
H1-moving   [1: 280 -- 5 --> 285]
H2-masonry  [1: 120 -- 35 --> 155]
H2-carpentry[1: 155 -- 15 --> 170]
H2-plumbing [1: 195 -- 40 --> 235]
H2-ceiling  [1: 205 -- 15 --> 220]
H2-roofing  [1: 195 -- 5 --> 200]
H2-painting [1: 265 -- 10 --> 275]
H2-windows  [1: 270 -- 5 --> 275]
H2-facade   [1: 240 -- 10 --> 250]
H2-garden   [1: 250 -- 5 --> 255]
H2-moving   [1: 275 -- 5 --> 280]
H3-masonry  [1: 121 -- 35 --> 156]
H3-carpentry[1: 180 -- 15 --> 195]
H3-plumbing [1: 195 -- 40 --> 235]
H3-ceiling  [1: 180 -- 15 --> 195]
H3-roofing  [1: 200 -- 5 --> 205]
H3-painting [1: 255 -- 10 --> 265]
H3-windows  [1: 250 -- 5 --> 255]
H3-facade   [1: 255 -- 10 --> 265]
H3-garden   [1: 265 -- 5 --> 270]
H3-moving   [1: 275 -- 5 --> 280]
H4-masonry  [1: 90 -- 35 --> 125]
H4-carpentry[1: 125 -- 15 --> 140]
H4-plumbing [1: 140 -- 40 --> 180]
```

```
H4-ceiling  [1: 180 -- 15 --> 195]
H4-roofing  [1: 156 -- 5 --> 161]
H4-painting [1: 245 -- 10 --> 255]
H4-windows  [1: 161 -- 5 --> 166]
H4-facade   [1: 240 -- 10 --> 250]
H4-garden   [1: 265 -- 5 --> 270]
H4-moving   [1: 275 -- 5 --> 280]
```

Using alternatives of interval variables: house building with worker allocation

The section describes how to model and solve a problem using optional interval variables.

In this section

Overview
Describes how to model and solve a problem using optional interval variables.

Describe
Describes the first stage in finding a solution to the house building problem with worker allocation requirements.

Model
Describes the second stage in finding a solution to the house building problem with worker allocation requirements.

Solve
Describes the third stage in finding a solution to the house building problem with worker allocation requirements.

Review exercises
Includes the review exercise.

Complete program
Lists the location of the complete house building with worker allocation problem.

Overview

In this lesson, you will learn how to:

- use the class `IloAlternative`;
- use the specialized constraint `IloPresenceOf`.

You will learn how to model and solve a house building problem, a problem of scheduling the tasks involved in building multiple houses. Some tasks must necessarily take place before other tasks, and each task has a predefined size. Each house has a maximal completion date. Moreover, there are three workers, and one of the three is required for each task. The three workers have varying levels of skills with regard to the various tasks; if a worker has no skill for a particular task, he may not be assigned to the task. For some pairs of tasks, if a particular worker performs one of the pair on a house, then the same worker must be assigned to the other of the pair for that house. The objective is to find a solution that maximizes the task associated skill levels of the workers assigned to the tasks. To find a solution to this problem using IBM® ILOG® CP Optimizer, you will use the three-stage method: describe, model and solve.

Describe

The problem consists of assigning start dates to a set of tasks in such a way that the schedule satisfies temporal constraints and minimizes an expression. The objective for this problem is to maximize the task associated skill levels of the workers assigned to the tasks.

For each task type in the house building project, the following table shows the duration of the task in days along with the tasks that must be finished before the task can start. A worker can only work on one task at a time; each task, once started, may not be interrupted.

House construction tasks

Task	Duration	Preceding tasks
masonry	35	
carpentry	15	masonry
plumbing	40	masonry
ceiling	15	masonry
roofing	5	carpentry
painting	10	ceiling
windows	5	roofing
facade	10	roofing, plumbing
garden	5	roofing, plumbing
moving	5	windows, facade, garden, painting

All of the houses must be completed by a deadline of day 318. There are three workers with varying skill levels in regard to the ten tasks. If a worker has a skill level of zero for a task, he may not be assigned to that type of task.

Worker-task skill levels

Task	Joe	Jack	Jim
masonry	9	5	0
carpentry	7	0	5
plumbing	0	7	0
ceiling	5	8	0
roofing	6	7	0
painting	0	9	6
windows	8	0	5
facade	5	5	0
garden	5	5	9
moving	6	0	8

For Jack, if he performs roofing or facade on a house, then he must perform the other on that house. For Jim, if he performs garden or moving on a house, then he must perform the other on that house. For Joe, if he performs masonry or carpentry on a house, then he must perform the other on that house. Also, if Joe performs carpentry or roofing on a house, then he must perform the other on that house.

Solving the problem consists of determining starting dates for the tasks and assigning a worker to each task such that sum of the skill levels used for the tasks is maximized.

Step 1: Describe the problem

The first step in modeling and solving a problem is to write a natural language description of the problem, identifying the decision variables and the constraints on these variables.

Write a natural language description of this problem. Answer these questions:

- What is the known information in this problem?
- What are the decision variables or unknowns in this problem?
- What are the constraints on these variables?
- What is the objective?

Discussion

What is the known information in this problem?

- There are five houses to be built by three workers. For each house, there are ten house building tasks, each with a given size. For each task, there is a list of tasks that must be completed before the task can start. Each worker has a skill level associated with each task. There is an overall deadline for the work to be completed on the five houses.

What are the decision variables or unknowns in this problem?

- The unknowns are the dates that the tasks will start. Also, unknown is which worker will be assigned to each task.

What are the constraints on these variables?

- There are constraints that specify that a particular task may not begin until one or more given tasks have been completed. In addition, there are constraints that specify that each task must have one worker assigned to it, that a worker can be assigned to only one task at a time and that a worker can be assigned only to tasks for which he has some level of skill. There are pairs of tasks that if one for a house is done by a particular worker, then the other for that house must be done by the same worker.

What is the objective?

- The objective is to maximize the skill levels used.

Model

After you have written a description of your problem, you can use IBM® ILOG® Concert Technology classes to model it.

Step 2: Open the example file

Open the example file `YourCPHome/examples/tutorial/cpp/sched_optional_partial.cpp` in your development environment. This file is a program that is only partially completed. You will fill in the blanks in each step in this lesson. At the end, you will have completed the program code, and you can compile and run the program.

In this lesson, you include the header file `<ilcp/cp.h>`. To catch exceptions that may be thrown, you use a `try/catch` block. The code for creating the environment and model and for printing out the solution found by the CP Optimizer engine is provided.

In addition, the data related to the tasks, such as the tasks (`Tasks`), the number of tasks (`NbTasks`), the names of the tasks (`TaskNames`) and sizes of the tasks (`TaskDurations`) and the data related to the workers, such as the workers (`Workers`), number of workers (`NbWorkers`) and the names of workers (`WorkerNames`), are provided. The matrix of skill levels (`SkillsMatrix`), one level for each worker/task pair and functions to return the skill level based on worker and task identifiers are also provided.

After you create an environment and a model, you need to define the decision variables and add the constraints and objective to the model. Since the requirements for each of the five houses are similar, you use a function `MakeHouse`. to create the decision variables, constraints and objective associated with each house. Information about individual houses that must be shared with the main function includes the expressions needed to create the objective function and, for each worker, the array of interval variables associated with the worker, in order to create the constraint regarding non overlapping intervals. In order to display the results of the optimization, it is also useful to maintain an array of all the interval variables.

To access this information, you create objects that will be updated in the `MakeHouse` function. The objective expression is represented by the integer expression `skill`. An array of task interval variables, `allTasks`, stores all the interval variables that are created. To maintain the interval variables related to each worker, you use an array of arrays of interval variables called `workerTasks`, that will be filled by the calls to `MakeHouse`.

Step 3: Declare the objects needed for MakeHouse

Add the following code after the comment `//Declare the objects needed for MakeHouse`

```
IloInt nbHouses = 5;
IloInt deadline = 318;
IloModel model(env);
IloIntExpr skill(env);
IloIntervalVarArray allTasks(env);
IloIntervalVarArray2 workerTasks(env, NbWorkers);
IloInt h, w;
for (w=0; w<NbWorkers; ++w)
```

```
      workerTasks[w] = IloIntervalVarArray(env);
```

You need to pass the model, the objective expression, the array of all tasks, the matrix of worker interval variables, the house identifier and the overall deadline as arguments to the `MakeHouse` function.

Step 4: Create the MakeHouse function

Add the following code after the comment `//Create the MakeHouse function`

```
void MakeHouse(IloModel model,
         IloIntExpr skill,
         IloIntervalVarArray allTasks,
         IloIntervalVarArray2 workerTasks,
         IloInt id,
         IloInt deadline) {
```

Each house has a list of `NbTasks` that must be scheduled. Task `i`, where `i` is in `0..NbTasks -1`, has a size of `TaskDurations[i]` and the name `TaskNames[i]`. Using these, you build an array `tasks` of interval variables.

Each task also requires one of the three workers from the start to the end of the interval. The worker assigned must have a non-negative level of skill on the task. To model this worker choice, you create an array of interval variables, one interval variable for each possible worker. You set these interval variables to be optional, so that each one may or may not be present in the solution. You also add each interval variable to the array `allTasks` that will be used to display the solution once the schedule has been determined. You also store these interval variables in a matrix called `TaskMatrix`, which is indexed on task and worker. This matrix will be used within the `MakeHouse` function in a later step.

If one of these interval variables is present in the solution, then its presence must be counted in the objective. Thus for each of these possible tasks, you increment the objective by the product of the skill level and the expression representing the presence of the time interval in the solution. The function `IloPresenceOf` takes the environment and an interval variable and returns a constraint that is true if the interval variable is present and false if it is absent.

To constrain the solution so that exactly one of this array of interval variables is to be present in the solution, you use the specialized constraint `IloAlternative`.

Note: **Alternative intervals constraint**

With the specialized constraint `IloAlternative`, you can create a constraint between an interval variable and a set of interval variables that specifies that if the given interval variable is present in the solution, then exactly one interval variable of the set is present in the solution.

In other words, consider an alternative constraint created with an interval variable `a` and an array of interval variables `bs`. If `a` is present in the solution, then exactly one

of the interval variables in `bs` will be present, and `a` starts and ends together with this chosen interval. If `a` is absent in the solution, then none of the interval variables in `bs` will be present.

The first argument passed to the constructor of the class `IloAlternative` is the environment. The second argument is the interval variable. The third argument is the array of interval variables that are the "alternatives". The final argument is an optional name used for debug and trace purposes. Here is a constructor:

```
IloAlternative(const IloEnv env,
               const IloIntervalVar a,
               const IloIntervalVarArray bs,
               const char* name =0);
```

Step 5: Create the interval variables

Add the following code after the comment `//Create the interval variables`

```
char name[128];
IloIntervalVarArray  tasks(env, NbTasks);
IloIntervalVarArray2 taskMatrix(env, NbTasks);

for (IloInt i=0; i<NbTasks; ++i) {
  sprintf(name, "H%ld-%s ", id, TaskNames[i]);
  tasks[i] = IloIntervalVar(env, TaskDurations[i], name);
  taskMatrix[i] = IloIntervalVarArray(env, NbWorkers);

  /* ALLOCATING TASKS TO WORKERS. */
  IloIntervalVarArray alttasks(env);
  for (IloInt w=0; w<NbWorkers; ++w) {
    if (HasSkill(w, i)) {
      sprintf(name, "H%ld-%s-%s ", id, TaskNames[i], WorkerNames[w]);
      IloIntervalVar wtask(env, TaskDurations[i], name);
      wtask.setOptional();
      alttasks.add(wtask);
      taskMatrix[i][w]=wtask;
      workerTasks[w].add(wtask);
      allTasks.add(wtask);
      /* DEFINING MAXIMIZATION OBJECTIVE. */
      skill += SkillLevel(w, i)*IloPresenceOf(env, wtask);
    }
  }
  model.add(IloAlternative(env, tasks[i], alttasks));
}
```

The tasks in the model have precedence constraints that are added to the model. Moreover, the "moving" task must be complete by the deadline.

Step 6: Add the temporal constraints

Add the following code after the comment `//Add the temporal constraints`

```
tasks[moving].setEndMax(deadline);
model.add(IloEndBeforeStart(env, tasks[masonry],   tasks[carpentry]));
model.add(IloEndBeforeStart(env, tasks[masonry],   tasks[plumbing]));
model.add(IloEndBeforeStart(env, tasks[masonry],   tasks[ceiling]));
model.add(IloEndBeforeStart(env, tasks[carpentry], tasks[roofing]));
model.add(IloEndBeforeStart(env, tasks[ceiling],   tasks[painting]));
model.add(IloEndBeforeStart(env, tasks[roofing],   tasks[windows]));
model.add(IloEndBeforeStart(env, tasks[roofing],   tasks[facade]));
model.add(IloEndBeforeStart(env, tasks[plumbing],  tasks[facade]));
model.add(IloEndBeforeStart(env, tasks[roofing],   tasks[garden]));
model.add(IloEndBeforeStart(env, tasks[plumbing],  tasks[garden]));
model.add(IloEndBeforeStart(env, tasks[windows],   tasks[moving]));
model.add(IloEndBeforeStart(env, tasks[facade],    tasks[moving]));
model.add(IloEndBeforeStart(env, tasks[garden],    tasks[moving]));
model.add(IloEndBeforeStart(env, tasks[painting],  tasks[moving]));
```

For each house and each given pair of tasks and worker that must have continuity, you constrain that if the interval variable for one of the two tasks for the worker is present, then the interval variable associated with that worker and the other task must also be present. To represent if a task is performed by a worker, you again use the constraint `IloPresenceOf`.

Step 7: Add the same worker constraints

Add the following code after the comment `//Add same worker constraints`

```
model.add(IloPresenceOf(env, taskMatrix[masonry][joe]) ==
          IloPresenceOf(env, taskMatrix[carpentry][joe]));
model.add(IloPresenceOf(env, taskMatrix[roofing][jack]) ==
          IloPresenceOf(env, taskMatrix[facade][jack]));
model.add(IloPresenceOf(env, taskMatrix[carpentry][joe]) ==
          IloPresenceOf(env, taskMatrix[roofing][joe]));
model.add(IloPresenceOf(env, taskMatrix[garden][jim]) ==
          IloPresenceOf(env, taskMatrix[moving][jim]));
```

This completes the `MakeHouse` function. In the main function, you now call the `MakeHouse` function, once for each house. At each call, the objective expression, `skill`, is updated and additional elements are appended to the arrays `allTasks` and the arrays in the matrix `workerTasks`. The deadline and house identifier are also passed to the `MakeHouse` function.

Step 8: Create the houses

Add the following code after the comment `//Create the houses`

```
for (h=0; h<nbHouses; ++h)
  MakeHouse(model, skill, allTasks, workerTasks, h, deadline);
```

To add the constraints that a given worker can be assigned only one task at a time, you use the classes `IloIntervalSequenceVar` and `IloNoOverlap` as in *Using no overlap constraints on interval variables: house building with workers*.

Step 9: Add the no overlap constraints

Add the following code after the comment `//Add the no overlap constraints`

```
for (w=0; w<NbWorkers; ++w) {
  IloIntervalSequenceVar seq(env, workerTasks[w], WorkerNames[w]);
  model.add(IloNoOverlap(env, seq));
}
```

The objective of this problem is to maximize the skill levels used for all the tasks, so you maximize the expression in the array `skill`.

Step 10: Add the objective

Add the following code after the comment `//Add the objective`

```
model.add(IloMaximize(env, skill));
```

Solve

You use an instance of the class `IloCP` to solve a problem expressed in a model. The constructor for `IloCP` takes an `IloModel` as its argument.

Step 11: Create an instance of IloCP

Add the following code after the comment `//Create an instance of IloCP`

```
   IloCP cp(model);
```

You now use the member function `IloCP::solve`, which solves the problem contained in the model by using constructive search and constraint propagation. The search for an optimal solution in this problem could potentiality take a long time, so you place a fail limit on the solve process. The search will stop when the fail limit is reached, even if optimality of the current best solution is not guaranteed.

Step 12: Search for a solution

Add the following code after the comment `//Search for a solution`

```
   cp.setParameter(IloCP::FailLimit, 10000);
   if (cp.solve()) {
```

The member function `IloCP::solve` returns a Boolean value of type `IloBool`. If a solution is found, the value `IloTrue` is returned.

After a solution has been found, you can use the member functions `IloCP::getObjValue` and `IloCP::domain` to examine the solution. The stream `IloAlgorithm::out` is the communication stream for general output. The code for displaying the solution has been provided for you:

```
      cp.out() << "Solution with objective " << cp.getObjValue() << ":" <<
std::endl;
      for (IloInt i=0; i<allTasks.getSize(); ++i) {
        if (cp.isPresent(allTasks[i]))
          cp.out() << cp.domain(allTasks[i]) << std::endl;
      }
```

Step 13: Compile and run the program

Compile and run the program. You should get the following results:

```
Solution with objective 357:
```

```
H0-masonry-Joe [1: 0 -- 35 --> 35]
H0-carpentry-Joe [1: 170 -- 15 --> 185]
H0-plumbing-Jack [1: 65 -- 40 --> 105]
H0-ceiling-Jack [1: 50 -- 15 --> 65]
H0-roofing-Joe [1: 215 -- 5 --> 220]
H0-painting-Jim [1: 65 -- 10 --> 75]
H0-windows-Joe [1: 250 -- 5 --> 255]
H0-facade-Joe [1: 265 -- 10 --> 275]
H0-garden-Jim [1: 220 -- 5 --> 225]
H0-moving-Jim [1: 275 -- 5 --> 280]
H1-masonry-Joe [1: 35 -- 35 --> 70]
H1-carpentry-Joe [1: 140 -- 15 --> 155]
H1-plumbing-Jack [1: 215 -- 40 --> 255]
H1-ceiling-Jack [1: 105 -- 15 --> 120]
H1-roofing-Joe [1: 225 -- 5 --> 230]
H1-painting-Jim [1: 120 -- 10 --> 130]
H1-windows-Joe [1: 240 -- 5 --> 245]
H1-facade-Joe [1: 275 -- 10 --> 285]
H1-garden-Jim [1: 255 -- 5 --> 260]
H1-moving-Jim [1: 285 -- 5 --> 290]
H2-masonry-Joe [1: 105 -- 35 --> 140]
H2-carpentry-Joe [1: 155 -- 15 --> 170]
H2-plumbing-Jack [1: 175 -- 40 --> 215]
H2-ceiling-Joe [1: 185 -- 15 --> 200]
H2-roofing-Joe [1: 230 -- 5 --> 235]
H2-painting-Jim [1: 200 -- 10 --> 210]
H2-windows-Joe [1: 245 -- 5 --> 250]
H2-facade-Joe [1: 255 -- 10 --> 265]
H2-garden-Jim [1: 235 -- 5 --> 240]
H2-moving-Jim [1: 265 -- 5 --> 270]
H3-masonry-Joe [1: 70 -- 35 --> 105]
H3-carpentry-Joe [1: 200 -- 15 --> 215]
H3-plumbing-Jack [1: 255 -- 40 --> 295]
H3-ceiling-Jack [1: 120 -- 15 --> 135]
H3-roofing-Joe [1: 220 -- 5 --> 225]
H3-painting-Jim [1: 135 -- 10 --> 145]
H3-windows-Joe [1: 235 -- 5 --> 240]
H3-facade-Joe [1: 295 -- 10 --> 305]
H3-garden-Jim [1: 295 -- 5 --> 300]
H3-moving-Jim [1: 305 -- 5 --> 310]
H4-masonry-Jack [1: 0 -- 35 --> 35]
H4-carpentry-Jim [1: 103 -- 15 --> 118]
H4-plumbing-Jack [1: 135 -- 40 --> 175]
H4-ceiling-Jack [1: 35 -- 15 --> 50]
H4-roofing-Jack [1: 295 -- 5 --> 300]
H4-painting-Jim [1: 93 -- 10 --> 103]
H4-windows-Joe [1: 305 -- 5 --> 310]
H4-facade-Jack [1: 300 -- 10 --> 310]
H4-garden-Jim [1: 300 -- 5 --> 305]
H4-moving-Jim [1: 310 -- 5 --> 315]
```

The complete program can be viewed online in the `YourCPHome/examples/src/cpp/sched_optional.cpp` file.

Review exercises

Includes the review exercise.

In this section

Exercises
Lists the review exercises.

Suggested answers
Provides the answers to the review exercises.

Exercises

For answers, see *Suggested answers*.

1. What is an alternative constraint?

Suggested answers

Exercise 1

What is an alternative constraint?

Suggested answer

An alternative constraint is a constraint that specifies that if a given interval is present in the solution, then exactly one interval variable of the given set is present in the solution.

In other words, consider an alternative constraint created with an interval variable `a` and an array of interval variables `bs`. If `a` is present in the solution, then exactly one of the interval variables in `bs` will be present, and interval variable `a` starts and ends together with this chosen one.

Complete program

The complete house building program can be viewed online in the file `YourCPHome/examples/src/cpp/sched_optional.cpp`.

Results

```
Solution with objective 357:
H0-masonry-Joe [1: 0 -- 35 --> 35]
H0-carpentry-Joe [1: 170 -- 15 --> 185]
H0-plumbing-Jack [1: 65 -- 40 --> 105]
H0-ceiling-Jack [1: 50 -- 15 --> 65]
H0-roofing-Joe [1: 215 -- 5 --> 220]
H0-painting-Jim [1: 65 -- 10 --> 75]
H0-windows-Joe [1: 250 -- 5 --> 255]
H0-facade-Joe [1: 265 -- 10 --> 275]
H0-garden-Jim [1: 220 -- 5 --> 225]
H0-moving-Jim [1: 275 -- 5 --> 280]
H1-masonry-Joe [1: 35 -- 35 --> 70]
H1-carpentry-Joe [1: 140 -- 15 --> 155]
H1-plumbing-Jack [1: 215 -- 40 --> 255]
H1-ceiling-Jack [1: 105 -- 15 --> 120]
H1-roofing-Joe [1: 225 -- 5 --> 230]
H1-painting-Jim [1: 120 -- 10 --> 130]
H1-windows-Joe [1: 240 -- 5 --> 245]
H1-facade-Joe [1: 275 -- 10 --> 285]
H1-garden-Jim [1: 255 -- 5 --> 260]
H1-moving-Jim [1: 285 -- 5 --> 290]
H2-masonry-Joe [1: 105 -- 35 --> 140]
H2-carpentry-Joe [1: 155 -- 15 --> 170]
H2-plumbing-Jack [1: 175 -- 40 --> 215]
H2-ceiling-Joe [1: 185 -- 15 --> 200]
H2-roofing-Joe [1: 230 -- 5 --> 235]
H2-painting-Jim [1: 200 -- 10 --> 210]
H2-windows-Joe [1: 245 -- 5 --> 250]
H2-facade-Joe [1: 255 -- 10 --> 265]
H2-garden-Jim [1: 235 -- 5 --> 240]
H2-moving-Jim [1: 265 -- 5 --> 270]
H3-masonry-Joe [1: 70 -- 35 --> 105]
H3-carpentry-Joe [1: 200 -- 15 --> 215]
H3-plumbing-Jack [1: 255 -- 40 --> 295]
H3-ceiling-Jack [1: 120 -- 15 --> 135]
H3-roofing-Joe [1: 220 -- 5 --> 225]
H3-painting-Jim [1: 135 -- 10 --> 145]
H3-windows-Joe [1: 235 -- 5 --> 240]
H3-facade-Joe [1: 295 -- 10 --> 305]
H3-garden-Jim [1: 295 -- 5 --> 300]
H3-moving-Jim [1: 305 -- 5 --> 310]
H4-masonry-Jack [1: 0 -- 35 --> 35]
H4-carpentry-Jim [1: 103 -- 15 --> 118]
H4-plumbing-Jack [1: 135 -- 40 --> 175]
```

```
H4-ceiling-Jack [1: 35 -- 15 --> 50]
H4-roofing-Jack [1: 295 -- 5 --> 300]
H4-painting-Jim [1: 93 -- 10 --> 103]
H4-windows-Joe [1: 305 -- 5 --> 310]
H4-facade-Jack [1: 300 -- 10 --> 310]
H4-garden-Jim [1: 300 -- 5 --> 305]
H4-moving-Jim [1: 310 -- 5 --> 315]
```

Using state functions: house building with state incompatibilities

This section describes how to model and solve a problem using state functions.

In this section

Overview
Describes how to model and solve a problem using state functions.

Describe
Describes the first stage in finding a solution to the house building problem with state incompatibilities.

Model
Describes the second stage in finding a solution to the house building problem with state incompatibilities.

Solve
Describes the third stage in finding a solution to the house building problem with state incompatibilities.

Review exercises
Includes the review exercises.

Complete program
Lists the location of the complete house building with state incompatibilities program.

Overview

In this lesson, you will learn how to:

- use the class `IloStateFunction`;
- use the specialized constraint `IloAlwaysEqual`.

You will learn how to model and solve a house building problem, a problem of scheduling the tasks involved in building multiple houses. Some tasks must necessarily take place before other tasks, and each task has a predefined size. Moreover, there are two workers, and each task requires either one of the two workers. A subset of the tasks require that the house be clean, whereas other tasks make the house dirty. A transition time is needed to change the state of the house from dirty to clean. To find a solution to this problem using IBM® ILOG® CP Optimizer, you will use the three-stage method: describe, model and solve.

Describe

The problem consists of assigning start dates to a set of tasks in such a way that the schedule satisfies temporal constraints and minimizes an expression. The objective for this problem is to minimize the overall completion date.

For each task type in the house building project, the following table shows the duration of the task in days along with state of the house during the task. A worker can only work on one task at a time; each task, once started, may not be interrupted.

House construction tasks

Task	Duration	State	Preceding tasks
masonry	35	dirty	
carpentry	15	dirty	masonry
plumbing	40	clean	masonry
ceiling	15	clean	masonry
roofing	5	dirty	carpentry
painting	10	clean	ceiling
windows	5	dirty	roofing
facade	10		roofing, plumbing
garden	5		roofing, plumbing
moving	5		windows, facade, garden, painting

Solving the problem consists of determining starting dates for the tasks such that the overall completion date is minimized.

Step 1: Describe the problem

The first step in modeling and solving a problem is to write a natural language description of the problem, identifying the decision variables and the constraints on these variables.

Write a natural language description of this problem. Answer these questions:

- What is the known information in this problem?
- What are the decision variables or unknowns in this problem?
- What are the constraints on these variables?
- What is the objective?

Discussion

What is the known information in this problem?

- There are five houses to be built by two workers. For each house, there are ten house building tasks, each with a given size. For each task, there is a list of tasks that must be

completed before the task can start. There are two workers. There is a transition time associated with changing the state of a house from dirty to clean.

What are the decision variables or unknowns in this problem?

- The unknowns are the dates that the tasks will start. The cost is determined by the assigned start dates.

What are the constraints on these variables?

- There are constraints that specify that a particular task may not begin until one or more given tasks have been completed. Each task requires either one of the two workers. Some tasks have a specified house cleanliness state.

What is the objective?

- The objective is to minimize the overall completion date.

Model

After you have written a description of your problem, you can use IBM® ILOG® Concert Technology classes to model it.

Step 2: Open the example file

Open the example file `YourCPHome/examples/tutorial/cpp/sched_state_partial.cpp` in your development environment. This file is a program that is only partially completed. You will fill in the blanks in each step in this lesson. At the end, you will have completed the program code, and you can compile and run the program.

In this lesson, you include the header file `<ilcp/cp.h>`. To catch exceptions that may be thrown, you use a `try/catch` block. The code for creating the environment and model and for printing out the solution found by the CP Optimizer engine is provided.

In addition, the data related to the tasks, such as the tasks (`Tasks`), the number of tasks (`NbTasks`), the names of the tasks (`TaskNames`) and sizes of the tasks (`TaskDurations`) and constants representing the number of houses (`NbHouses`), the number of workers (`NbWorkers`) and the two cleanliness states (`Clean` and `Dirty`) are provided.

After you create an environment and a model, you need to define the decision variables and add the constraints and objective to the model. Since the requirements for each of the five houses are similar, you use a function `MakeHouse`. to create the decision variables, constraints and objective associated with each house. Information about individual houses that must be shared with the main function includes the expressions needed to create the objective function and information about worker usage. In order to display the results of the optimization, it is also useful to maintain an array of all the interval variables.

To access this information, you create objects that will be updated in the `MakeHouse` function. The cost expression involves the date at which moving is completed for each house; the integer expression array `ends` is used to store this information. An array of task interval variables, `allTasks`, stores all the interval variables that are created. In addition, the expression used to represent worker usage is included in the global information that is updated in each call to the `MakeHouse` function.

Since the workers are equivalent in this problem, it is better to represent them as one pool of workers instead of as individual workers with no overlap constraints as was done in the earlier examples. This representation removes symmetry. The cumulative function expression representing usage of the pool of workers can be modified by the interval variables that require a worker.

Step 3: Declare the objects needed for MakeHouse

Add the following code after the comment `//Declare the objects needed for MakeHouse`

```
IloInt i;
IloModel model(env);
IloIntExprArray ends(env);
IloIntervalVarArray allTasks(env);
```

```
   IloCumulFunctionExpr workers(env);
```

The transition time from a dirty state to a clean state is the same value for all houses. Thus you can create one `IloTransitionDistance` object that can be shared by all houses. Here there are two transition types.

Step 4: Create the transition times

Add the following code after the comment `//Create the transition times`

```
   IloTransitionDistance ttime(env, 2);
   ttime.setValue(Dirty, Clean, 1);
```

You need to pass the model, the house identifier, the array of expressions representing the completion dates of the houses, the array of all tasks, the cumulative function expression for the worker usage and the transition object as arguments to the `MakeHouse` function.

Step 5: Create the MakeHouse function

Add the following code after the comment `//Create the MakeHouse function`

```
void MakeHouse(IloModel model,
         IloInt id,
         IloIntExprArray ends,
         IloIntervalVarArray allTasks,
         IloCumulFunctionExpr& workers,
         IloTransitionDistance ttime) {
```

Each house has a list of `NbTasks` that must be scheduled. Task `i`, where `i` is in `0..NbTasks-1`, has a size of `TaskDurations[i]` and the name `TaskNames[i]`. Using these, you build an array `tasks` of interval variables.

Each task also requires one worker from the start to the end of the task interval. To represent the fact that a worker is required for the task, you modify the cumulative function expression, `workerUsage`, using the function `IloPulse`.

You also add each interval variable to the array `allTasks` that will be used to display the solution once the schedule has been determined.

Step 6: Create the interval variables

Add the following code after the comment `//Create the interval variables`

```
  char name[128];
  IloIntervalVarArray tasks(env, NbTasks);
  for (IloInt i=0; i<NbTasks; ++i) {
    sprintf(name, "H%ld-%s", id, TaskNames[i]);
```

```
  tasks[i] = IloIntervalVar(env, TaskDurations[i], name);
  workers += IloPulse(tasks[i], 1);
  allTasks.add(tasks[i]);
}
```

The tasks in the model have precedence constraints that are added to the model.

Step 7: Add the temporal constraints

Add the following code after the comment `//Add the temporal constraints`

```
model.add(IloEndBeforeStart(env, tasks[masonry],   tasks[carpentry]));
model.add(IloEndBeforeStart(env, tasks[masonry],   tasks[plumbing]));
model.add(IloEndBeforeStart(env, tasks[masonry],   tasks[ceiling]));
model.add(IloEndBeforeStart(env, tasks[carpentry], tasks[roofing]));
model.add(IloEndBeforeStart(env, tasks[ceiling],   tasks[painting]));
model.add(IloEndBeforeStart(env, tasks[roofing],   tasks[windows]));
model.add(IloEndBeforeStart(env, tasks[roofing],   tasks[facade]));
model.add(IloEndBeforeStart(env, tasks[plumbing],  tasks[facade]));
model.add(IloEndBeforeStart(env, tasks[roofing],   tasks[garden]));
model.add(IloEndBeforeStart(env, tasks[plumbing],  tasks[garden]));
model.add(IloEndBeforeStart(env, tasks[windows],   tasks[moving]));
model.add(IloEndBeforeStart(env, tasks[facade],    tasks[moving]));
model.add(IloEndBeforeStart(env, tasks[garden],    tasks[moving]));
model.add(IloEndBeforeStart(env, tasks[painting],  tasks[moving]));
```

Certain tasks require the house to be clean, and other tasks cause the house to be dirty. To model the possible states of the house, Concert Technology provides the class `IloStateFunction` to represent the disjoint states through time.

Note: **State function**

A state function, represented in IBM ILOG Concert Technology by `IloStateFunction`, is a decision variable whose value is a set of non-overlapping intervals over which the function maintains a particular non-negative integer state. In between those intervals, the state of the function is not defined, typically because of an ongoing transition between two states.

You create one state function object in each execution of `MakeHouse`, one for each house.

The first argument passed to the constructor of the class `IloStateFunction` is the environment. The second argument is the transition time object. The final argument is an optional name used for debug and trace purposes. Here is a constructor:

```
IloStateFunction(const IloEnv env,
                 const IloTransitionDistance tdist,
```

```
                    const char* name =0);
```

To model the state required or imposed by a task, you create a constraint that specifies the state of the house throughout the interval variable representing that task.

Note: **Constraint on cumul function expression**

With the specialized constraint `IloAlwaysEqual`, you can create a constraint that specifies the value of a state function over the interval variable.

The constraint takes a state function, an interval variable and a state value. Whenever the interval variable is present, then the state function is defined everywhere between the start and the end of the interval variable and remains equal to the specified state value over this interval.

The first argument passed to the function `IloAlwaysEqual` is the environment. The second argument is the state function. The third argument is the interval variable on which you want to place the constraint. The fourth argument is the state value that the state function must take during the interval.

Here is a function signature:

```
IloConstraint IloAlwaysEqual(const IloEnv env,
                             const IloStateFunction f,
                             const IloIntervalVar a,
                             IloInt v);
```

You create a state function and constrain the state function to take the appropriate values during the tasks that require the house to be in a specific state.

Step 8: Add the state constraints

Add the following code after the comment `//Add the state constraints`

```
IloStateFunction houseState(env, ttime);
model.add(IloAlwaysEqual(env, houseState, tasks[masonry],   Dirty));
model.add(IloAlwaysEqual(env, houseState, tasks[carpentry], Dirty));
model.add(IloAlwaysEqual(env, houseState, tasks[plumbing],  Clean));
model.add(IloAlwaysEqual(env, houseState, tasks[ceiling],   Clean));
model.add(IloAlwaysEqual(env, houseState, tasks[roofing],   Dirty));
model.add(IloAlwaysEqual(env, houseState, tasks[painting],  Clean));
model.add(IloAlwaysEqual(env, houseState, tasks[windows],   Dirty));
```

To model the cost of building the houses, you will need to determine the maximum completion date among the individual house projects. To access the expression representing the completion date of the house currently in consideration, you use the function `IloEndOf` on

the last task in building a house (here, it is the moving task) and store this expression in the array `ends`.

Step 9: Add the cost expression

Add the following code after the comment `//Add the cost expression`

```
ends.add(IloEndOf(tasks[moving]));
```

This completes the `MakeHouse` function. In the main function, you now call the `MakeHouse` function, once for each house. At each call, the cumulative expression, `workers`, is updated and additional elements are appended to the arrays `ends` and `allTasks`. The model, house identifier and transition object are also passed to the `MakeHouse` function.

Step 10: Create the houses

Add the following code after the comment `//Create the houses`

```
for (i=0; i<NbHouses; ++i) {
  MakeHouse(model, i, ends, allTasks, workers, ttime);
}
```

To add the constraint that there can be only two workers working at a given time, you constrain the cumulative function expression representing worker usage to be no greater than the value `NbWorkers`.

Step 11: Add the cumulative constraints

Add the following code after the comment `//Add the cumulative constraints`

```
model.add(workers <= NbWorkers);
```

The objective of this problem is to minimize the overall completion date (the completion date of the house that is completed last). To do this, you minimize the maximal expression in the array `ends`.

Step 12: Add the objective

Add the following code after the comment `//Add the objective`

```
model.add(IloMinimize(env, IloMax(ends)));
```

Solve

You use an instance of the class `IloCP` to solve a problem expressed in a model. The constructor for `IloCP` takes an `IloModel` as its argument.

Step 13: Create an instance of IloCP

Add the following code after the comment `//Create an instance of IloCP`

```
   IloCP cp(model);
```

You now use the member function `IloCP::solve`, which solves the problem contained in the model by using constructive search and constraint propagation. The search for an optimal solution in this problem could potentiality take a long time, so you place a fail limit on the solve process. The search will stop when the fail limit is reached, even if optimality of the current best solution is not guaranteed.

Step 14: Search for a solution

Add the following code after the comment `//Search for a solution`

```
   cp.setParameter(IloCP::FailLimit, 10000);
   if (cp.solve()) {
```

The member function `IloCP::solve` returns a Boolean value of type `IloBool`. If a solution is found, the value `IloTrue` is returned.

After a solution has been found, you can use the member functions `IloCP::getObjValue` and `IloCP::domain` to examine the solution. The stream `IloAlgorithm::out` is the communication stream for general output.The code for displaying the solution has been provided for you:

```
      cp.out() << "Solution with objective " << cp.getObjValue() << ":" <<
std::endl;
      for (i=0; i<allTasks.getSize(); ++i) {
        cp.out() << cp.domain(allTasks[i]) << std::endl;
      }
```

Step 15: Compile and run the program

Compile and run the program. You should get the following results:

```
Solution with objective 357:
H0-masonry-Joe [1: 0 -- 35 --> 35]
```

```
H0-carpentry-Joe [1: 170 -- 15 --> 185]
H0-plumbing-Jack [1: 65 -- 40 --> 105]
H0-ceiling-Jack [1: 50 -- 15 --> 65]
H0-roofing-Joe [1: 215 -- 5 --> 220]
H0-painting-Jim [1: 65 -- 10 --> 75]
H0-windows-Joe [1: 250 -- 5 --> 255]
H0-facade-Joe [1: 265 -- 10 --> 275]
H0-garden-Jim [1: 220 -- 5 --> 225]
H0-moving-Jim [1: 275 -- 5 --> 280]
H1-masonry-Joe [1: 35 -- 35 --> 70]
H1-carpentry-Joe [1: 140 -- 15 --> 155]
H1-plumbing-Jack [1: 215 -- 40 --> 255]
H1-ceiling-Jack [1: 105 -- 15 --> 120]
H1-roofing-Joe [1: 225 -- 5 --> 230]
H1-painting-Jim [1: 120 -- 10 --> 130]
H1-windows-Joe [1: 240 -- 5 --> 245]
H1-facade-Joe [1: 275 -- 10 --> 285]
H1-garden-Jim [1: 255 -- 5 --> 260]
H1-moving-Jim [1: 285 -- 5 --> 290]
H2-masonry-Joe [1: 105 -- 35 --> 140]
H2-carpentry-Joe [1: 155 -- 15 --> 170]
H2-plumbing-Jack [1: 175 -- 40 --> 215]
H2-ceiling-Joe [1: 185 -- 15 --> 200]
H2-roofing-Joe [1: 230 -- 5 --> 235]
H2-painting-Jim [1: 200 -- 10 --> 210]
H2-windows-Joe [1: 245 -- 5 --> 250]
H2-facade-Joe [1: 255 -- 10 --> 265]
H2-garden-Jim [1: 235 -- 5 --> 240]
H2-moving-Jim [1: 265 -- 5 --> 270]
H3-masonry-Joe [1: 70 -- 35 --> 105]
H3-carpentry-Joe [1: 200 -- 15 --> 215]
H3-plumbing-Jack [1: 255 -- 40 --> 295]
H3-ceiling-Jack [1: 120 -- 15 --> 135]
H3-roofing-Joe [1: 220 -- 5 --> 225]
H3-painting-Jim [1: 135 -- 10 --> 145]
H3-windows-Joe [1: 235 -- 5 --> 240]
H3-facade-Joe [1: 295 -- 10 --> 305]
H3-garden-Jim [1: 295 -- 5 --> 300]
H3-moving-Jim [1: 305 -- 5 --> 310]
H4-masonry-Jack [1: 0 -- 35 --> 35]
H4-carpentry-Jim [1: 103 -- 15 --> 118]
H4-plumbing-Jack [1: 135 -- 40 --> 175]
H4-ceiling-Jack [1: 35 -- 15 --> 50]
H4-roofing-Jack [1: 295 -- 5 --> 300]
H4-painting-Jim [1: 93 -- 10 --> 103]
H4-windows-Joe [1: 305 -- 5 --> 310]
H4-facade-Jack [1: 300 -- 10 --> 310]
H4-garden-Jim [1: 300 -- 5 --> 305]
H4-moving-Jim [1: 310 -- 5 --> 315]
```

The complete program can be viewed online in the `YourCPHome/examples/src/cpp/sched_state.cpp` file.

Review exercises

Includes the review exercises.

In this section

Exercises
Lists the review exercises.

Suggested answers
Provides the answers to the review questions.

Exercises

For answers, see *Suggested answers*.

1. What is a state function?

Suggested answers

Exercise 1

What is a state function?

Suggested answer

A state function is a decision variable whose value is a set of non-overlapping intervals over which the function maintains a particular non-negative integer state. In between those intervals, the state of the function is not defined, typically because of an ongoing transition between two states.

Complete program

The complete house building program can be viewed online in the file `YourCPHome/examples/src/cpp/sched_state.cpp`.

Results

```
Solution with objective 357:
H0-masonry-Joe [1: 0 -- 35 --> 35]
H0-carpentry-Joe [1: 170 -- 15 --> 185]
H0-plumbing-Jack [1: 65 -- 40 --> 105]
H0-ceiling-Jack [1: 50 -- 15 --> 65]
H0-roofing-Joe [1: 215 -- 5 --> 220]
H0-painting-Jim [1: 65 -- 10 --> 75]
H0-windows-Joe [1: 250 -- 5 --> 255]
H0-facade-Joe [1: 265 -- 10 --> 275]
H0-garden-Jim [1: 220 -- 5 --> 225]
H0-moving-Jim [1: 275 -- 5 --> 280]
H1-masonry-Joe [1: 35 -- 35 --> 70]
H1-carpentry-Joe [1: 140 -- 15 --> 155]
H1-plumbing-Jack [1: 215 -- 40 --> 255]
H1-ceiling-Jack [1: 105 -- 15 --> 120]
H1-roofing-Joe [1: 225 -- 5 --> 230]
H1-painting-Jim [1: 120 -- 10 --> 130]
H1-windows-Joe [1: 240 -- 5 --> 245]
H1-facade-Joe [1: 275 -- 10 --> 285]
H1-garden-Jim [1: 255 -- 5 --> 260]
H1-moving-Jim [1: 285 -- 5 --> 290]
H2-masonry-Joe [1: 105 -- 35 --> 140]
H2-carpentry-Joe [1: 155 -- 15 --> 170]
H2-plumbing-Jack [1: 175 -- 40 --> 215]
H2-ceiling-Joe [1: 185 -- 15 --> 200]
H2-roofing-Joe [1: 230 -- 5 --> 235]
H2-painting-Jim [1: 200 -- 10 --> 210]
H2-windows-Joe [1: 245 -- 5 --> 250]
H2-facade-Joe [1: 255 -- 10 --> 265]
H2-garden-Jim [1: 235 -- 5 --> 240]
H2-moving-Jim [1: 265 -- 5 --> 270]
H3-masonry-Joe [1: 70 -- 35 --> 105]
H3-carpentry-Joe [1: 200 -- 15 --> 215]
H3-plumbing-Jack [1: 255 -- 40 --> 295]
H3-ceiling-Jack [1: 120 -- 15 --> 135]
H3-roofing-Joe [1: 220 -- 5 --> 225]
H3-painting-Jim [1: 135 -- 10 --> 145]
H3-windows-Joe [1: 235 -- 5 --> 240]
H3-facade-Joe [1: 295 -- 10 --> 305]
H3-garden-Jim [1: 295 -- 5 --> 300]
H3-moving-Jim [1: 305 -- 5 --> 310]
H4-masonry-Jack [1: 0 -- 35 --> 35]
H4-carpentry-Jim [1: 103 -- 15 --> 118]
H4-plumbing-Jack [1: 135 -- 40 --> 175]
```

```
H4-ceiling-Jack [1: 35 -- 15 --> 50]
H4-roofing-Jack [1: 295 -- 5 --> 300]
H4-painting-Jim [1: 93 -- 10 --> 103]
H4-windows-Joe [1: 305 -- 5 --> 310]
H4-facade-Jack [1: 300 -- 10 --> 310]
H4-garden-Jim [1: 300 -- 5 --> 305]
H4-moving-Jim [1: 310 -- 5 --> 315]
```

Using search parameters: team building

This section describes how to model and solve a problem using search parameters.

In this section

Overview
Describes how to model and solve a problem using search parameters.

Describe
Describes the first stage in finding a solution to the team configuration problem.

Model
Describes the second stage in finding a solution to the team configuration problem.

Solve
Describes the third stage in finding a solution to the team configuration problem.

Review exercises
Includes the review exercises and suggested answers.

Complete program
Lists the location of the complete team configuration problem and the results.

Overview

In this lesson, you will learn how to:

- use subproblems to build data for modeling constraints;
- use search parameters to modify constraint propagation.

You will learn how to model and solve an assignment problem regarding configuring teams at a corporation. To find an optimal solution to the problem using IBM® ILOG® CP Optimizer, you will use the three-stage method: describe, model and solve.

Describe

In this lesson, you will search for a solution to a team configuration problem. A corporation is planning an orientation day for new hires. During the day, the people attending the orientation, both new hires and existing employees, need to be separated into teams. There are 30 new hires and 30 existing employees who must be assigned to teams. These 60 employees from six service units need to be assigned to one of 10 teams, each of which has six slots. Each team must have three new hires and three existing employees assigned to it. At most four people from the same service unit can be on the same team. Twenty of the new hires have been paired with a coach, who is an existing employee from the same service unit as the newly hired employee. Both people of a pair must be on the same team.

The sixty people are labeled with unique identifiers in the range 0 to 59. The existing employees are assigned odd values, and the new hires are assigned even values. For the service unit memberships, Service A is the range of people with identifiers numbered 0 to 19, Service B is 20-39, Service C is 40-44, Service D is 45-49, Service E is 50-54, and Service F is 55-59. The pairs are the first six pairs from Services A and B and the first two pairs from Services C, D, E and F.

People from Services A and B cannot be assigned to the same team, and people from Services E and F cannot be assigned to the same team.

Using the identifiers, the additional preference constraints can be written as:

- person number 5 wants to be with person 41 or person 51,
- person number 15 wants to be with person 40 or person 51,
- person number 25 wants to be with person 40 or person 50, and
- person 20 is to be on the same team as person 24, or person 22 is to be on the same team as person 50.

Step 1: Describe the problem

The first step in modeling and solving a problem is to write a natural language description of the problem, identifying the decision variables and the constraints on these variables.

Write a natural language description of this problem. Answer these questions:

- What are the decision variables or unknowns in this problem?
- What are the constraints on these variables?

Discussion

What is the known information in this problem?

- There are thirty existing employees and thirty new hires. Each person belongs to one of six service units, and there are twenty coach/new-hire pairs.
- There are ten teams with six slots each.

What are the decision variables or unknowns in this problem?

- The unknown is to which team each person will be assigned. In other words, there is a decision variable for each of the 60 slots. The domain of each of these variables is the set of employees, or [0..59].

What are the constraints on these variables?

- Each team must have three existing employees and three new hires.
- Coach/new hire pairs must be assigned to the same team.
- No team can have more than four people from a given service unit.
- No team can have people from both Services A and B nor from both Services E and F.
- There are four constraints which describe additional preferences.

Model

After you have written a description of your problem, you can use IBM® ILOG® Concert Technology classes to model it.

Step 2: Open the example file

Open the example file `YourCPHome/examples/tutorial/cpp/teambuilding_partial.cpp` in your development environment. This file is a program that is only partially completed. You will fill in the blanks in each step in this lesson. At the end, you will have completed the program code and you can compile and run the program.

In this lesson, you include the header file `<ilcp/cp.h>`. To catch exceptions that may be thrown, you use a `try/catch` block. The code for declaring an environment and a model, for calculating the coaching pairs and for printing out the solution found by the IBM® ILOG® CP Optimizer engine is provided for you.

First, you represent the data of the program. The number of people, `nbPersons`, is set to 60 in this example, but can be easily modified. The number of teams, `nbTeams`, is 10 in this example. The size of a team, `teamSize`, is 6 in this example.

```
const IloInt nbPersons = 60;
const IloInt nbTeams = 10;
const IloInt teamSize = 6;
const IloInt nbServices = 6;
```

As the teams and the slots in each team are interchangeable, there is a lot of symmetry in this problem. To remove the symmetry, you will use a two-step process to solve this problem along with adding extra constraints. The first step is to determine a set of integer tuples, such that each tuple represents a possible set of new hires and existing employees to be assigned to a single team with six slots. The second step is to select a feasible set of ten of these tuples, one for each team, as the solution.

The first step requires solving a subproblem using IBM ILOG CP Optimizer. Since this subproblem is a constraint programming problem itself, you follow the three step process to solve the subproblem.

Step 3: Describe the subproblem

In this subproblem, you will search for all possible combinations for a single team. There are 60 employees from six service units who can to be assigned to a team of six people. The team must have three existing employees and three new hires assigned to it. At most four people from the same service unit can be on the team. If one person from a coach/new hire pair is on the team, the other must be as well.

The sixty people are labeled with unique identifiers in the range 0 to 59. The existing employees are those with odd numbered identifiers, and the new hires are those with even numbered identifiers. The service unit memberships are the same as outlined in the description.

Employees from Services A and B cannot be on a team together, and employees from Services E and F cannot be on a team together.

Discussion

What is the known information in this subproblem?

- There are thirty existing employees and thirty new hires. Each employee belongs to one of six service units, and there are twenty coach/new hire pairs.
- There is one team with slots for six people.

What are the decision variables or unknowns in this problem?

- The unknown is all of the possible combinations of employees that can be assigned to one team. In other words, there are six decision variables, one for each slot. The domain of each of these variables is the set of employees, or [0..59].

What are the constraints on these variables?

- The team must have three existing employees and three new hires.
- The team cannot have just one person from a coach/new hire pair.
- The team cannot have more than four people from a given service unit.
- The team cannot have people from both Services A and B nor from both Services E and F.

This subproblem is modeled and solved in the function `MakeTeamTuples`, which returns an instance of `IloIntTupleSet`. The tupleset is the set of all feasible solutions to the subproblem. In this function, the code for creating the environment and model for the subproblem and for filling the data arrays `newEmployee` and `service`, both indexed on the person identifiers, is provided for you. Each element of the array `service` states to which service unit the associated person belongs, where the value 0 indicates the person is an employee of Service A, the value 1 indicates the person is in Service B, etc. The array `newEmployee` is a Boolean array; an element of `newEmployee` has the value 1 if and only if that person is a new hire.

Each tuple in the set to be returned has `teamSize`, or 6, elements. You create the `IloIntTupleSet` using the global environment that is the input argument to the function `MakeTeamTuples`.

Step 4: Create the set of tuples

Add the following code after the comment `//Add the tupleset`

```
IloIntTupleSet ts(globalEnv, teamSize);
```

Each solution to the subproblem is an integer tuple of size `teamSize`. To represent the subproblem unknowns, you declare an array of decision variables, `teamMembers`, of length `teamSize`. The possible values for these variables are the person identifiers, or `[0..nbPersons-1]`.

Step 5: Create the team members variable array

Add the following code after the comment `//Add the team members variable array`

```
IloIntVarArray teamMembers(env, teamSize, 0, nbPersons-1);
```

Next, you begin to add the subproblem constraints. To model the constraint that there must be an equal number of new and existing employees on a team, you use an expression `nbNewEmployees` to count the number of new employees assigned to the array `teamMembers`. You use element expressions to determine if the person assigned to each element of `teamMembers` is a new employee and use the self-assigned addition operator on the expression to find the total number of new employees assigned to `teamMembers`. You constrain this expression to be equal to half the number of people on the team.

Step 6: Add the constraint on number of new employees

Add the following code after the comment `//Add the constraint on the number of new employees`

```
IloIntExpr nbNewEmployees(env);
for (i = 0; i < teamSize; i++)
  nbNewEmployees += newEmployee[teamMembers[i]];
model.add(nbNewEmployees == teamSize / 2);
```

To model the constraint that pairs must be together, you constrain that each member of the pair be assigned to `teamMembers` an equal number of times (in a later step you will add a constraint that ensures each person is assigned to `teamMembers` at most once in any given solution). You count how many times that person has been assigned to elements of `teamMembers` by using the IBM ILOG Concert Technology function `IloCount`. To make sure that either both or neither person in a pair is assigned to variables in `teamMembers`, you constrain that the number of times each person in the pair is assigned to the array `teamMembers` is equal. Here you use the array `coaching` which is initialized in the main program such that the `i`-th element has the value of -1 if the `i`-th person does not have a coach or the identifier of the coach if the `i`-th person has a coach.

Step 7: Add the constraint on coaching pairs

Add the following code after the comment `//Add the constraint on coaching pairs`

```
  for (i = 0; i < 60; i += 2) {
    if (coaching[i] >= 0)
      model.add(IloCount(teamMembers, i) == IloCount(teamMembers, coaching[i]
));
  }
```

To add the constraints on incompatible service units and the maximum number of people from a service unit that may be on the same team, you need to be able to represent the service unit of each employee. To do this, you create an array of auxiliary constrained integer variables `serviceVar`, which is indexed similarly to `teamMembers`. (For example, the third element of `serviceVar` represents to which service unit the person assigned to the third element of `teamMembers` belongs. These variables are linked using an element expression and the `service` data array.

Step 8: Add the service variables array

Add the following code after the comment `//Add the service unit variables`

```
IloIntVarArray serviceVar(env, teamSize, 0, nbServices - 1);
for (i = 0; i < teamSize; i++)
  model.add(serviceVar[i] == service[teamMembers[i]]);
```

You add the constraints that at most four people from a single service unit can be assigned to the same team using the IBM® ILOG® Concert Technology function `IloCount` and the array `serviceVar`.

Step 9: Add the constraint on cardinality of service unit on team

Add the following code after the comment `//Add the constraint on cardinality of service unit on team`

```
for (i = 0; i < nbServices; i++)
  model.add(IloCount(serviceVar, i) <= 4);
```

Employees of Services A and B may not be on the same team; likewise, employees of Services E and F may not be on the same team. To model this, you again use the IBM ILOG Concert Technology function `IloCount`, but instead of constraining the expression returned, you use it to form the subconstraints of logical constraints. These logical constraints state that the count of employees from either Service A or Service B who are assigned to the team must be zero, and likewise for Services E and F.

Step 10: Add the constraint on disjoint service units

Add the following code after the comment `//Add the constraint on disjoint service units`

```
model.add(IloCount(serviceVar, 0) == 0 || IloCount(serviceVar, 1) == 0);
model.add(IloCount(serviceVar, 4) == 0 || IloCount(serviceVar, 5) == 0);
```

As the elements, or slots, of the array `teamMembers` are not ordered, the search will likely encounter symmetry. In order to reduce symmetry, you introduce order among the variables in the array as discussed in *Using specialized constraints and tuples: scheduling teams*. By

introducing this order, you also ensure that no employee will be assigned to `teamMembers` more than once in any solution.

Step 11: Add the symmetry reducing constraint

Add the following code after the comment `//Add the symmetry reducing constraint`

```
for (i = 0; i < teamSize-1; i++)
  model.add(teamMembers[i] < teamMembers[i+1]);
```

You now search for all possible solutions to the submodel. Each solution is a valid assignment for one team. As each solution is found, the assigned values are stored in an integer tuple which is then added to the set of tuples that will be returned to the main program. You create an integer tuple, or array, to temporarily store each newly found valid team. You use an instance of `IloCP` to find the solutions to the problem in the model. Since you will need to find all solutions for this subproblem instead of one solution, you will not use `IloCP::solve`, which terminates after a single solution has been found, in the case that there is a solution.

When solving a problem that does not have an objective, the member functions `IloCP::startNewSearch` and `IloCP::next` can be used in a while loop to find all feasible solutions to a problem. The function `IloCP::startNewSearch` initializes the optimizer, the function `IloCP::next` searches for a new solution in the search space. The function `IloCP::end` cleans up the internal memory and data structures used by the optimizer.

Step 12: Begin the search for a solution to the subproblem

Add the following code after the comment `//Start the search`

```
IloIntArray tuple(globalEnv, teamSize);
IloCP cp(model);
cp.setParameter(IloCP::LogVerbosity, IloCP::Quiet);
cp.setParameter(IloCP::SearchType, IloCP::DepthFirst);
cp.startNewSearch();
while (cp.next()) {
```

As each solution is found, you copy the values assigned to the decision variables in the array `teamMembers` and store these values in a tuple. You add this new tuple to the tupleset and search for another solution. This loop is repeated until all solutions have been found.

Step 13: Search for all solutions to the subproblem

Add the following code after the comment `//Search for all solutions`

```
  for (IloInt i = 0; i < teamSize; i++)
    tuple[i] = cp.getValue(teamMembers[i]);
  ts.add(tuple);
```

```
}
```

The search terminates after all possible team tuples have been found. In order to clean up and reclaim the memory, you end the search and end the environment for the subproblem. You return the created tupleset of team configurations to the main program. Since the tupleset was created on the environment of the main program and not for the subproblem, it will not be destroyed in the clean up of the subproblem.

Step 14: Clean up the subproblem

Add the following code after the comment `//End the env for the subproblem`

```
cp.end();
env.end();
return ts;
```

Now that you have completed searching for all possible assignments for one team, you need to select ten of these tuples such that the employees are all assigned to teams and such that the additional preference constraints are satisfied.

You represent the people assigned to the teams with a matrix of constrained integer variables `group`, which is indexed on the team and slot. The array of constrained variables `group[i]` is the group of people assigned to the `i`-th team. Each element of `group` can be assigned a value in the interval [`0..nbPersons-1`].

For each `group[i]`, or set of people on a team, the only possible values are those in the tupleset retuned by the function `MakeTeamTuple`. To model this, you use the IBM® ILOG® Concert Technology function `IloAllowedAssignments`.

Step 15: Add the group variables and allowed assignments

Add the following code after the comment `//Add the group variables and allowed assignments`

```
IloArray<IloIntVarArray> groups(env, nbTeams);
for (i = 0; i < nbTeams; i++) {
  groups[i] = IloIntVarArray(env, teamSize, 0, nbPersons-1);
  model.add(IloAllowedAssignments(env,groups[i], tupleSet));
}
```

Since each person can be assigned to only one team, you must ensure that in any solution every variable in the array `group` takes a unique value. First you flatten the matrix `groups` into an array of decision variables. These are not new decision variables, but merely an alternate representation of the variables created previously. Then by using this new array as the argument to the constraint `IloAllDiff`, you can assure that no person will be assigned to two team slots and also that every person will be assigned to a slot.

Step 16: Add the all diff constraint

Add the following code after the comment `//Add the all diff constraint`

```
IloIntVarArray allVars(env, nbPersons);
IloInt s = 0, w, p;
for (w = 0; w < nbTeams; ++w) {
  for (p = 0; p < teamSize; ++p) {
    allVars[s] = groups[w][p];
    ++s;
  }
}
model.add(IloAllDiff(env, allVars));
```

To add the four preference constraints, you need to represent the team to which each employee is assigned. You declare an array of decision variables `team` of length `nbPersons`. The domain of each of the variables in `team` is `[0..nbTeams-1]` and the value assigned to `team[i]` represents the team to which person `i` is assigned.

Step 17: Add the team variables

Add the following code after the comment `//Add the team variables`

```
IloIntVarArray team(env, nbPersons, 0, nbTeams);
for (w = 0; w < nbTeams; ++w) {
  for (p = 0; p < teamSize; ++p) {
    model.add(team[groups[w][p]]==w);
  }
}
```

Using these variables, you add logical constraints to model the four preference constraints.

Step 18: Add the preference constraints

Add the following code after the comment `//Add the preference constraints`

```
model.add(team[5]== team[41] || team[5]==team[51]);
model.add(team[15]== team[40] || team[15]==team[51]);
model.add(team[25]== team[40] || team[25]==team[50]);
model.add(team[20]== team[24] || team[22]==team[50]);
```

Since the teams are all equivalent, each unique solution could be represented in 10! "different" solutions to this model. To reduce the symmetry in the search, you add constraints to introduce order to the groups. The person assigned to the initial slot in each element array of `group` should have a larger identifier than the person assigned to the initial slot of the previous element array of `group`.

Step 19: Add the symmetry constraint

Add the following code after the comment `//Add the symmetry constraint`

```
for (i=0; i<nbTeams-1; i++)
  model.add(groups[i][0] < groups[i+1][0]);
```

Solve

Solving a problem using constraint programming consists of assigning a value to each decision variable so that all constraints are satisfied. You may not always know beforehand whether there is a solution that satisfies all the constraints of the problem. In some cases, there may be no solution. In other cases, there may be many solutions to a problem.

You use an instance of the class `IloCP` to solve a problem expressed in a model. The constructor for `IloCP` takes an instance of `IloModel` as an argument.

Step 20: Create an instance of IloCP

Add the following code after the comment `//Create an instance of IloCP`

```
IloCP cp(model);
```

In this problem, using the built-in search leads to a long search time. To modify the search to make it more efficient, you use a search tuning parameter to change the *inference level* of the `IloAllDiff` constraint.

Each constraint is associated with a domain reduction algorithm. This algorithm performs domain reductions based on the associated constraint; the algorithm will remove values from the current domains of variables that do not belong to a solution. Constraint propagation is the mechanism used to communicate the effects of these domain reductions.

For some types of constraints, you can set an inference level that specifies the type of domain reduction algorithm. For specialized constraints such as `IloAllDiff`, the reduction algorithm varies depending on the inference level. This inference level can be changed by setting a parameter on `IloCP`.

To understand the difference between the inference levels, consider the `IloAllDiff` constraint. You can view this constraint in two ways. It can be seen as a set of inequality constraints or as one specialized constraint. For example, consider a graph coloring problem with three nodes: *x*, *y* and *z*. All the nodes must be colored a different color. Node *x* can be colored red or blue; node *y* can be colored red or blue; and node *z* can be colored red, blue, or yellow. If you set the inference level of `IloAllDiff` to the basic level using the tuning parameter, the domain reduction algorithm treats this specialized constraint as a set of inequality constraints. Looking at each set of binary constraints individually, the domain reduction algorithm is not able to reduce the domains.

If you set the inference level of `IloAllDiff` to the extended level, the domain reduction algorithm treats this constraint as a truly specialized constraint. The reduction algorithm is not able to reduce the domains of *x* and *y*. However, the reduction algorithm can “realize” that between them, variables *x* and *y* must use both of the values red and blue. This leaves only the value of yellow available for variable *z*.

Given that the extended level is the most thorough inference level, why would you use any other inference level? There is a trade-off in using the extended level. In general, the extended level takes longer. The basic inference level is less thorough, but faster. The medium level is a compromise between the two levels--faster than the extended level and more thorough than the basic. However, these are general rules and are not true for every situation. Depending on your application, different inference levels may be appropriate.

You use the member function `IloCP::setParameter` to set the inference levels for specialized constraints. This function takes two arguments: the first specifies the type of specialized constraint, the second specifies the inference level.

Step 21: Modify the search

Add the following code after the comment `//Modify the search`

```
cp.setParameter(IloCP::AllDiffInferenceLevel, IloCP::Extended);
```

You now use the member function `IloCP::solve`, which searches for a single solution to the problem using constructive search and constraint propagation.

Step 22: Search for a solution

Add the following code after the comment `//Search for a solution`

```
if (cp.solve()) {
  cp.out() << std::endl << "SOLUTION" << std::endl;
  for (p=0; p < nbTeams; ++p) {
    cp.out() << "team " << p << " : ";
    for (w=0; w < teamSize; ++w) {
      cp.out() << cp.getValue(groups[p][w]) << " ";
    }
    cp.out() << std::endl;
  }
}
else
  cp.out() << "**** NO SOLUTION ****" << std::endl;
```

Step 23: Compile and run the program

Compile and run the program. You should get the following results:

```
SOLUTION
team 0 : 0 1 2 3 55 56
team 1 : 4 5 15 18 50 51
team 2 : 6 7 16 19 45 46
team 3 : 8 9 12 14 49 59
team 4 : 10 11 13 17 44 54
team 5 : 20 21 24 25 40 41
team 6 : 22 23 32 33 57 58
team 7 : 26 27 38 39 52 53
team 8 : 28 29 34 35 42 43
team 9 : 30 31 36 37 47 48
```

The complete program can be viewed online in the `YourCPHome/examples/src/cpp/teambuilding.cpp` file.

Review exercises

Includes the review exercises and suggested answers.

In this section

Exercises
Lists the review exercises.

Suggested answers
Provides the suggested answers to the review exercises.

Exercises

For answers, see *Suggested answers*.

1. What is a search parameter?
2. Compare the search time with and without the search parameter that sets the inference level for the `IloAllDiff` constraint.
3. How can a subproblem help in modeling the original problem?

Suggested answers

Exercise 1

What is a search parameter?

Suggested answer

Setting search parameters such as `IloAllDiffInferenceLevel` allows the user to try different strategies to solve the problem. Increasing the inference level to `IloCP::Extended` may increase the time spent in constraint propagation after each choice, but it may also reduce the domains in such a way as to reduce the size of the search tree.

Search parameters also control the log output, the time limit and the search techniques that are used.

Exercise 2

Compare the search time with and without the search parameter that sets the inference level for the `IloAllDiff` constraint.

Suggested answer

The number of decisions required to find a solution is much greater and the running time is much longer when removing the inference level search parameter.

Exercise 3

How can a subproblem help in modeling the original problem?

Suggested answer

You can build data for the original problem by creating and solving a subproblem. Sometimes this is useful if the original problem is too large to model or if there is symmetry that can be removed by generating data.

Complete program

The complete team building program can be viewed online in the file `YourCPHome/examples/src/cpp/teambuilding.cpp`.

Results

```
SOLUTION
team 0 : 0 1 2 3 55 56
team 1 : 4 5 15 18 50 51
team 2 : 6 7 16 19 45 46
team 3 : 8 9 12 14 49 59
team 4 : 10 11 13 17 44 54
team 5 : 20 21 24 25 40 41
team 6 : 22 23 32 33 57 58
team 7 : 26 27 38 39 52 53
team 8 : 28 29 34 35 42 43
team 9 : 30 31 36 37 47 48
```

Using search phases on integer variables: steel mill

This section describes how to model and solve a problem using search phases.

In this section

Overview
Describes how to model and solve a problem using search phases.

Describe
Describes the first stage in finding a solution to the steel mill problem.

Model
Describes the second stage in finding a solution to the steel mill problem.

Solve
Describes the third stage in finding a solution to the steel mill problem.

Review exercises
Includes the review exercises and suggested answers.

Complete program
Lists the location of the complete steel mill program and the results.

Overview

In this lesson, you will learn how to:

- use the specialized constraint `IloPack`;
- use the specialized constraint `IloOr`;
- use search phases to direct the search.

You will learn to model and solve a problem regarding selecting steel slabs of various sizes and assigning a batch of steel coil orders to the selected slabs. This problem is based on problem 38 in the CSP Library (www.csplib.org). To find an optimal solution to this problem using IBM® ILOG® CP Optimizer, you will use the three-stage method: describe, model and solve.

Describe

In this lesson, you will solve an assignment problem. A steel mill needs to process a batch of coil orders using steel slabs of varying sizes. Each order has a size and a color associated with it. This color represents the specific process used to build the coil. A coil order must be built from only one slab. A slab can be used to process multiple coil orders from the batch; however, there can be at most two colors among the set of orders assigned to a given slab.

There are a finite number of slab sizes, but there are an unlimited number of slabs of each size available. The cumulative sum of the sizes of the coil orders assigned to a particular slab is called its load. The load assigned to a slab must not exceed the size, or capacity, of the slab. In addition, the production plan should minimize the unused capacity, the waste, of the selected slabs.

In the data set for this problem, there are 12 orders. In this batch of orders, there are eight different colors represented. As there are 12 orders and no order can be split among slabs, it is possible to deduce that at most 12 slabs will be needed.

Step 1: Describe the problem

The first step in modeling and solving a problem is to write a natural language description of the problem, identifying the decision variables and the constraints on these variables.

Write a natural language description of this problem. Answer these questions:

- What is the known information in this problem?
- What are the decision variables or unknowns in this problem?
- What are the constraints on these variables?
- What is the objective?

Discussion

What is the known information in this problem?

- There are 12 orders of varying colors and sizes.
- There are steel slabs of various sizes available. At most 12 slabs will be used.

What are the decision variables or unknowns in this problem?

- The unknowns are what size slabs will be selected and to which slab each order will be assigned. In other words, there is a decision variable associated with each order that indicates to which slab the order will be assigned. The domain of each of these variables is the set of slabs, or [0..11]. This formulation makes the assumption that the selected slabs have a placement, in other words, there is a first slab, a second slab and so on.

What are the constraints on these variables?

- Each slab can be assigned orders of at most two different colors. The load of a slab is equal to the cumulative sum of the size of the orders assigned to it.

What is the objective?

- The objective is to minimize loss, where loss is the excess capacity of a slab, the unused capacity of the selected slabs.

Model

After you have written a description of your problem, you can use IBM® ILOG® Concert Technology classes to model it. After you create a model of your problem, you can use IBM ILOG CP Optimizer classes and member functions to search for a solution.

Step 2: Open the example file

Open the example file `YourCPHome/examples/tutorial/cpp/steelmill_partial.cpp` in your development environment. This file is a program that is only partially completed. You will fill in the blanks in each step in this lesson. At the end, you will have completed the program code and you can compile and run the program.

The first step in converting your natural language description of the problem into code using IBM ILOG Concert Technology classes is to create an environment and a model.

Step 3: Create the environment and model

The following code which creates the environment and model has been provided for you

```
int main(int argc, const char * argv[]) {
  IloEnv env;
  try {
    IloModel model(env);
    IloInt m, o, c, q;
```

You create standard variables to represent the number of orders, `nbOrders`, the maximum number of slabs needed, `nbSlabs`, and the number of colors, `nbColors`. You model the order sizes and colors as arrays of integer values. The possible sizes, or capacities, of the slabs are also modeled as an array.

Step 4: Create the data

Add the following code after the comment `//Create the data`

```
    IloInt         nbOrders   = 12;
    IloInt         nbSlabs    = 12;
    IloInt         nbColors   = 8;
    IloIntArray    capacities(env, 20, 0, 11, 13, 16, 17, 19, 20,
                                       23, 24, 25, 26, 27, 28, 29,
                                       30, 33, 34, 40, 43, 45);
    IloIntArray    sizes(env, nbOrders, 22, 9, 9, 8, 8, 6, 5, 3, 3, 3, 2, 2);

    IloIntArray    colors(env, nbOrders,  5, 3, 4, 5, 7, 3, 6, 0, 2, 3, 1, 5)
;
```

The first array of decision variables represents which slab should be used to process each order. To represent this information, you declare an array of decision variables `where`. This array has `nbOrders` elements or, in this example, 12. The possible values for these variables represent the ordered slabs. In this example, a value of 0 represents the first slab, a value of 1 represents the second slab, and so on. The array of decision variables `where` will represent the solution to the problem, after the CP Optimizer engine has found a solution.

The second array of decision variables represents the load assigned to each slab. To represent this information, you declare an array of decision variables `load`. This array has `nbSlabs` elements, or, in this example, 12. The possible values for these variables represent the cumulative sum of the sizes of the orders assigned to each slab. The minimum load is 0 and the maximum load is the cumulative sum of the sizes of the entire batch of orders.

Step 5: Declare the decision variables

Add the following code after the comment `//Declare the decision variables`

```
  IloIntVarArray where(env, nbOrders, 0, nbSlabs-1);
  IloIntVarArray load(env, nbSlabs, 0, IloSum(sizes));
```

You now begin to add the constraints. First you create the constraint that links the `where` and `load` variables. For each slab, its load must be equal to the sum of sizes of orders that are assigned to it. To model this, you use the predefined constraint `IloPack`.

Note: **Load balancing constraint**

Using specialized constraints, such as `IloPack`, makes modeling simpler and solving more efficient.

The constraint `IloPack` maintains the load of a set of containers, given a set of weighted items and an assignment of items to containers.

The class `IloPack` is a subclass of the class `IloConstraint`. The first argument of the constructor of `IloPack` is the environment. The second argument is the array of load variables. The third argument is the array for assignments, in this case the slab to which each order is assigned. The fourth argument is the array of order sizes or weights. The fifth argument is an optional name used for debug and trace purposes. Here is the constructor you will use:

```
IloPack(const IloEnv env,
        const IloIntVarArray load,
        const IloIntVarArray where,
        const IloIntArray weight,
        const char * name=0);
```

The arrays `where` and `sizes` are indexed by the orders. For each slab `i`, the packing constraint requires that the value of `load[i]` is equal to the sum of the `sizes[j]` such that the value of `where[j]` is `i`.

Step 6: Add the pack constraint

Add the following code after the comment `//Add the pack constraint`

```
model.add(IloPack(env, load, where, sizes));
```

Next, you create the constraints to constrain the number of colors represented by the orders assigned to a slab. For a given slab, you create an array of expressions `colorExpArray` which is indexed on the set of colors. Each element has a domain of `[0..1]` and indicates whether or not at least one order of the corresponding color is assigned to the slab. For each color, you need to determine if an order of that color has been assigned to the given slab. Using the array `colorExpArray` and the predefined IBM ILOG Concert Technology constraint `IloOr`, you can link the order assignments to the `colorExpArray` to represent whether an order of a particular color has been assigned to the slab.

Note: **Disjunctive constraint**

Using specialized constraints such as `IloOr` makes modeling easier.

An instance of `IloOr` represents a disjunctive constraint. In other words, it defines a disjunctive-OR among any number of constraints. To add a constraint to this logical constraint, you use the method `IloOr::add`.

For example, you may write:

```
IloOr or(env); or.add(constraint1); or.add(constraint2); or.add
(constraint3);
```

Those lines are equivalent to:

```
IloConstraint or = constraint1 || constraint2 || constraint3;
```

For a given slab `m` and a given color, you iterate through the orders. If the order, `o`, is of the given color, then you add the constraint `(where[o] == m)` to the `IloOr` logical constraint. If any one of the orders of the given color is assigned to the given slab, then the value of the expression will be 1.

Finally, you constrain the sum of the elements of the `colorExpArray` to be no greater than two.

Step 7: Add the color constraints

Add the following code after the comment `//Add the color constraints`

```
for(m = 0; m < nbSlabs; m++) {
  IloExprArray colorExpArray(env);
  for(c = 0; c < nbColors; c++) {
    IloOr orExp(env);
```

```
      for(o = 0; o < nbOrders; o++){
        if (colors[o] == c){
          orExp.add(where[o] == m);
        }
      }
      colorExpArray.add(orExp);
    }
    model.add(IloSum(colorExpArray) <= 2);
  }
```

To create the objective expression, you calculate an expression to represent the loss for each slab. First you determine, for each possible value of load, the minimum sized slab needed and thus the loss that would be incurred. If a slab has a given load, the loss is the difference between the load assigned to the slab and the minimal size slab needed to process the load.

Step 8: Add the objective

Add the following code after the comment `//Add the objective`

```
IloIntArray lossValues(env);
lossValues.add(0);
for(q = 1; q < capacities.getSize(); q++){
  for(IloInt p = capacities[q-1] + 1; p <= capacities[q]; p++){
    lossValues.add(capacities[q] - p);
  }
}
IloExpr obj(env);
for(m = 0; m < nbSlabs; m++){
  obj += lossValues[load[m]];
}
model.add(IloMinimize(env, obj));
```

Since the slabs are ordered with no distinguishing characteristics, reordering the slabs would produce a number of equivalent solutions. One way to reduce symmetry is to introduce order among variables, as was done in previous lessons. You add constraints that state that the ordered slabs must have decreasing loads. This constraint eliminates some, though not all, symmetric solutions.

Step 9: Add the constraints to reduce symmetry

Add the following code after the comment `//Add the constraints to reduce symmetry`

```
for(m = 1; m < nbSlabs; m++){
  model.add(load[m-1] >= load[m]);
}
```

Solve

Solving a problem using constraint programming consists of assigning a value to each decision variable so that all constraints are satisfied and minimize the objective representing the cost of the solution. You may not always know beforehand whether there is a solution that satisfies all the constraints of the problem. In some cases, there may be no solution. In other cases, there may be many solutions to a problem.

You use an instance of the class `IloCP` to solve a problem expressed in a model. The constructor for `IloCP` takes an instance of `IloModel` as an argument.

Step 10: Create an instance of IloCP

Add the following code after the comment `//Create an instance of IloCP`

```
IloCP cp(model);
```

Since you know information about the structure of this problem, you can tune the optimizer to help it perform better. Once all the decision variables in the array `where` have been assigned values, the values for all of the other variables in the model will have been determined through constraint propagation. A search strategy that assigns values to the variables in this array first works well in this problem.

To tune the search to concentrate on the `where` variables, you create an instance of the class `IloSearchPhase` which takes the environment and the array of `where` variables as arguments. This tuning object is then passed as an argument to the `IloCP::solve` method.

Step 11: Search for a solution

Add the following code after the comment `//Search for a solution`

```
if (cp.solve(IloSearchPhase(env, where))){
  cp.out() << "Optimal value: " << cp.getValue(obj) << std::endl;
  for (m = 0; m < nbSlabs; m++) {
    IloInt p = 0;
    for (o = 0; o < nbOrders; o++)
      p += cp.getValue(where[o]) == m;
    if (p == 0) continue;
    cp.out() << "Slab " << m << " is used for order";
    if (p > 1) cp.out() << "s";
    cp.out() << " :";
    for (o = 0; o < nbOrders; o++) {
      if (cp.getValue(where[o]) == m)
        cp.out() << " " << o;
    }
    cp.out() << std::endl;
  }
```

```
    }
```

Step 12: Compile and run the program

Compile and run the program. You should get the following results:

```
Optimal value: 0
Slab 0 is used for orders : 0 8
Slab 1 is used for orders : 2 11
Slab 2 is used for orders : 5 6
Slab 3 is used for orders : 1 10
Slab 4 is used for orders : 3 9
Slab 5 is used for orders : 4 7
```

As you can see, sixslabs are used and the orders have been distributed such that there is no loss.

The complete program can be viewed online in the `YourCPHome/examples/src/cpp/steelmill.cpp` file.

Review exercises

Includes the review exercises and suggested answers.

In this section

Exercises
Lists the review exercises.

Suggested answers
Provides the suggested answers to the review exercises.

Exercises

For answers, see *Suggested answers*.

1. What is the packing constraint?
2. Modify the code to add a fixed of cost of six for each slab used.

Suggested answers

Exercise 1

What is the packing constraint?

Suggested answer

The constraint `IloPack` maintains the load of a set of containers, given a set of weighted items and an assignment of items to containers.

Exercise 2

Modify the code to add a fixed of cost for each slab used.

Suggested answer

The code that has changed from `steelmill.cpp` follows. You can view the complete program online in the file `YourCPHome/examples/src/cpp/steelmill_ex3.cpp`.

Just before the objective is added to the model, the additional variables and constraints are added to the code:

```
IloInt fixedCost=1;
IloIntVarArray used(env,nbSlabs,0,1);
for (m=0; m < nbSlabs; m++)
   model.add((load[m] > 0) == used[m]);
obj += fixedCost*IloSum(used);
```

The solution to the problem using the data given in the lesson code and using a fixed cost of 5 has assigned orders to five slabs.

Complete program

The complete steel mill program can be viewed online in the file `YourCPHome/examples/src/cpp/steelmill.cpp`.

Results

```
Optimal value: 0
Slab 0 is used for orders : 0 8
Slab 1 is used for orders : 2 11
Slab 2 is used for orders : 5 6
Slab 3 is used for orders : 1 10
Slab 4 is used for orders : 3 9
Slab 5 is used for orders : 4 7
```

Index