IBM ILOG Dispatcher V4.7

User's Manual

June 2009

C O N T E N T S

Preface **Welcome to IBM ILOG Dispatcher . . . 15**

What is Dispatcher? . . . 15

How to Use Dispatcher with Solver . . . 16

About this manual . . . 16

How this manual is organized . . . 17

Prerequisites . . . 17

Related documentation . . . 17

Installing IBM ILOG Dispatcher . . . 17

Linking IBM ILOG Dispatcher . . . 18

Typographic and naming conventions . . . 18

Include Files . . . 19

Accessing software support . . . 20

Contact via web . . . 20

Contact via phone . . . 21

Part I **The Basics . . . 24**

Chapter 1 **IBM ILOG Dispatcher Concepts . . . 25**

Describe . . . 26

Model . . . 27

Basic Modeling Objects . . . 27

Decision Variables . . . 28

Dimensions . . . 29

Using dimensions to model side constraints . . . 30

Modeling other side constraints . . . 35

Modeling costs and the objective . . . 37

Summary: The Dispatcher Model . . . 38

Solve **39**

Displaying the solution 42

Chapter 2 **Modeling a Vehicle Routing Problem 43**

Describe **43**

Model **44**

Declare the RoutingModel class 45

Define the addDimensions function 46

Define the createIloNodes function 47

Define the createVehicles function. 49

Define the createVisits function 51

Define the RoutingModel constructor. 53

Review Exercises **54**

Suggested Answers **54**

Exercise 1 54

Exercise 2 54

Chapter 3 **Solving a Vehicle Routing Problem 55**

Solve **55**

Declare the RoutingSolver class 56

Define the RoutingSolver constructor 57

Define the findFirstSolution function 59

Define the improveWithNHood function. 59

Define the printInformation function. 62

Define the main function 62

Review Exercises **66**

Suggested Answers **67**

Exercise 1 67

Exercise 2 67

Exercise 3 67

Complete Program **73**

Complete Output 78

Chapter 4 **Minimizing the Number of Vehicles 81**

Describe 82

Model. 82

Declare the RoutingModel class 83

Define the RoutingModel constructor. 84

Define the loadGraphInformation function 85

Define the lastMinuteGraphChanges function 86

Define the addDimensions function 87

Define the createIloNodes function 87

Define the createVehicles function. 88

Define the createVisits function 89

Solve 90

Declare the RoutingSolver class 91

Define the RoutingSolver constructor 92

Define the findFirstSolution function 92

Define the improveWithNHood function. 92

Define the closeEmptyVehicles function 93

Define the getShortestRoute function 94

Define the reduceActiveVehicles function 94

Define the printInformation function. 96

Define the main function 97

Review Exercises. 99

Suggested Answers. 100

Exercise 1 100

Exercise 2 100

Complete Program. 100

Complete Output 106

Chapter 5 **Adding Visit Disjunctions 119**

Describe 120

Model. 120

Declare the RoutingModel class 121

Define the RoutingModel constructor 122
Define the loadGraphInformation function 123
Define the lastMinuteGraphChanges function 123
Define the addDimensions function 124
Define the createIloNodes function 124
Define the createVehicles function. 125
Define the createVisits function 125
Define the createAdditionalVisits function 128
Define the function removeVisit 128
Solve 129
Declare the RoutingSolver class 129
Define the RoutingSolver constructor 129
Define the findFirstSolution function 130
Define the improveWithNHood function 130
Define the addNewVisit function 131
Define the removeVisitAndResolve function 132
Define the printInformation function 133
Define the main function 133
Review Exercises 137
Suggested Answers 137
Exercise 1 137
Exercise 2 137
Complete Program 137
Complete Output 143

Chapter 6 Multiple Tours per Vehicle 149
Describe 150
Model 150
Declare the RoutingModel class 151
Define the RoutingModel constructor 152
Define the loadGraphInformation and lastMinuteGraphChanges functions 152
Define the addDimensions function 152

Define the createIloNodes function 152
Define the createVehicles function. 153
Define the createVisits function 155
Solve 156
Declare the RoutingSolver class 157
Define the RoutingSolver constructor 157
Define the insertAllReturnVisits function 158
Define the orderVisits function 159
Define the insertCustomerVisits function 163
Define the improveWithNHood function 164
Define the printInformation function 165
Define the main function 165
Review Exercises 167
Suggested Answers. 167
Exercise 1 167
Exercise 2 168
Complete Program. 168
Complete Output 174

Part II Transportation Industry Solutions 178

Chapter 7 Pickup and Delivery Problems 179
Describe 179
Model. 181
Solve 184
Review Exercises 185
Suggested Answers. 185
Exercise 1 185
Exercise 2 185
Exercise 3 185
Complete Program. 186
Complete Output 190

Chapter 8 **Adding Vehicle Breaks** 193

Describe 193

Model 194

Solve 197

Review Exercises 200

Suggested Answers 201

Exercise 1 201

Exercise 2 201

Exercise 3 201

Complete Program 202

Complete Output 209

Chapter 9 **Adding Early and Late Costs** 223

Describe 223

Early and Late Cost Interdependence 224

Model 226

Solve 227

Review Exercises 228

Suggested Answers 229

Exercise 1 229

Exercise 2 229

Exercise 3 229

Exercise 4 230

Complete Program 230

Complete Output 235

Chapter 10 **Pickup and Delivery by Multiple Vehicles from Multiple Depots** 239

Describe 240

Model 242

Declare the Depot class 243

Define the Depot constructor 244

Define the Depot destructor 245

Define the Depot::improve function 245
Define the Depot::fillModel function 245
Define the Depot::createOneVehicle function 246
Define the Depot::createVehicles function 247
Declare the RoutingModel class 248
Define the RoutingModel constructor 249
Define the RoutingModel::parse function 249
Define the RoutingModel::createModel function 249
Define the RoutingModel::createDimensions function 250
Define the RoutingModel::createNodes function 251
Define the RoutingModel::createDepots function. 251
Define the RoutingModel::createVisits function 253
Solve 255
Declare the RoutingSolver class 256
Define the RoutingSolver constructor 256
Define the RoutingSolver::findFirstSolution function 257
Define the syncSolution functions 258
Define the RoutingSolver::improveDepots function 258
Define the RoutingSolver::improvePlan function 259
Define the RoutingSolver::printInformation function. 260
Define the main function 260
Review Exercises 263
Suggested Answers 263
Exercise 1 263
Exercise 2 263
Exercise 3 263
Complete Program 264
Complete Output 272

Chapter 11 Modeling Complex Costs 279

Describe cost1 280
Model cost1 280

Solve cost1 . . . . **281**
Describe cost2 . . . . **282**
Model cost2 . . . . **283**
Solve cost2 . . . . **283**
Review Exercises **284**
Suggested Answers **285**
Exercise 1 285
Exercise 2 285
Exercise 3 285
Complete cost1 Program **286**
Complete cost1 Output 291
Complete cost2 Program **293**
Complete cost2 Output 297

Chapter 12 Docking Bays: Modeling External Resources 305
Describe **305**
Model **306**
Solve **309**
Review Exercises **311**
Suggested Answers **311**
Exercise 1 311
Exercise 2 312
Exercise 3 312
Exercise 4 312
Complete Program **312**
Complete Output 318

Part III Field Service Solutions 322

Chapter 13 Dispatching Technicians 323
Describe **323**
Model **324**

Solve 328
Review Exercises 329
Suggested Answers 330
Exercise 1 330
Exercise 2 330
Exercise 3 330
Complete Program 330
Complete Output 336

Chapter 14 **Dispatching Technicians II** **339**
Describe 339
Model 340
Solve 349
Review Exercises 350
Suggested Answers 350
Exercise 1 350
Exercise 2 351
Exercise 3 351
Complete Program 351
Complete Output 359

Chapter 15 **CARP: Visiting Arcs Using Multiple Vehicles** **361**
Describe 361
Model 363
Solve 367
Review Exercises 374
Suggested Answers 374
Exercise 1 374
Exercise 2 375
Exercise 3 375
Complete Program 375
Complete Output 382

Part IV Developing Dispatcher Applications 394

Chapter 16 **Designing Dispatcher Models 395**

Simplify the model 395

Suggested procedure. 396

Decompose the problem 396

Temporal decomposition 397

Other modeling hints 406

Chapter 17 **Developing Your Own First Solution Heuristics 407**

Using the predefined first solution heuristics. 407

Deciding which heuristic to use 408

More hints 408

Creating your own first solution heuristic. 409

Heuristic Description 409

Building Routes 409

Building a Complete Solution. 411

Using the custom first solution goal 413

Using the Dispatcher First Solution Framework. 413

Creating the decision class 414

Creating the decision maker class 416

Creating the decision tracer class 418

Creating the first solution framework goal 419

Using the first solution framework goal 420

Chapter 18 **Developing Your Own Neighborhoods. 421**

Neighborhood Description 422

Defining the Neighborhood 422

Creating the Solution Delta 424

Implementing the Neighborhood 426

Appendix A **Predefined First Solution Heuristics. 429**

Example data. 429

Enumeration Heuristic 430
Savings Heuristic 431
Sweep Heuristic 433
Nearest-to-Depot Heuristic. 434
Nearest Addition Heuristic 435
Insertion Heuristic 437

Appendix B Predefined Neighborhoods 439

Intra-Route Improvement: IloTwoOpt neighborhood 440
Intra-Route Improvement: IloOrOpt neighborhood. 440
Inter-Route Improvement: IloRelocate neighborhood 441
Inter-Route Improvement: IloCross neighborhood 442
Inter-Route Improvement: IloExchange neighborhood 443
Other neighborhoods 444

Appendix C Predefined Search Heuristics and Metaheuristics 445

Basic Search Heuristics 445
First Accept Search 446
Best Accept Search 446
MetaHeuristics 447
Tabu Search Metaheuristic 447
Guided Local Search Metaheuristic 448
Using Metaheuristics In Search 448
Tabu Search. 448
Guided Local Search 450
Guided Tabu Search 452

Index 453

Welcome to IBM ILOG Dispatcher

This chapter introduces IBM® ILOG® Dispatcher. In it, you will learn:

- what Dispatcher is
- how to use it with Solver in a methodical way
- what is in this manual and how to use it with the reference manual
- where to find more information

If you are already familiar with Solver, then you know first-hand the advantages of that generic tool for object-oriented constraint programming. It provides a library of re-usable and maintainable C++ classes that can be exploited just as they are or extended to meet special needs.

What is Dispatcher?

So, what exactly is Dispatcher? It's a C++ library based on Solver, so it exploits all the facilities of object-orientation and constraint programming, too. One product is meant to work in combination with the other, and this manual aims to show you, through programming examples, exactly how to exploit this powerful combination.

Dispatcher offers features especially adapted to solving problems in vehicle routing and maintenance-technician dispatching. There are, for example, classes of objects particularly designed to represent such aspects as routing plans themselves, their visits, their vehicles, and their constraints, such as capacity or time-window constraints. Dispatcher offers you a workbench of tools that tackle the issues inherent in vehicle routing and maintenance-technician dispatching in a straightforward manner.

One of the main advantages of constraint programming is that it enables you to represent your problem explicitly, 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. Dispatcher, like Solver, embodies this advantage of modeling and solving in a ready-to-use library of tools.

How to Use Dispatcher with Solver

We mentioned that Dispatcher is based on Solver. What does this mean in practice? On the whole, in Dispatcher, you’ll find predefined classes of objects that will adequately and accurately represent aspects of your vehicle routing or technician dispatching problem. For special purposes, however, you can easily extend Dispatcher by defining additional classes yourself to represent objects particular to your problem.

In addition, if your routing problem includes unusual constraints not adequately represented by the standard Solver routing constraints described in this manual, you can define additional constraints that are specific to your problem.

Then with your problem well represented in the object model, you will use the control primitives of Dispatcher to solve the problem. In other words, you will implement a heuristic search procedure using the predefined features of Dispatcher. Throughout this manual, you will find examples of how to use the control primitives of Dispatcher to search for a solution.

About this manual

This is the *IBM ILOG Dispatcher User’s Manual*. It is composed of lessons that use a procedural-based learning strategy. Each lesson is built around a sample problem, and a user works on a partially completed code example. As you follow the steps in the lesson, you complete the code and learn about concepts. Then, you compile and run the code and analyze the results. At the end of each lesson, there are review exercises.

The manual is designed to be used by C++ 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 Dispatcher installed. You work through the lessons and exercises.

If you are a novice Dispatcher user, start at the beginning of this manual, since the lessons build on each other. If you are a more experienced Dispatcher user, you can jump ahead to a later part of the manual, focusing on more advanced topics.

How this manual is organized

This manual is divided into four parts:

- Part I, *The Basics*
- Part I, *Transportation Industry Solutions*
- Part I, *Field Service Solutions*
- Part I, *Developing Dispatcher Applications*

Prerequisites

Dispatcher requires a working knowledge of C++. However, it does not require you to learn a new language since it does not impose any syntactic extensions on C++.

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 IBM ILOG is required to use this manual.

You should have IBM ILOG Dispatcher, IBM ILOG Solver and IBM ILOG Concert Technology installed in your development environment before starting to use this manual. You should be able to compile, link, and execute a sample program provided with Dispatcher before starting to use this manual. For more information, see the sections *Installing IBM ILOG Dispatcher* and *Linking IBM ILOG Dispatcher.*

Related documentation

The following documentation ships with IBM ILOG Dispatcher and will be useful for you to refer to as you complete the lessons and exercises:

- The *IBM ILOG Dispatcher Reference Manual* fully documents all the Dispatcher C++ classes and member functions used in the User's Manual. The Reference Manual also explains certain concepts more formally. The Reference Manual provides the last word on any given topic.

Installing IBM ILOG Dispatcher

In this manual, it is assumed that you have already successfully installed the Dispatcher, Solver, and Concert Technology 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 the jacket of the CD-ROM of the standard distribution of Dispatcher. The instructions cover all the details you need to know to install Dispatcher, Concert Technology and Solver on your system.

Linking IBM ILOG Dispatcher

When you use Dispatcher, you link the Dispatcher, Solver, and Concert Technology libraries to your application. The command that you use for linking depends on your platform. In the standard distribution of the product, you will find a file that contains details appropriate to your platform and that points you toward a subdirectory containing a suitable makefile or project. If you are using UNIX, you will find the information in `YourDispatcherHome/README`; on a PC, you will find comparable information in `YourDispatcherHome/readme.htm`

Typographic and naming conventions

The names of types, classes, and functions defined in the library begin with `Ilo` or `Ilc`. Those beginning with Ilo are predefined modeling and solving classes, functions, and types provided with Dispatcher, Solver, and Concert Technology. You can use classes, functions, and types in the Solver library that begin with Ilc to customize your code and extend the library.

The name of a class is written as concatenated words with the first letter of each word in upper case. 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 IloSolver sol);
};
```

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

To make porting easier from platform to platform, Concert Technology isolates characteristics that vary from system to system. For that reason, use the following names for basic types in C++:

- `IloInt` stands for signed long integers
- `IloAny` stands for pointers (`void*`)
- `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.

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

Include Files

In an effort to minimize the number of include files (that is, header files) that you have to cope with, Dispatcher provides the basic file `ilodispatcher.h`. If you include it in your application, you can exploit all the functions, classes, and member functions documented in the *IBM ILOG Dispatcher Reference Manual*.

If you prefer to include only some of Dispatcher's header files in your application, a list of the individual files and an explanation of each one's purpose follows:

- `ilonode.h` deals with geographical locations.
- `ilovisit.h` represents visits. These visits can be performed at a depot or at a customer's location.
- `ilodim.h` deals with the dimensions in which quantitative constraints (delays, capacities, distances) are expressed.
- `ilovehicle.h` declares the objects representing vehicles.
- `ilobreak.h` declares the objects representing vehicle breaks.
- `ilogoals.h` contains declarations for the primitives that generate a first solution to a problem.
- `lsearch.h` contains declarations for the iterative improvement techniques used to improve the solution of a given problem.
- `ilodist.h` declares classes and functions related to distance and vehicle equivalence.
- `ilorsol.h` contains declarations for the routing solution class.

IBM software support handbook

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

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

Accessing software support

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

IBM Customer Number

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

Company name

Contact name

Preferred means of contact (voice or email)

Telephone number where you can be reached if request is voice

Related product and version information

Related operating system and database information

Detailed description of the issue

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

Contact via web

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

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

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

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

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

Contact via phone

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

Part I

The Basics

This part consists of the following lessons:

- Chapter 1, *IBM ILOG Dispatcher Concepts*
- Chapter 2, *Modeling a Vehicle Routing Problem*
- Chapter 3, *Solving a Vehicle Routing Problem*
- Chapter 4, *Minimizing the Number of Vehicles*
- Chapter 5, *Adding Visit Disjunctions*
- Chapter 6, *Multiple Tours per Vehicle*

C H A P T E R

1

IBM ILOG Dispatcher Concepts

In this lesson, you will learn how to:

- use the three-stage method to describe, model, and solve problems
- learn how to use Dispatcher classes to model routing problems
- understand how to solve routing problems with local search

IBM® ILOG® Dispatcher is a C++ library that makes it easier to develop applications for routing problems. It is based on IBM ILOG Concert Technology and IBM ILOG Solver, a generic tool for object-oriented constraint programming. One of the key advantages of constraint programming lies in the fact that it dissociates the *representation* of the problem, called the *model*, from the search algorithms used to *solve* it.

To find a solution to a routing problem using Dispatcher, 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” on page 26.

The second stage is to use Dispatcher classes to model the problem. The model is composed of basic objects: *nodes*—or geographic locations, *visits*, and *vehicles*. These objects are associated with *dimensions*, such as weight, time, and distance. The *decision variables* in a Dispatcher model are the variables representing if a visit is performed—*performed-variables*—and the variables representing the visit immediately after a given visit—*next-variables*. A solution to a routing problem—a *routing plan*—is defined by the values assigned to each of these decision variables. A Dispatcher model also includes *side constraints*, such as capacity constraints, time windows, and precedence constraints. The *objective* is to lower the cost of a routing plan. The total cost of a routing plan is determined by summing the total costs for all vehicles and adding any costs related to unperformed visits. For more information, see the section "Model" on page 27.

The third stage is to use local search to solve the routing problem. Dispatcher uses a two-phase method to solve problems. The first phase consists of *generating a first solution* that satisfies the problem. In the second phase, you improve this first solution using *local search*. For more information, see the section "Solve" on page 39.

This first lesson is designed to help you understand the basic concepts of Dispatcher. This overview is somewhat theoretical; it presents the definitions of terms that appear in the rest of the manual. In later lessons, you will work through problems by describing, modeling, and solving routing problems. In these lessons, you will learn how to use the classes and functions that you learn about here.

Describe

The first stage is to describe the problem in natural language. *Routing* can be defined as the process of assigning *visits* to *vehicles*, while taking into account constraints such as *capacity* or *time windows,* to produce a *routing plan* with the least possible *cost.*

The complexity of routing problems varies greatly. The Vehicle Routing Problem (VRP) requires a *set* of vehicles to visit a *set* of customers from one or more depots. To solve a VRP you have to solve two intertwined problems. An assignment problem—whether or not to assign a particular visit to a given vehicle—is added to the geometry problem, where each vehicle has to visit a given set of customers as expeditiously as possible. A second type of routing problem, the Pickup and Delivery Problem (PDP), extends the VRP by allowing pickups and deliveries within the same route.

Each class of routing problem—whether VRP, PDP, and so on—itself comes in many guises. For example, the VRP may be modified by introducing multiple depots or alternative delivery sites; by requiring vehicles to make multiple round trips or tours; or by coupling the vehicle with a technician.

A problem may include different kinds of side *constraints*—such as capacity, time windows, precedence, technician skill levels, and so on—or specific constraints that depend on the business area of the application. Constraints may be added to allow for work breaks, such as lunch and coffee breaks or overnight stops, to minimize the number of vehicles, or to introduce costs for late deliveries or technical service.

There also exists wide variation in the *size* of routing problems. The size may vary from a few dozen activities to thousands of visits. The methods that work well for small problems may not be applicable to bigger problems, due to the increased complexity.

Besides the diversity in the possible data of routing problems, different environments may require different performance from the algorithms solving the problems. For example, in some applications the construction of a routing plan must be performed in seconds while in other applications computation times of a few days are allowed. Similarly, any feasible solution is sufficient for many applications, whereas others require near-optimal solutions.

In each lesson in this manual, you will describe the routing problem before modeling and solving it. Though the Describe stage of the process may seem trivial in certain simple problems, you will find that taking the time to fully describe a more complex, real-world 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 before writing your application.

Model

The second stage is to use the Dispatcher library of C++ classes and functions to implement the concepts of nodes, visits, and vehicles in terms of constrained Solver variables and constraints.

Basic Modeling Objects

Dispatcher models include the following basic objects: nodes, visits, and vehicles.

Nodes

Routing problems have a geometric component: the visits must be performed at specific physical locations.

Dispatcher represents these physical locations as nodes. These nodes are then used to compute distances and times (and subsequently cost) between customers and depots.

Theoretically, a *node* represents an intersection of roads or some other place where a vehicle can stop, such as the buildings—factories, offices, houses, and so on—where visits are made. Nodes can be *depots* (where the vehicles pick up the goods to be delivered) or *customer locations* (where the goods or services are delivered). A node may be defined by coordinates that give its location.

The following code shows how to create a node at coordinates `x` and `y`.

```
IloEnv env;
IloModel mdl(env);
IloNum x=4, y=-5;
IloNode node(env, x, y);
```

***Note:** For more information on the classes and functions in the example code, see Chapter 2, Modeling a Vehicle Routing Problem and Chapter 3, Solving a Vehicle Routing Problem. You can also refer to the IBM ILOG Dispatcher Reference Manual.*

Visits

A *visit* represents an activity that the vehicle has to perform. A visit occurs at a single node or at a single arc (composed of two nodes), and it is performed by only one vehicle. You can create many different visits at the same node.

The following code shows how to create a visit associated with a customer node.

```
IloEnv env;
IloModel mdl(env);
IloNode node(env);
IloVisit visit(node);
mdl.add(visit);
```

Visits have *quantities*, which can be weights, volumes, numbers of objects, and so on. These quantities represent the amount of a good picked up or dropped off by the vehicle at the visit.

Vehicles

Vehicles are resources that convey goods from one visit to another. A vehicle has a start and an end visit and can have variable start and end times associated with those visits. In Dispatcher, a vehicle can be created without start and end visits; this means the vehicle starts and ends at a “dummy” node which is at distance 0 of all the other nodes.

The following code shows how to create a vehicle associated with the locations where it must start and end its route.

```
IloEnv env;
IloModel mdl(env);
IloNode depot(env);
IloVisit startVisit(depot);
IloVisit endVisit(depot);
IloVehicle vehicle(startVisit, endVisit);
mdl.add(vehicle);
```

Vehicles can be given *capacities*. These capacities represent the total quantity the vehicle can carry. Capacity, like quantity, can be a weight, volume, number of objects, and so on.

Decision Variables

From a modeling perspective, each visit `v` is associated with a next-variable, representing the visit immediately after `v` (accessible through the member function `IloVisit::getNextVar`) and a performed-variable which is set to `IloTrue` if the `v` is performed and `IloFalse` if it is not (accessible through the member function

`IloVisit::performed`). These variables are the *decision variables* of a routing problem: a solution to a routing problem is uniquely described by the values they take.

Each time a visit is extracted it is associated with a unique integer identifier (accessible through the member function `IloDispatcher::getIndex`). The next-variables are extracted to instances of `IlcIntVar` which represent indices of visits.

To simplify programming with Dispatcher, the next-variables have been complemented by redundant variables. The prev-variables, accessible through the member function `IloVisit::getPrevVar`, are the constrained variables representing the visit immediately before a given visit. They are also extracted to instances of `IlcIntVar` representing indices of visits.

Another constrained variable associated with each visit is the vehicle variable (accessible through `IloVisit::getVehicleVar`) which holds the vehicle performing the visit. As with visits, once extracted, vehicles are associated with a unique integer identifier and vehicle-variables are extracted to instances of `IlcIntVar` which correspond to these indices.

Vehicles are also associated with a number of variables and objects. The start-visit and the end-visit (accessible through `IloVehicle::getFirstVisit` and `IloVehicle::getLastVisit`, respectively) represent the starting and ending points of the vehicle's route.

Dimensions

Dimensions are objects closely associated with visits and vehicles. The most common dimensions are weight, time, and distance. Dimensions are used to model side constraints such as capacity, time windows, deadlines, service delays, and so on. Dimensions are also used to model costs and the objective. An understanding of dimensions and the constrained variables derived from them is essential to understanding Dispatcher.

Vehicles may have different *capacities* in terms of weight (for solid goods, for example) or volume (for liquids, for instance); routes may entail different costs or distances traveled; visits may require different amounts of time (perhaps for waiting, unloading, reloading, and so on). In short, there may be many different dimensions (such as weights, volumes, distances, times) in a given routing problem. These dimensions are most closely associated with vehicles (to model capacity and costs) and visits (to model quantities, delays, time windows, and deadlines).

The class `IloDimension` makes it possible to handle easily the various dimensions that occur in a problem. When you create an instance of `IloDimension`, a constrained variable is associated with this dimension for each object (if needed). Constraints can subsequently be posted on those variables.

The class IloDimension has two subclasses to represent the *intrinsic* and *extrinsic* dimensions of an object. An intrinsic dimension depends only on the single object with which it is associated. An extrinsic dimension depends on two objects.

The subclass IloDimension1 represents the dimensions that are intrinsic to an object. For example, weight is represented by IloDimension1 because the weight of an object depends only on that object.

IloDimension2 represents the dimensions that are extrinsic to an object; that is, those dimensions that depend on something outside the object itself. For example, time is usually represented as an instance of IloDimension2 because the time to travel from one visit to another depends on both those visits.

By definition, IloDimension2 is closely linked to the concept of *distance*. For that reason, one of the data members of IloDimension2 is in fact an instance of IloDistance.

Objects of IloDistance define how distances are computed between nodes with respect to a dimension and a vehicle. IloEuclidean computes the Euclidean distance between nodes according to their coordinates. IloManhattan computes the grid pattern distance between nodes. IloDispatcherGraph computes distance from a road network graph. A simple C++ representation of IloDistance could be a pointer to a function. The following code shows how to create three dimensions: weight, time, and distance:

```
IloEnv env;
IloModel mdl(env);
IloDimension1 weight(env);
mdl.add(weight);
IloDimension2 time(env, IloEuclidean);
mdl.add(time);
IloDimension2 distance(env, IloEuclidean);
mdl.add(distance);
```

Using dimensions to model side constraints

In Dispatcher, side constraints such as capacity, time windows, and deadlines are implemented by associating one floating-point constrained variable per dimension with each visit. These *cumulative variables* represent the accumulation of the dimension from the beginning of the vehicle route to the visit. Constraints can then be placed on these variables. They can be accessed through the member function IloVisit::getCumulVar.

While the cumulative variable is accessed through the visit, it is also linked to a specific vehicle—the accumulation of a dimension, such as weight, is accumulated as one vehicle makes visits along its route.

For example, if a truck starts empty from the depot and performs 4 visits, where it collects goods of weight 4, 7, 8, and 3, respectively, the cumulative variable associated to the dimension weight at the fourth visit is equal to 4+7+8 = 19 (the cumulative variable corresponds to the quantity in the truck upon arrival at the visit). The constraint placed on this cumulative variable is that it must be always less than or equal to the capacity of the vehicle.

Dispatcher also uses *transit variables* to model side constraints. The transit variable associated with a visit is a floating-point constrained variable which represents the change in the cumulative variable between that visit and the following visit. It can be accessed through the member function `IloVisit::getTransitVar`.

Cumulative and transit variables for dimensions—instances of both `IloDimension1` and `IloDimension2`—are defined by the *path constraint*. The path constraint is functionally equivalent to one constraint of the following form for dimension `d`, and each pair of visits `v1` and `v2:`

```
mdl.add(IloIfThen(env,
                  v1.getNextVar()   == v2,
                  v2.getCumulVar(d) == v1.getCumulVar(d) + v1.getTransitVar(d)
                  ));
```

For instances of `IloDimension2`, such as time, the transit variable is defined as the sum of the delay variable, the wait variable, and the travel variable. For a visit `v` and a dimension `d`, the transit variables are maintained by the following rule:

```
v.getTransitVar(d) == v.getDelayVar(d) + v.getTravelVar(d) + v.getWaitVar(d);
```

The delay variable of `v1`, returned by `v1.getDelayVar(d)`, is a floating-point constrained variable which represents the delay in terms of dimension `d` at visit `v1`. This is useful to represent the time needed to unload a truck. It can be accessed through the member function `IloVisit::getDelayVar`.

The wait variable of `v1`, returned by `v1.getWaitVar(d)`, is a floating-point constrained variable which represents the additional quantity of dimension `d` consumed between `v1` and its successor, over the time required to serve `v1` and travel from `v1` to its successor. In the case where `d` represents time, `v1.getWaitVar(d)` represents the waiting time. It can be accessed through the member function `IloVisit::getWaitVar`.

The travel variable of `v1`, returned by `v1.getTravelVar(d)`, is a floating-point constrained variable which represents the quantity of dimension `d` taken by the vehicle serving `v1` to get to `v2`. It can be accessed through the member function `IloVisit::getTravelVar`. The travel variable is maintained by the following rule:

```
mdl.add(IloIfThen(env,
                  v1.getNextVar()    == v2 && v1.getVehicleVar() == veh,
                  v1.getTravelVar(d) == v1.getDistanceTo(v2, d, veh)
                  / veh.getSpeed(d)
                  ));
```

In the previous code, `v1.getVehicleVar().getSpeed(d)` returns the speed of the vehicle associated with the vehicle performing `v1`.

> ***Notes:*** *The introduction of a vehicle break can change the value of both wait and cumulative variables. For more information on vehicle breaks, see Chapter 8, Adding Vehicle Breaks.*
>
> *To avoid getting infinite cumuls on* `IloDimension1` *variables, make sure you do the following: check for infinite bounds on vehicles' first and last visit dimension variables; check for infinite bounds on visit transits (check for forgotten transits); and try to instantiate the vehicle's first cumul when possible, which will automatically instantiate cumuls along routes.*

Modeling visit quantity constraints

Visits have quantities, which can be expressed using `IloDimension1` objects. These quantities, which can be weights, volumes, numbers of objects, and so on, represent the amount of a good picked up or dropped off by the vehicle. A visit can have more than one quantity associated with it. The quantity can be a positive or negative value or a variable. The demand for each visit must also be defined in terms of dimensions and constraints.

For example, to model a visit quantity, you first define the dimension `quantity` and then set a constraint on the transit variable associated with `quantity` using the member function `IloVisit::getTransitVar`. Using the member function `setBounds` is a bit more efficient in terms of memory and speed than using an equality or range constraint. However you might need to use a constraint if you want to give different values to the transit variable in different models or if you want to metapost the constraint. The following code shows how to define `quantity` (an `IloDimension1`) and how to express the quantity of goods related to `visit`:

```
IloEnv env;
IloModel mdl(env);
IloNode node(env);
IloDimension1 quantity(env);
mdl.add(weight);
IloVisit visit(node);
mdl.add(visit);
visit.getTransitVar(quantity).setBounds(12,12); // 12 items
```

Modeling vehicle capacity constraints

Vehicles have capacity; they cannot hold more than a certain weight or a given number of pallets, for example. To express this idea, you define the *dimension* in which the capacity will be expressed, and you post a *constraint* on the variable that represents capacity.

For example, to model a vehicle capacity, you first define the dimension `weight` and then set a constraint on the dimension using the member function `IloVehicle::setCapacity`.

The following code shows how to define `weight` (an `IloDimension1`) and how to express the capacity of a truck in terms of this intrinsic dimension.

```
IloEnv env;
IloModel mdl(env);
IloDimension1 weight(env);
mdl.add(weight);
IloVehicle vehicle(env);
vehicle.setCapacity(weight, 15000); // weight associated with vehicle
                                    // it has a capacity of 15000 kg
```

Capacity constraints correspond to constraints on the bounds of the cumulative variables. For example, the maximum load of a truck is handled internally by Dispatcher by setting bounds on all the cumulative variables associated with weight. The constraint placed on this cumulative variable is that it must be always less than or equal to the capacity of the vehicle.

Modeling time window constraints

Time windows occur in problems from many business sectors. For example, courier services have to pickup parcels after a certain time, but before another. Likewise, deliveries usually have to be made within a given time window—before one hour, but after another.

For example, to model a time window, you first define the dimension `time` and then set a constraint on the cumulative variable associated with `time` using the member function `IloVisit::getCumulVar`. To see how to implement a time window with one of these cumulative variables, assume that a dimension, node and visit have been defined already, like this:

```
IloEnv env;
IloModel mdl(env);
IloDimension2 time(env, IloEuclidean);
IloInt x=4, y=-5;
IloNode node(env, x, y);
IloVisit visit(node);
mdl.add(visit);
```

To define a time window between 10 and 14, use the following code:

```
mdl.add(visit.getCumulVar(time) >= 10);
mdl.add(visit.getCumulVar(time) <= 14);
```

You can also define a time window between 10 and 14 like this:

```
mdl.add(10 <= visit.getCumulVar(time) <= 14);
```

Not all time windows consist of a continuous block of time, of course. There are situations where the acceptable time window occurs in pieces. These are called *disjoint time windows*. Deliveries that can be made only between 7 and 9 in the morning or between 5 and 7 in the evening are an example of disjoint time windows.

To represent a disjoint time window, Dispatcher provides the `IloDimensionWindows` class. Using this class you can represent disjoint feasible periods of time. For example, to represent two disjoint time windows, the first starting at 8 and ending at 10 and the second starting at 14 and ending at 16, you could write:

```
IloDimensionWindows windows(env, time);
windows.setBounds(8, 16);
windows.setForbiddenInterval(10, 14);
```

You can then apply this window to a visit using the `IloExecutionWindowsToVisitCon` constraint:

```
mdl.add(IloExecutionWindowsToVisitCon(visit, windows));
```

Modeling service delays constraints

A *delay* is often encountered when performing a visit. This delay can be expressed in terms of distance (for example, to travel inside a large factory) or in terms of time (to unload the truck, for instance).

For example, to model a service delay, you first define the dimensions `time` and `volume`. You set a constraint on the transit variable associated with `volume` using the member function `IloVisit::getTransitVar`. You then set a constraint on the delay variable associated with `time` using the member function `IloVisit::getDelayVar`. The last line of code expresses that the time needed to unload the truck is equal to 0.5 units of time per unit of volume.

```
IloEnv env;
IloModel mdl(env);
IloDimension1 volume(env);
mdl.add(volume);
IloDimension2 time(env, IloEuclidean);
mdl.add(time);
IloNode node(env);
IloVisit visit(node);
mdl.add(visit);
mdl.add(visit.getTransitVar(volume) == 10);     // 10 m3
mdl.add(visit.getDelayVar(time) == .5*visit.getTransitVar(volume));
                                                // .5 minute / m3
```

Modeling deadline constraints

It may be useful to set deadlines on vehicles. For example, in some problems the vehicle must return to the depot before a certain time of the day.

For example, to model deadlines, you first define the dimension `time`. You also use the member functions `IloVehicle::getFirstVisit` and `IloVehicle::getLastVisit` together with `IloVisit::getCumulVar`.

```
IloEnv env;
IloModel mdl(env);
IloDimension2 time(env, IloEuclidean);
IloNode depot(env);
IloVisit first(depot);
IloVisit last(depot);
IloVehicle vehicle(first, last);
mdl.add(vehicle);
const IloNum timeDeadline = 17;
```

In the following code fragment, `vehicle` is constrained to start at 8:

```
mdl.add(first.getCumulVar(time) == 8);
```

and to come back before 17:

```
mdl.add(last.getCumulVar(time) <= timeDeadline);
```

Modeling other side constraints

Dispatcher provides specific constraints that enable you to model other important side constraints, such as lunch breaks or visit disjunctions.

Modeling vehicle breaks

A *vehicle break* is a period of time in which a vehicle is not available to make a visit, such as during the driver's lunch period. In Dispatcher, breaks are modeled as constraints. A *break* is performed by a vehicle on a dimension (usually time). The break has a start time, a duration, and a position in the route, all of which can vary. A break can interrupt a customer visit or not, as desired.

To create a break where `vehicle` is the vehicle performing the break, `time` is the dimension, `startLunch` is the start variable, and `1.0` is the duration:

```
IloNumVar startLunch(env, 12.0, 13.0);
IloVehicleBreakCon lunch(vehicle, time, startLunch, 1.0)
```

Once the break is created it is added, just like any constraint:

```
mdl.add(lunch);
```

After a break has been created and added to the routing plan, it must be instantiated. In Dispatcher, this must be done when a move is taken. The following goal can be used for doing so:

```
IloInstantiateVehicleBreaks(env);
```

More than one break can be specified per vehicle, and breaks can be involved in meta-constraints. For example:

```
mdl.add(lunch && (coffee1 || coffee2));
```

For more information, see Chapter 8, *Adding Vehicle Breaks*.

When vehicle breaks are allowed to interrupt customer visits, two other visit-related variables come into play. These are the duration variable and the end-cumul variable.

The duration variable represents the real time taken to process the visit. When a visit cannot be broken by vehicle breaks, this is simply equal to the value of the delay variable of the visit. However, in the case where visits can be so interrupted, the value of the duration variable of a visit is equal to the value of the delay variable plus the durations of any vehicle breaks which interrupt the visit.

At all times, the end-cumul variable is equal to the cumul variable plus the duration variable. It represents the time at which processing of the visit is complete.

Modeling visit disjunctions

There are times when only one of two visits can be performed, or when alternative sites for a delivery exist. To solve problems with these conditions, visit disjunctions can be created.

In Dispatcher, a visit can be *performed* (assigned to a vehicle) or *unperformed*. These constraints can be used to set up visit disjunctions.

There are two constraints for stating that a visit has to be performed or not:

```
IloConstraint IloVisit::performed() const;
IloConstraint IloVisit::unperformed() const;
```

If there are two visits, and only one visit can be performed, a disjunction can be created as follows:

```
mdl.add(visit1.performed() != visit2.performed());
```

For more information, see *Chapter 5, Adding Visit Disjunctions.*

Modeling costs and the objective

The total cost of a routing plan is determined by summing the total costs for all vehicles and adding any costs related to unperformed visits. The objective is usually to minimize the cost of the routing plan, though sometimes the objective may also include other goals, such as minimizing the number of vehicles (see Chapter 4, *Minimizing the Number of Vehicles*).

The cost variable of the entire routing plan can be obtained with `IloDispatcher::getCostVar()`.

Costs associated with visits

Visits can have a *penalty cost* associated with them, which represents the cost of *not* performing the visit. Penalty costs can be expressed by the member function `IloVisit::setPenaltyCost(penaltyCost)`. By default, this cost is set to `IloInfinity`, which means that the visit *must* be performed.

Costs associated with vehicles

The total cost of a vehicle is determined by summing the cost for each dimension specified for the vehicle and adding any fixed costs for that vehicle. If a fixed cost is specified, it is added to the cost of the vehicle if the vehicle is in use (that is, if the vehicle performs any visits other than its first and last visits). The cost variable of a vehicle can be obtained with the member function `IloVehicle::getCostVar()`.

Cost can vary with many different dimensions: the distance traveled, the time spent, or the number of stops. Costs in Dispatcher are attached to vehicles via `IloVehicle::setCost(IloDimension dim, IloNum unitCost)`. *Fixed costs* are expressed by the member function `IloVehicle::setCost(fixedCost)`.

The following code shows how to express a cost in terms of distance.

```
IloEnv env;
IloModel mdl(env);
IloDimension2 distance(env, IloEuclidean); // in mi.
mdl.add(distance);
IloVehicle vehicle(env);
vehicle.setCost(distance, 3.2); // $ 3.2 per mile
```

The statement `vehicle.setCost(distance, 3.2)` means that for each unit of distance that the vehicle travels, 3.2 units of cost are accrued. Cost for the vehicle is computed by multiplying the usage variable of the dimension (in this case, `distance`) by the coefficient (3.2 in this case). If cost is specified in more than one dimension (for instance, `time` and `distance`), the cost for each dimension specified for the vehicle is computed in the same fashion.

Vehicle cost is bound as follows: the transit variables of all visits performed by the vehicle (including the first and last visits representing the start and end points of the vehicle), are added together to give the usage of the dimension for the vehicle. This usage is also reinforced by bounds computed from the cumulative variable at the last visit plus the transit variable at the last visit, minus the cumulative variable at the first visit, as follows. Suppose `time` is an `IloDimension2`:

```
IloVisit first = vehicle.getFirstVisit();
IloVisit last = vehicle.getLastVisit();
IloNumVar usage = last.getCumulVar(time) + last.getTransitVar(time) -
                    first.getCumulVar(time);
```

For more information on modeling costs, see Chapter 11, *Modeling Complex Costs*.

Summary: The Dispatcher Model

The different objects and variables associated with vehicles and visits are summarized in the two following tables.

Table 1.1 *Objects Associated with a Vehicle*

Object Name	Accessor
capacity	`IloNum IloVehicle::getCapacity(IloDimension1 d) const;`
cost	`IloNumVar IloVehicle::getCostVar() const;`
first visit	`IloVisit IloVehicle::getFirstVisit() const;`
index	`IloInt IloDispatcher::getIndex(IloVehicle) const;`
last visit	`IloVisit IloVehicle::getLastVisit() const;`
speed	`IloNum IloVehicle::getSpeed(IloDimension2 d) const;`

Table 1.2 *Objects Associated with a Visit*

Object Name	Accessor
cumul variable	`IloNumVar IloVisit::getCumulVar(IloDimension d) const;`
delay variable	`IloNumVar IloVisit::getDelayVar(IloDimension2 d) const;`
index	`IloInt IloDispatcher::getIndex(IloVisit) const;`
next-variable	`IloVisitVar IloVisit::getNextVar() const;`
prev-variable	`IloVisitVar IloVisit::getPrevVar() const;`
transit variable	`IloNumVar IloVisit::getTransitVar(IloDimension d) const;`

Table 1.2 Objects Associated with a Visit

Object Name	Accessor
travel variable	`IloNumVar IloVisit::getTravelVar(IloDimension2 d);`
vehicle variable	`IloVehicleVar IloVisit::getVehicleVar() const;`
wait variable	`IloNumVar IloVisit::getWaitVar(IloDimension2 d);`

Solve

Dispatcher uses a two-phase approach for solving routing problems. The first phase consists of generating a solution that satisfies the problem. In phase two, you improve this first solution using local search.

Solving a routing problem can be translated into the simple question: "Who does what?" In the case of the routing problem, this question can be answered by giving the list of visits performed by each vehicle. This is a routing plan, which can be represented and accessed in several ways:

◆ through an instance of the class `IloModel`, which contains all the objects of the routing plan, such as visits and vehicles;

◆ through an instance of the class `IloDispatcher`, to access or modify the values of the model's variables during or after a search;

◆ through an instance of the class `IloRoutingSolution`, to access the routing plan resulting from a search and stored in a solution.

To allow dynamic problem solving, optimization in Dispatcher is based on first generating a solution, and then improving it using local search methods. As its name implies, local search acts by locally modifying a given solution to decrease the cost using neighborhoods. The central idea of a neighborhood is to define a set of solution changes that represent alternative moves that can be taken. In the case of vehicle routing, this translates into modifying the next-variables of some visit variables.

To achieve this in Dispatcher, a double representation of the problem is used. A passive representation based on instances of `IloRoutingSolution`, depicting the last valid solution encountered, is associated with each decision variable in the problem—the next-vars, prev-vars and performed-vars. The saved values of the passive representation can be accessed through member functions such as `IloRoutingSolution::getNext(IloVisit)`, `IloRoutingSolution::getPrev(IloVisit)` and `IloRoutingSolution::isPerformed(IloVisit)`.

An active representation of the problem is used to search for a new, modified solution using neighborhoods.

Thus, the process for solving a problem using Dispatcher is:

1. Generate, create, or input a first solution.
2. Store the first solution in an instance of `IloRoutingSolution`.
3. Generate a new, modified, solution using a neighborhood.
4. Set the decision variables affected by the move operator to their new values.
5. Set the remaining decision variables to their saved values.
6. Propagate the constraints.
7. If the propagation fails, the new solution is refused. Otherwise, the solution is accepted and the decision variables of the new solution are saved.
8. The domains of variables are restored to their initial values on backtracking.

Generating a first solution

In a normal Dispatcher application, you would generate a first solution using one of Dispatcher's predefined first solution generation functions. For more information, see Chapter 3, *Solving a Vehicle Routing Problem* and Appendix A, *Predefined First Solution Heuristics*. You could also load a first solution from a file or write your own first solution generation method. For more information, see Chapter 17, *Developing Your Own First Solution Heuristics*.

To demonstrate the concepts involved in local search, imagine that you have a simple routing problem with one vehicle, one depot, and three customer visits.

```
IloEnv env;
IloModel mdl(env);
IloNode depot(env, 0.0, 0.0);
IloNode n1(env, 1.0, 1.0);
IloNode n2(env, 1.0, 0.0);
IloNode n3(env, 0.0, 1.0);
IloVisit visit1(n1, "Visit 1");
IloVisit visit2(n2, "Visit 2");
IloVisit visit3(n3, "Visit 3");
mdl.add(visit1);
mdl.add(visit2);
mdl.add(visit3);
IloVisit first(depot, "Start"), last(depot, "End");
IloVehicle w(first, last);
mdl.add(w);
```

You set the cost of the vehicle to be proportional to the distance traveled `length`.

```
IloDimension2 length(env, IloEuclidean);
w.setCost(length, 1);
```

You input a first solution where the vehicle starts at the depot, makes visit 1, then visit 2, then visit 3, and finally returns to the depot. The cost of this routing plan would be 4.83 ($2 + 2\sqrt{2}$). The following figure shows this routing plan.

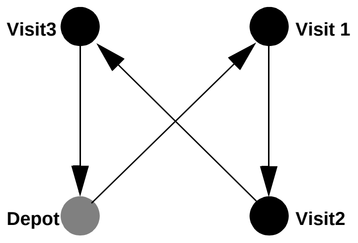

Figure 1.1 *First Routing Solution Created from Scratch*

Improving the solution using local search

You then use local search to modify the next-variables of some variables and search for a routing plan with a lower cost. To do this, you use Dispatcher's predefined neighborhoods and search heuristics to guide how the alternative moves in a neighborhood are taken. For more information, see Chapter 3, *Solving a Vehicle Routing Problem* and Appendix B, *Predefined Neighborhoods*.

The improved solution has a cost of 4. The following figure shows this routing plan.

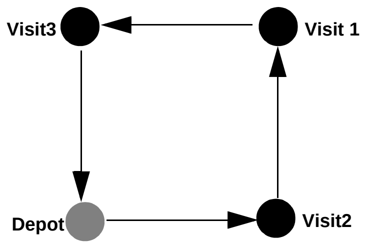

Figure 1.2 *Solution Improved with Local Search*

Displaying the solution

Each vehicle is associated with a *route*. For each vehicle, the route it serves is identified by the first visit and last visit. More formally, a route is a set of visits that are served by one vehicle.

The route of a given vehicle is accessible through an *iterator*, an instance of the class `IloDispatcher::RouteIterator` for iterating on the route depicted by the state of the extracted variables or `IloRoutingSolution::RouteIterator` for iterating on the route saved in a solution. Both iterators are initialized with a vehicle. `RouteIterator::operator++` moves from one visit in the route to the next visit, while `RouteIterator::operator*` returns the current visit pointed to by the iterator. The following code shows how to use such iterators to display the route associated with a vehicle.

```
IloSolver solver(mdl);
IloDispatcher dispatcher(solver);
for(IloDispatcher::RouteIterator ri(dispatcher, vehicle); ri.ok(); ++ri) {
  IloVisit visit = *ri;
  solver.out() << visit.getName() << " ";
}
solver.out() << endl;
```

C H A P T E R

2

Modeling a Vehicle Routing Problem

In this lesson, you will learn how to:

- ◆ model a vehicle routing problem
- ◆ create dimensions, nodes, visits, and vehicles
- ◆ use IBM® ILOG® Concert Technology’s csv reader functionality
- ◆ use the classes `IloDimension1`, `IloDimension2`, `IloCsvReader`, `IloNode`, `IloVisit`, and `IloVehicle`

You will learn how to model a simple vehicle routing problem (VRP). To find a solution to a problem using IBM® ILOG® Dispatcher, you will use the three-stage method: describe, model, and solve. This lesson shows how to describe and model a VRP. The next lesson is a continuation of this one and shows how to solve a VRP.

Describe

The problem of delivery by multiple vehicles is known as the Vehicle Routing Problem (VRP). In a VRP, the goal is to build a *set* of routes, so that each visit is performed exactly once. The aim is to minimize the global cost of these routes. The side constraints of a VRP, such as time windows or capacity constraints, make it necessary to create several routes. A solution to a VRP assigns customers to routes and builds those routes.

The following figure shows a sample solution for a VRP:

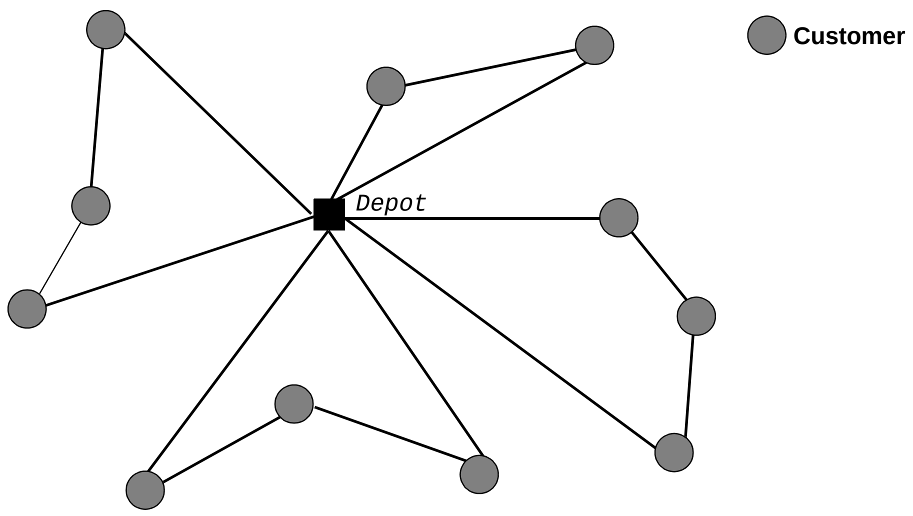

Figure 2.1 ***Example of a Solution for a Vehicle Routing Problem***

Step 1 Describe the problem

The first step is to write a natural language description of the problem.

The components of the routing model for this VRP include: vehicles, customers, and a depot.

What are the constraints in this problem?

◆ Capacity constraints on vehicles. Vehicles can carry up to a maximum amount of goods, which may be measured by weight, volume, number of parcels, and so on.

◆ Time constraints or windows related to customers. The customer's time window is the period of time during which a vehicle can make the visit.

◆ Opening and closing times at depots.

The objective is to minimize the cost of the delivery of all the goods.

Model

Once you have written a description of your problem, you can use Dispatcher classes to model it. Dispatcher is based on IBM® ILOG® Concert Technology and IBM® ILOG®

Solver. If you are not familiar with these products, please refer to the Concert Technology and Solver documentation.

Step 2 Open the example file

Open the example file `YourDispatcherHome/examples/src/tutorial/vrp_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 of this lesson, you will have completed the program code for modeling the problem. At the end of the next lesson, you will have completed the code for solving the problem and you will be able to compile and run it.

You will use object oriented programming techniques to write this program. In this lesson, you create the `RoutingModel` class, which is used to build the model of the vehicle routing problem. In the next lesson, you will create the `RoutingSolver` class, which is used to solve the problem, and you will also create the main function and compile and run the program.

Declare the RoutingModel class

The code for the declaration of the class `RoutingModel` is provided for you:

```
class RoutingModel {
  IloEnv              _env;
  IloModel            _mdl;
  IloDimension2       _time;
  IloDimension2       _distance;
  IloDimension1       _weight;

  void addDimensions();
  void createIloNodes(const char* nodeFileName);
  void createVehicles(const char* vehicleFileName);
  void createVisits(const char* visitsFileName);

public:
  RoutingModel(IloEnv env, int argc, char* argv[]);
  ~RoutingModel() {}
  IloEnv    getEnv() const { return _env; }
  IloModel  getModel() const { return _mdl; }
};
```

The class `RoutingModel` has five data members. The environment, an instance of the class `IloEnv`, manages internal modeling issues. It handles output, memory management for modeling objects, and termination of search algorithms. The model, an instance of the class `IloModel`, is a container for modeling objects such as variables and constraints. For more information on environments and models, see the *IBM ILOG Solver Reference Manual*. The dimensions in this problem are time, distance, and weight. For more information on dimensions, see Chapter 1, *IBM ILOG Dispatcher Concepts*.

The class `RoutingModel` has four member functions, `addDimensions`, `CreateIloNodes`, `CreateVehicles`, and `CreateVisits`, as well as a constructor. These member functions and constructor are explained in the following sections. You can use the public member functions `getEnv` and `getModel` to return the environment and model.

Define the addDimensions function

The member function `addDimensions` is used to create the three dimensions—weight, time, and distance—and add them to the model. You must explicitly add the dimensions to the model or Solver will not be able to use them in the search for a solution.

The following code is provided for you:

```
void RoutingModel::addDimensions() {
```

Next, you create the weight dimension.

Step 3 Create the weight dimension and add to model

Add the following code after the comment
`//Create the weight dimension and add to model`

```
  _weight =IloDimension1 (_env, "weight");
  _mdl.add(_weight);
```

The dimension `_weight` is an instance of the subclass `IloDimension1`. Weight is a dimension that is intrinsic to an object and depends only on that object. For example, a vehicle can carry a certain amount of weight or a visit can require a vehicle to drop off or pick up a certain amount of weight. The constructor for `IloDimension1` used in this lesson takes two parameters: the environment and a name used for debug and trace purposes.

Now, you create the time and distance dimensions.

Step 4 Create the time and distance dimensions and add to model

Add the following code after the comment
`//Create the time and distance dimensions and add to model`

```
  _time  =IloDimension2 (_env, IloEuclidean, "time");
  _mdl.add(_time);

  _distance =IloDimension2 (_env, IloEuclidean, "distance");
  _mdl.add(_distance);
}
```

The dimensions _time and _distance are instances of the subclass IloDimension2. Time and distance are both extrinsic to an object—they depend on something outside the object itself. For example, the time to travel from one visit to another depends on both those visits. The constructor for IloDimension2 used in this lesson takes three parameters: the environment, a distance function, and a name used for debug and trace purposes. The distance function used is IloEuclidean, a predefined distance function that returns the Euclidean distance between two locations. You can also use the predefined distance function IloManhattan, which computes the grid distance between locations. You can obtain distances from a graph using IloDispatcherGraph functionality (more on this in Chapter 4, *Minimizing the Number of Vehicles*) or define your own distance functions.

Define the createIloNodes function

As you remember from Chapter 1, *IBM ILOG Dispatcher Concepts*, Dispatcher represents physical locations as nodes. Nodes can be depots, where the vehicles pick up goods to be delivered, or customer locations, where the goods are delivered. These nodes are then used to compute distances and times—and therefore cost—between customers and the depot. In this lesson, a node is defined by coordinates that give its location.

You will use Concert Technology's csv reader functionality to input the node data from a csv file. An instance of the class IloCsvReader is used to read a csv file. The constructor takes two parameters. The first parameter is the environment and the second parameter is the name of the csv file.

Step 5 Create the node csv reader

Add the following code after the comment //Create the node csv reader

```
void RoutingModel::createIloNodes(const char* nodeFileName) {
  IloCsvReader csvNodeReader(_env, nodeFileName);
```

Now, you use the nested class IloCsvReader::Iterator to create an iterator to step through all the lines of the csv data file, except blank lines and commented lines. The IloCsvReader operator * returns the current instance of IloCsvLine, which is the current line in the csv file. You use the member function IloCsvLine::getStringByHeader to return a pointer to the string contained in the field name in the csv line. In this lesson, this name is the name of the customer site or depot.

Step 6 Create the iterator

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

```
  IloCsvReader::LineIterator  it(csvNodeReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char * name = line.getStringByHeader("name");
```

You then create an instance of `IloNode` using the data from the csv file. The constructor for `IloNode` used in this lesson takes five parameters. The first parameter is the environment. The second, third, and fourth parameters are the x, y, and z coordinates of the node. In this lesson, the z coordinate is set to 0. The x and y coordinates are obtained by using the member function `IloCsvLine::getFloatByHeader` to return a reference to the floating-point values contained in the fields `x` and `y` in the csv line. The last parameter is a name used for debug and trace purposes.

Step 7 Create the node

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

```
    IloNode node(_env, line.getFloatByHeader("x"),
        line.getFloatByHeader("y"), 0, name);
```

Next, you set a unique key on the node in order to be able to access it easily when you create vehicles and visits.

Step 8 Set the key

Add the following code after the comment `//Set the key`

```
    node.setKey(name);
```

Then, you move the iterator to the next line in the csv file using the `IloCsvReader::Iterator` operator `++`.

Step 9 Move to the next line in the csv file

Add the following code after the comment
`//Move to the next line in the csv file`

```
    ++it;
  }
```

Finally, you should use the member function `IloCsvReader::end` to deallocate the memory used by the csv reader. This code is provided for you:

```
    csvNodeReader.end();
}
```

Define the createVehicles function

As you remember from Chapter 1, *IBM ILOG Dispatcher Concepts*, vehicles have start and end visits and capacities. In this lesson, you will use Concert Technology's csv functionality to input vehicle data from a csv file. You use the classes `IloCsvReader`, `IloCsvReader::Iterator`, and `IloCsvLine`. This code is provided for you:

```
void RoutingModel::createVehicles(const char* vehicleFileName) {
  IloCsvReader csvVehicleReader(_env, vehicleFileName);
  IloCsvReader::LineIterator  it(csvVehicleReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char * namefirst = line.getStringByHeader("first");
    char * namelast = line.getStringByHeader("last");
    char * name = line.getStringByHeader("name");
    IloNum capacity = line.getFloatByHeader("capacity");
```

Next, you use the static member function `IloNode::Find` to find the nodes associated with the first and last visits the vehicle makes. This static member function takes two parameters: the environment and the key associated to the node.

Step 10 Find the first and last nodes

Add the following code after the comment `//Find the first and last nodes`

```
    IloNode node1 = IloNode::Find(_env, namefirst);
    IloNode node2 = IloNode::Find(_env, namelast);
```

Next, you create the vehicle's first visit, an instance of the class `IloVisit`. In a VRP, the first and last visits of a vehicle usually take place at the depot. The constructor for `IloVisit` takes two parameters: an instance of `IloNode` and a name used for debug and trace purposes.

Step 11 Create the first visit

Add the following code after the comment `//Create the first visit`

```
    IloVisit first(node1, "depot");
```

Then, you add constraints on the first visit to the model. The first constraint is that when the vehicle makes its first visit to the depot, it will not have delivered any weight. You also add the constraint that the vehicle cannot leave the first visit—the depot—until after the depot's opening time.

Step 12 Add constraints to the first visit

Add the following code after the comment `//Add constraints to the first visit`

```
    _mdl.add(first.getCumulVar(_weight) == 0);
    _mdl.add(first.getCumulVar(_time) >= line.getFloatByHeader("open"));
```

Next, you create the vehicle's last visit, which also takes place at the depot. You add the constraint that the vehicle must arrive at the last visit—the depot—before the depot's closing time.

Step 13 Create the last visit and add constraints

Add the following code after the comment
`//Create the last visit and add constraints`

```
    IloVisit last(node2, "depot");
    _mdl.add(last.getCumulVar(_time) <= line.getFloatByHeader("close"));
```

Then you create the vehicle, an instance of the class `IloVehicle`. The constructor for `IloVehicle` takes three parameters. The first two parameters are instances of `IloVisit` representing the first and last visits of this vehicle—in this lesson, these both take place at the depot. The last parameter is a name used for debug and trace purposes.

Step 14 Create the vehicle

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

```
    IloVehicle vehicle(first, last, name);
```

After you create a vehicle, you set a proportional cost associated with operating this vehicle using the member function `IloVehicle::setCost`. This member function takes two parameters. The first parameter is a dimension. In this lesson, the cost is proportional to the distance the vehicle travels. The second parameter associates a unit of cost per unit of dimension. In this lesson, the cost is directly proportional to the distance traveled and this parameter is set to 1.0.

Step 15 Set a cost for vehicle operation

Add the following code after the comment `//Set a cost for vehicle operation`

```
    vehicle.setCost(_distance, 1.0);
```

You also set the capacity of the vehicle using the member function `IloVehicle::setCapacity`.

Step 16 Set the vehicle capacity

Add the following code after the comment `//Set the vehicle capacity`

```
    vehicle.setCapacity(_weight, capacity);
```

When you have finished creating the vehicle, you add it to the model.

Step 17 Add the vehicle to the model

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

```
    _mdl.add(vehicle);
```

Finally, you move to the next line in the csv file. When the end of the file is reached, you deallocate the memory used by the csv reader. This code is provided for:

```
    ++it;
  }
  csvVehicleReader.end();
}
```

Define the createVisits function

As you remember from Chapter 1, *IBM ILOG Dispatcher Concepts*, visits occur at a single node and are performed by only one vehicle. A visit must be associated to a node, its location. A visit also has a quantity—the amount of goods delivered to the location. A visit can have a minimum time and a maximum time during which it can be performed—a time window. Additionally, visits have a drop time—the amount time required to perform the visit.

In this lesson, you will use the csv reader functionality to input this visit data from a csv file. You use the classes `IloCsvReader`, `IloCsvReader::Iterator`, and `IloCsvLine`. This code is provided for you:

```
void RoutingModel::createVisits(const char* visitsFileName) {
  IloCsvReader csvVisitReader(_env, visitsFileName);
  IloCsvReader::LineIterator  it(csvVisitReader);
  while(it.ok()){
    IloCsvLine line = *it;
    //read visit data from files
    char * visitName =  line.getStringByHeader("name");
    char * nodeName = line.getStringByHeader("node");
    IloNum quantity = line.getFloatByHeader("quantity");
    IloNum minTime  = line.getFloatByHeader("minTime");
    IloNum maxTime  = line.getFloatByHeader("maxTime");
    IloNum dropTime = line.getFloatByHeader("dropTime");
```

Next, you use the static member function `IloNode::Find` to find the node associated with the visit.

Step 18 Find the visit node

Add the following code after the comment `//Find the visit node`

```
    IloNode node = IloNode::Find(_env, nodeName);
```

Next, you create a visit using an instance of `IloVisit`. The constructor for `IloVisit` takes two parameters: an instance of `IloNode` and a name for debug and trace purposes. The instance of `IloNode` represents the location of the visit. You add the constraint that the amount of delay time at the visit must equal the drop time. You add the constraint that the amount of weight dropped off at the visit must equal the quantity of goods to be delivered at the visit. You also add the constraint that the visit must be performed within the visit time window.

Step 19 Create the visit and add constraints

Add the following code after the comment
`//Create the visit and add constraints`

```
    IloVisit visit(node, visitName);
    _mdl.add(visit.getDelayVar(_time) == dropTime);
    _mdl.add(visit.getTransitVar(_weight) == quantity);
    _mdl.add(minTime <= visit.getCumulVar(_time) <= maxTime);
```

Then, you add the visit to the model.

Step 20 Add the visit to the model

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

```
    _mdl.add(visit);
```

Finally, you move to the next line in the csv file. When the end of the file is reached, you deallocate the memory used by the csv reader. This code is provided for you:

```
    ++it;
  }
  csvVisitReader.end();
}
```

Define the RoutingModel constructor

The constructor allows you to specify the names of the input files using command line syntax. If you do not specify input files, the defaults will be used. This constructor will be called from the `main` function. It calls the following functions: `addDimensions`, `createIloNodes`, `createVehicles`, and `createVisits`. The following code is provided for you:

```
   RoutingModel::RoutingModel( IloEnv env,
                             int argc,
                             char* argv[]):
  _env(env), _mdl(env) {
  addDimensions();

  //create IloNodes
  char * nodeFileName;
  if(argc < 2)  nodeFileName =
       (char*) "../../../examples/data/vrp20/vrp20nodes.csv";
  else          nodeFileName = argv[1];
  createIloNodes(nodeFileName);

  //create vehicles
  char * vehiclesFileName;
  if(argc < 3)  vehiclesFileName =
       (char*) "../../../examples/data/vrp20/vrp20vehicles.csv";
  else          vehiclesFileName = argv[2];
  createVehicles(vehiclesFileName);

  //create visits
  char * visitsFileName;
  if(argc < 4)  visitsFileName =
       (char*)  "../../../examples/data/vrp20/vrp20visits.csv";
  else          visitsFileName = argv[3];
  createVisits(visitsFileName);
}
```

Now you have finished creating the `RoutingModel` class. In the next lesson, you will create the `RoutingSolver` class and compile and run the program. The complete VRP program is listed at the end of Chapter 3, *Solving a Vehicle Routing Problem*.

Review Exercises

1. To find a solution using Dispatcher, you use the three-stage method: describe, model, and solve. What do each of these stages involve?
2. What are typical side constraints in a vehicle routing problem?

Suggested Answers

Exercise 1

To find a solution using Dispatcher, you use the three-stage method: describe, model, and solve. What do each of these stages involve?

Suggested Answer

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

The second stage is to use Dispatcher classes to model the problem.

The third stage is to use local search to solve the routing problem.

Exercise 2

What are typical side constraints in a vehicle routing problem?

Suggested Answer

A Dispatcher model may include side constraints, such as capacity constraints, time windows, and precedence constraints.

C H A P T E R

3

Solving a Vehicle Routing Problem

In this lesson, you will learn how to:

- solve a vehicle routing problem
- generate a first solution
- improve the solution using local search
- use the classes `IloDispatcher`, `IloRoutingSolution`, `IloGoal`, `IloDichotomize`, `IloRestoreSolution`, `IloSavingsGenerate`, `IloNHood`, `IloTwoOpt`, `IloOrOpt`, `IloRelocate`, `IloCross`, `IloExchange`, `IloSingleMove`, and `IloImprove`

You will learn how to solve a simple vehicle routing problem (VRP). To find a solution to a problem using IBM® ILOG® Dispatcher, you generate a first solution and then improve this solution using local search. This lesson is a continuation of Chapter 2, *Modeling a Vehicle Routing Problem*.

Solve

In many routing problems, good solutions must be computed very quickly. However, the computational complexity of routing in general makes it impractical to use *complete* search methods to obtain optimal solutions except for very small problems.

Therefore, Dispatcher uses a two-phase approach for solving routing problems. The first phase consists of generating a solution that satisfies the problem. In phase two, you improve this first solution using *local search*.

Step 1 Open the example file

Open the example file that you worked on in Chapter 2, *Modeling a Vehicle Routing Problem* in your development environment. In this lesson, you will continue to fill in the blanks at each step. At the end of this lesson, you will have completed the program code and you will be able to compile and run it.

You will use object oriented programming techniques to write this program. In the previous lesson, you created the `RoutingModel` class, which is used to model the problem. In this lesson, you create the `RoutingSolver` class, which is used to solve the problem. You will also create the `main` function and compile and run the program.

Declare the RoutingSolver class

The code for the declaration of the class `RoutingSolver` is provided for you:

```
class RoutingSolver {
  IloEnv              _env;
  IloModel            _mdl;
  IloSolver           _solver;
  IloDispatcher       _dispatcher;
  IloRoutingSolution  _solution;
  IloGoal             _instantiateCost;
  IloGoal             _restoreSolution;
  IloGoal             _goal;

public:
  RoutingSolver(RoutingModel mdl);
  ~RoutingSolver() {}
  IloBool findFirstSolution();
  void improveWithNhood();
  void printInformation(const char* =0) const;
};
```

The class `RoutingSolver` has eight data members. The environment, an instance of the class `IloEnv`, manages internal modeling issues. The model, an instance of the class `IloModel`, is a container for modeling objects such as variables and constraints. An instance of the class `IloSolver` is used to solve the problem expressed in the model. An instance of the class `IloDispatcher` organizes all the details of a routing problem, including routes, vehicles, visits, and costs. An instance of the class `IloRoutingSolution` stores the details of a solution to a routing problem. Three goals, instances of `IloGoal`, are used in the search for a solution. Goals are the mechanism by which Solver implements search strategies. For more information on goals, see the *IBM ILOG Solver User's Manual*.

The class `RoutingSolver` has three member functions, `findFirstSolution`, `improveWithNhood`, and `printInformation`, as well as a constructor. These member functions and constructor are explained in the following sections.

Define the RoutingSolver constructor

The constructor takes an instance of `RoutingModel` as a parameter. The environment, model, solver, dispatcher, and solution are initialized. This constructor will be called from the `main` function. The following code is provided for you:

```
RoutingSolver::RoutingSolver(RoutingModel mdl):
  _env(mdl.getEnv()),
  _mdl(mdl.getModel()),
  _solver(mdl.getModel()),
  _dispatcher(_solver),
  _solution(mdl.getModel()){
```

Next, the three goals used in the search for a solution are created.

Step 2 Create the goal to instantiate cost

Add the following code after the comment `//Create the goal to instantiate cost`

```
     _instantiateCost =
          IloDichotomize(_env, _dispatcher.getCostVar(), IloFalse);
```

You use the function `IloDichotomize` to instantiate the cost variable to its minimum value—Dispatcher's constraints only provide a lower bound on the cost. The cost variable is returned by the member function `IloDispatcher::getCostVar`. This variable is the sum of the cost of the vehicles and of the unperformed visits. The function `IloDichotomize` creates a goal which attempts to assign a value to the variable. To do so, it recursively searches half the domain of the variable at a time. Since you are minimizing the cost variable, you set the last parameter to `IloFalse`, which tries the lower half of the domain first.

Next, you create the goal to restore the solution.

Step 3 Create the goal to restore the solution

Add the following code after the comment
`//Create the goal to restore the solution`

```
      _restoreSolution = IloRestoreSolution(_env, _solution);
```

This goal will be used in the solution improvement phase. Solver will try to iteratively improve the solution. The last solution found will be the lowest cost solution. You will then restore this last solution using the goal `_restoreSolution`.

Now, you create the goal to find a first solution.

Step 4 Create the goal to find a first solution

Add the following code after the comment
`//Create the goal to find a first solution`

```
    _goal = IloSavingsGenerate(_env) && _instantiateCost;
}
```

You use the savings heuristic to create a first solution using the predefined function `IloSavingsGenerate`. Figure 3.1 shows how the savings generation heuristic works. For problems with multiple vehicles, it is very important to consider the trade-off between more vehicles with shorter routes and fewer vehicles with longer routes. For example, the following figure shows two ways of making visits *a* and *b.*

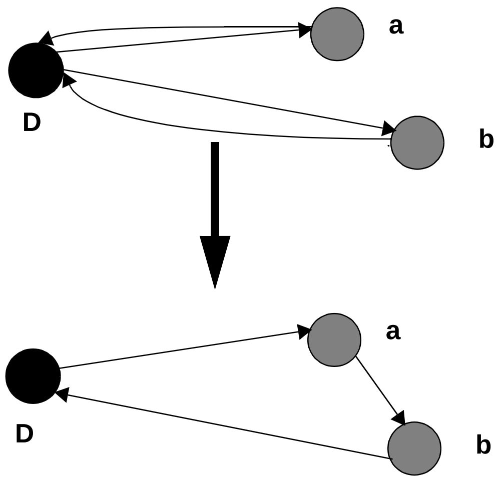

Figure 3.1 *The Savings Generation Heuristic*

The *savings* of serving a and b in the same route, denoted *savings (a, b)*, is defined as *savings (a, b) = cost (a, Depot) + cost (Depot, b) – cost (a, b)*. The savings heuristic generates a solution based on this equation.

You can use other predefined heuristics to find a first solution including: sweep, nearest-to-depot, nearest addition, insertion, and enumeration heuristics. For more information on predefined first solution heuristics, see Appendix A, *Predefined First Solution Heuristics*. The first solution can also be loaded from a file or you can write your own first solution heuristic. For more information, see *Chapter 17, Developing Your Own First Solution Heuristics.*

After the heuristic finds a first solution, you use the goal `_instantiateCost` to find the cost associated with this solution.

Define the findFirstSolution function

Now, you create the function that searches for the first solution.

Step 5 Find the first solution

Add the following code after the comment `//Find the first solution`

```
IloBool RoutingSolver::findFirstSolution() {
  if (!_solver.solve(_goal)) {
    _solver.error() << "Infeasible Routing Plan" << endl;
    return IloFalse;
  }
```

You use the member function `IloSolver::solve` to search for a solution using `_goal`. This goal searches for a first solution using the savings heuristic. It then finds the cost associated with this first solution.

Next, you use the member function `IloRoutingSolution::store` to store the solution and its cost. These will be used in the solution improvement phase.

Step 6 Store the first solution

Add the following code after the comment `//Store the first solution`

```
  _solution.store(_solver);
  return IloTrue;
}
```

Define the improveWithNHood function

After you have found a first solution, you create the neighborhoods you will use in the solution improvement phase. The central idea of the neighborhood is to define a set of solution changes, or deltas, that represent alternative moves that can be taken. Neighborhoods are classified in two groups: those that modify only one route are known as intra-route neighborhoods; those that make changes between routes are known as inter-route neighborhoods. Inter-route neighborhoods can sometimes be used to improve a single route and thus become—strictly speaking—intra-route neighborhoods themselves.

Step 7 Create the neighborhoods

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

```
void RoutingSolver::improveWithNhood() {
  IloNHood nhood =  IloTwoOpt(_env)
                  + IloOrOpt(_env)
                  + IloRelocate(_env)
                  + IloCross(_env)
                  + IloExchange(_env);
  _solver.out() << "Improving solution" << endl;
```

This neighborhood, an instance of `IloNHood`, is composed of a combination of Dispatcher's predefined neighborhoods. These neighborhoods include: Two-Opt, OrOpt, Relocate, Cross, and Exchange. Figure 3.2 shows how the Two-Opt neighborhood works. In this example, the cost is proportional to the length of the route. The move eliminates the crossing by destroying two arcs and creating two new arcs. The resulting route is shorter, and thus less costly. For information about how the other predefined neighborhoods work, see Appendix B, *Predefined Neighborhoods*.

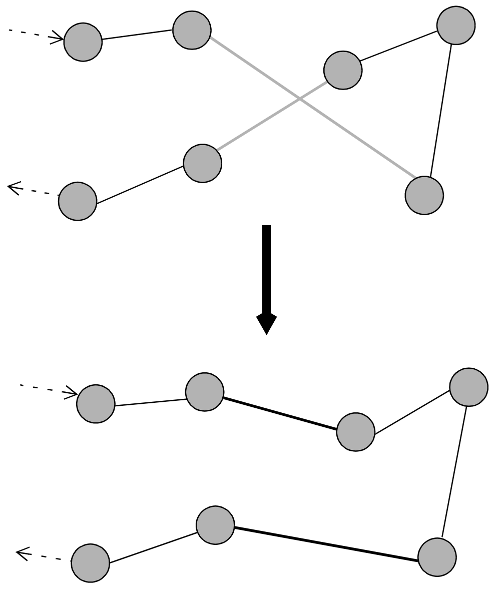

Figure 3.2 *The Two-Opt Neighborhood*

Now, you use iterative improvement techniques to modify the first solution using the neighborhoods you just created. Local search uses search heuristics to guide how the alternative moves in a neighborhood are taken. The search is iterative in that it tries a series of moves in order to decrease the cost of the solution found.

In this lesson, Dispatcher uses the first accept search heuristic. First accept search takes the first legal cost decreasing move encountered. The search continues to take such moves until the neighborhood contains no legal cost-reducing moves. This point is usually termed a local minimum. The word *local* is used to signify that this point is not guaranteed—and, in fact, is not usually—a global minimum. This method accepts any legal improving move and so is not too expensive in terms of computational cost. Thus, it is preferred when the problem is very large or an optimized solution is required as quickly as possible.

Dispatcher also offers the best accept search heuristic, which works like the first accept except that best accept search takes the legal move from the neighborhood that decreases cost by the greatest amount. Dispatcher also offers metaheuristics, such as guided local search and tabu search, that allow neighborhood moves that degrade the current solution to escape current local minimums. For more information, see Appendix C, *Predefined Search Heuristics and Metaheuristics*.

Step 8 Improve the solution

Add the following code after the comment `//Improve the solution`

```
IloGoal improve = IloSingleMove(_env,
                                _solution,
                                nhood,
                                IloImprove(_env),
                                _instantiateCost);
while (_solver.solve(improve)) {
}
```

You use the function `IloSingleMove` to return a goal that makes a single local move as defined by a neighborhood and a search heuristic. The first legal move reducing costs results. The goal scans the neighborhood `nhood` using `_solution` as the current solution. The function `IloSingleMove` will first see if it can make a legal, cost-decreasing move using the neighborhood `IloTwoOpt`. If it cannot, it tries `IloOrOpt`, and so on. The search heuristic `IloImprove` is used to filter moves by implementing a greedy search heuristic. It accepts only new routing plans that strictly decrease the cost and, thus, allows only improvements in the objective. The subgoal `_instantiateCost` is executed after the deltas from the neighborhood are applied to the current solution.

If a successful move can be found, the goal succeeds. The constrained variables are in the state corresponding to the application of the move. This new state is saved to `_solution` before the goal succeeds. If no successful move can be found, the goal fails, and `_solution` is left unchanged. You then use the `_restoreSolution` goal you created in the `RoutingSolver` constructor to restore this last solution.

Step 9 Restore the last solution

Add the following code after the comment `//Restore the last solution`

```
    _solver.solve(_restoreSolution);
}
```

Define the printInformation function

The `printInformation` function displays the routing plans found during the first solution generation and solution improvement phases and their costs. This function also displays information about the search, including number of variables, number of constraints, elapsed time since creation of search, number of fails, and number of choice points. The number of choice points is linked to the number of explored alternatives in the search tree. The number of fails is the number of incorrect decisions at choice points.

This code is provided for you:

```
void RoutingSolver::printInformation(const char* heading) const {
  if(heading)
      _solver.out()<<heading<<endl;
  _solver.printInformation();
  _dispatcher.printInformation();
  _solver.out() << "==============" << endl
    << "Cost          : " << _dispatcher.getTotalCost() << endl
    << "Number of vehicles used : "
    << _dispatcher.getNumberOfVehiclesUsed() << endl
    << "Solution      : " << endl
    << _dispatcher << endl;
}
```

Define the main function

After you finish creating the `RoutingModel` and `RoutingSolver` classes and the `printInformation` function, you use them in the `main` function. You can use command line syntax to pass the names of input files to the model. If you do not specify input files, the defaults will be used. In the `main` function, you first create an environment. Then you create an instance of the `RoutingModel` class, which takes the environment and input files as parameters. You create an instance of the `RoutingSolver` class. This takes one parameter, the model. You use an `if` loop to find a solution. If Solver finds a first solution, you print the information and then improve the solution. When Solver finds a solution that cannot be improved any further by the first accept search heuristic, Solver prints that solution. The following code is provided for you:

```
int main(int argc, char * argv[]) {
  IloEnv env;
  try {
    RoutingModel mdl(env, argc, argv);
    RoutingSolver solver(mdl);
    if (solver.findFirstSolution()) {
      solver.printInformation("***First Solution***");
      solver.improveWithNhood();
      solver.printInformation("***Improved Solution***");
    }
  } catch(IloException& ex) {
    cerr << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Step 10 Compile and run the program

Compile and run the program. You will get results that show the routing plan and information for both the first solution and the improved solution. The first solution has a cost of 417.834 units. The improved solution has a cost of 366.447 units. During the improvement phase, Dispatcher made 11 moves and lowered the cost of the routing plan by 51.387 units. Both routing plans use four vehicles.

First Solution Information

The first solution phase finds a solution with a total cost of 417.834 units:

```
***First Solution***
Number of fails              : 76
Number of choice points      : 1051
Number of variables          : 869
Number of constraints        : 153
Reversible stack (bytes)     : 92484
Solver heap (bytes)          : 478660
Solver global heap (bytes)   : 50860
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11160
Total memory used (bytes)    : 701576
Elapsed time since creation  : 0.06
Number of nodes              : 21
Number of visits             : 30
Number of vehicles           : 5
Number of dimensions         : 3
Number of accepted moves     : 0
===============
Cost         : 417.834
Number of vehicles used : 4
```

Improved Solution Information

The solution improvement phase finds a solution with a total cost of 366.447 units after making 11 cost-decreasing moves:

```
Improving solution
***Improved Solution***
Number of fails              : 0
Number of choice points      : 0
Number of variables          : 869
Number of constraints        : 149
Reversible stack (bytes)     : 92484
Solver heap (bytes)          : 478660
Solver global heap (bytes)   : 62920
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11160
Total memory used (bytes)    : 713636
Elapsed time since creation  : 0
Number of nodes              : 21
Number of visits             : 30
Number of vehicles           : 5
Number of dimensions         : 3
Number of accepted moves     : 11
===============
Cost         : 366.447
Number of vehicles used : 4
```

First Solution Routing Plan

During the first solution phase, Dispatcher found the following routing plan using four vehicles:

```
Solution     :
Unperformed visits : None
vehicle1 :
 -> depot weight[0] time[0..2.76114] distance[0..Inf) -> visit18 weight[0..87]
time[15.8114..18.5725] distance[15.8114..Inf) -> visit7 weight[12..99]
time[35.8114..38.5725] distance[25.8114..Inf) -> visit19 weight[17..104]
time[56.9917..59.7529] distance[36.9917..Inf) -> visit11 weight[34..121]
time[74.0628..76.8239] distance[44.0628..Inf) -> visit10 weight[46..133]
time[95.2431..98.0043] distance[55.2431..Inf) -> visit20 weight[62..149]
time[121.055..123.816] distance[71.0545..Inf) -> visit3 weight[71..158]
time[153.415..156.176] distance[93.4152..Inf) -> visit12 weight[84..171]
time[174.596..177.357] distance[104.596..Inf) -> visit1 weight[103..190]
time[201.239..204] distance[121.239..Inf) -> depot weight[113..200]
time[226.47..230] distance[136.47..Inf)
vehicle2 :
 -> depot weight[0] time[0..49.4126] distance[0..Inf) -> visit6 weight[0..114]
time[11.1803..60.5929] distance[11.1803..Inf) -> visit5 weight[3..117]
time[31.1803..80.5929] distance[21.1803..Inf) -> visit8 weight[29..143]
time[95..104.521] distance[35.1087..Inf) -> visit17 weight[38..152]
time[118.928..128.45] distance[49.0371..Inf) -> visit16 weight[40..154]
time[140.109..149.63] distance[60.2175..Inf) -> visit14 weight[59..173]
time[161.289..170.81] distance[71.3978..Inf) -> visit2 weight[79..193]
time[192.479..202] distance[92.5874..Inf) -> depot weight[86..200]
time[220.479..230] distance[110.587..Inf)
vehicle3 :
 -> depot weight[0] time[0..40.5862] distance[0..Inf) -> visit15 weight[0..173]
time[61..71] distance[30.4138..Inf) -> visit4 weight[8..181] time[149..159]
distance[59.5686..Inf) -> depot weight[27..200] time[184..230]
distance[84.5686..Inf)
vehicle4 :
 -> depot weight[0] time[0..74.9844] distance[0..Inf) -> visit9 weight[0..161]
time[97..107] distance[32.0156..Inf) -> visit13 weight[16..177] time[159..169]
distance[75.0272..Inf) -> depot weight[39..200] time[180.18..230]
distance[86.2076..Inf)
vehicle5 : Unused
```

Here is how to interpret this program output:

`vehicle1 :`

`-> depot weight[0] time[0..2.76114] distance[0..Inf) -> visit18 weight[0..87] time[15.8114..18.5725] distance[15.8114..Inf)` indicates that the truck (`vehicle1`) leaves the depot at 0 time units. At this point, the truck has traveled 0 distance units and delivered 0 weight units. Next, the truck arrives at customer 18, not earlier than 15.8114 time units, and not later than 18.5725 time units. At this point, the truck has traveled at least 15.8114 distance units. The weight delivered at this point is still 0 weight units. Remember that weight is delivered *after* arrival.

Improved Solution Routing Plan

During the solution improvement phase, Dispatcher found the following routing plan:

```
Solution     :
Unperformed visits : None
vehicle1 :
 -> depot weight[0] time[0..25.8217] distance[0..Inf) -> visit10 weight[0..112]
time[25.4951..51.3168] distance[25.4951..Inf) -> visit11 weight[16..128]
time[67..72.4972] distance[36.6754..Inf) -> visit19 weight[28..140]
time[84.0711..89.5683] distance[43.7465..Inf) -> visit7 weight[45..157]
time[105.251..110.749] distance[54.9268..Inf) -> visit18 weight[50..162]
time[125.251..130.749] distance[64.9268..Inf) -> visit6 weight[62..174]
time[146.432..151.929] distance[76.1072..Inf) -> visit13 weight[65..177]
time[163.503..169] distance[83.1783..Inf) -> depot weight[88..200]
time[184.683..230] distance[94.3586..Inf)
vehicle2 :
 -> depot weight[0] time[0..60.4561] distance[0..Inf) -> visit5 weight[0..124]
time[20.6155..81.0716] distance[20.6155..Inf) -> visit8 weight[26..150]
time[95..105] distance[34.5439..Inf) -> visit17 weight[35..159]
time[118.928..144.639] distance[48.4723..Inf) -> visit16 weight[37..161]
time[140.109..165.82] distance[59.6526..Inf) -> visit14 weight[56..180]
time[161.289..187] distance[70.833..Inf) -> depot weight[76..200]
time[203.305..230] distance[102.849..Inf)
vehicle3 :
 -> depot weight[0] time[0..30] distance[0..Inf) -> visit2 weight[0..166]
time[18..48] distance[18..Inf) -> visit15 weight[7..173] time[61..71]
distance[31..Inf) -> visit4 weight[15..181] time[149..159]
distance[60.1548..Inf) -> depot weight[34..200] time[184..230]
distance[85.1548..Inf)
vehicle4 :
 -> depot weight[0] time[0..45.8197] distance[0..Inf) -> visit12 weight[0..133]
time[15..60.8197] distance[15..Inf) -> visit3 weight[19..152] time[36.1803..82]
distance[26.1803..Inf) -> visit9 weight[32..165] time[97..107]
distance[41.1803..Inf) -> visit20 weight[48..181] time[118.18..177.508]
distance[52.3607..Inf) -> visit1 weight[57..190] time[144.673..204]
distance[68.8531..Inf) -> depot weight[67..200] time[169.904..230]
distance[84.0846..Inf)
vehicle5 : Unused
```

As you can see, the improved solution still uses four vehicles, but with a lower cost. This solution also better balances the visits between the trucks. In the first solution routing plan, vehicle 1 makes nine visits, vehicle 2 makes six visits, and vehicles 3 and 4 each make two visits. In the improved solution routing plan, vehicle 1 makes seven visits, vehicles 2 and 4 each make five visits, and vehicle 3 makes three visits.

The complete program and output are listed in “Complete Program” on page 73. You can also view it online in the `YourDispatcherHome/examples/src/vrp.cpp` file.

Review Exercises

1. What are the two phases to finding a solution using Dispatcher?

2. What is a neighborhood?
3. Rewrite the example to sequentially generate first solutions using the following predefined first solution heuristics: `IloSavingsGenerate`, `IloSweepGenerate`, `IloNearestDepotGenerate`, `IloNearestAdditionGenerate`, and `IloInsertionGenerate`. Analyze the output to see how the different first solution heuristics change the first solution and its cost.

Suggested Answers

Exercise 1

What are the two phases to finding a solution using Dispatcher?

Suggested Answer

Dispatcher uses a two-phase method to solve problems. The first phase consists of *generating a first solution* that satisfies the problem. In the second phase, you improve this first solution using *local search*.

Exercise 2

What is a neighborhood?

Suggested Answer

The central idea of a neighborhood is to define a set of solution changes, or deltas, that represent alternative moves that can be taken.

Exercise 3

Rewrite the example to sequentially generate first solutions using the following predefined first solution heuristics: `IloSavingsGenerate`, `IloSweepGenerate`, `IloNearestDepotGenerate`, `IloNearestAdditionGenerate`, and `IloInsertionGenerate`. Analyze the output to see how the different first solution heuristics change the first solution and its cost.

Suggested Answer

To sequentially generate first solutions using different heuristics, create a function `IloRoutingSolver::sequentialGenerate`. This function is called in the `main` function using `solver.sequentialGenerate()`:

```
void RoutingSolver::sequentialGenerate() {
  generate(IloSavingsGenerate(_env) && _instantiateCost,
           (char *)"Savings");
  generate(IloSweepGenerate(_env) && _instantiateCost,
           (char *)"Sweep");
  generate(IloNearestDepotGenerate(_env) && _instantiateCost,
           (char *)"Nearest to Depot");
  generate(IloNearestAdditionGenerate(_env) && _instantiateCost,
           (char *)"Nearest addition");
  generate(IloInsertionGenerate(_env, _instantiateCost),
           (char *)"Insertion");
}
```

Different first solution heuristics will work differently depending on the particulars of your problem. In this example, using the sweep heuristic found the lowest cost solution, 270.403 units. The savings heuristic found the next lowest cost solution, 280.947 units, followed by nearest-addition, insertion, and nearest-to-depot. The solution found using the nearest-to-depot heuristic had a cost of 369.297 units. From this example, you can see that trying different first solution heuristics can greatly change the cost of your final solution. For more information on choosing the right first solution heuristic for you problem, see “Using the predefined first solution heuristics” on page 407.

You can view this complete program and output online in the `YourDispatcherHome/examples/src/generate.cpp` file.

The following output is generated for the savings heuristic:

```
Savings
Number of fails              : 38
Number of choice points      : 1052
Number of variables          : 390
Number of constraints        : 98
Reversible stack (bytes)     : 60324
Solver heap (bytes)          : 337960
Solver global heap (bytes)   : 42744
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11160
Total memory used (bytes)    : 520600
Elapsed time since creation  : 0.047
Number of nodes              : 12
Number of visits             : 21
Number of vehicles           : 5
Number of dimensions         : 2
Number of accepted moves     : 0
===============
Cost         : 280.947
Number of vehicles used : 4
Solution     :
Unperformed visits : None
truck1 :
 -> depot weight[0..6] distance[0..Inf) -> visit5 weight[0..6]
distance[14.1421..Inf) -> visit10 weight[21..27] distance[28.3548..Inf) ->
visit9 weight[26..32] distance[40.3964..Inf) -> visit1 weight[37..43]
distance[64.6038..Inf) -> depot weight[44..50] distance[78.4963..Inf)
truck2 :
 -> depot weight[0..4] distance[0..Inf) -> visit3 weight[0..4]
distance[32.5576..Inf) -> visit2 weight[16..20] distance[47.8547..Inf) -> depot
weight[46..50] distance[68.8785..Inf)
truck3 :
 -> depot weight[0..8] distance[0..Inf) -> visit8 weight[0..8]
distance[22.0227..Inf) -> visit7 weight[23..31] distance[36.0584..Inf) -> depot
weight[42..50] distance[62.4781..Inf)
truck4 :
 -> depot weight[0..7] distance[0..Inf) -> visit6 weight[0..7]
distance[11.4018..Inf) -> visit4 weight[15..22] distance[32.4256..Inf) ->
visit11 weight[24..31] distance[59.0526..Inf) -> depot weight[43..50]
distance[71.0942..Inf)
truck5 : Unused
```

The following output is generated for the sweep heuristic:

```
Sweep
Number of fails              : 14
Number of choice points      : 1056
Number of variables          : 390
Number of constraints        : 93
Reversible stack (bytes)     : 60324
Solver heap (bytes)          : 337960
Solver global heap (bytes)   : 42744
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11160
Total memory used (bytes)    : 520600
Elapsed time since creation  : 0
Number of nodes              : 12
Number of visits             : 21
Number of vehicles           : 5
Number of dimensions         : 2
Number of accepted moves     : 0
===============
Cost         : 270.403
Number of vehicles used : 4
Solution     :
Unperformed visits : None
truck1 :
 -> depot weight[0..4] distance[0..Inf) -> visit4 weight[0..4]
distance[17.2047..Inf) -> visit5 weight[9..13] distance[37.6007..Inf) ->
visit10 weight[30..34] distance[51.8134..Inf) -> visit9 weight[35..39]
distance[63.855..Inf) -> depot weight[46..50] distance[86.9418..Inf)
truck2 :
 -> depot weight[0..1] distance[0..Inf) -> visit11 weight[0..1]
distance[12.0416..Inf) -> visit2 weight[19..20] distance[22.6717..Inf) -> depot
weight[49..50] distance[43.6955..Inf)
truck3 :
 -> depot weight[0..4] distance[0..Inf) -> visit3 weight[0..4]
distance[32.5576..Inf) -> visit1 weight[16..20] distance[51.767..Inf) -> visit8
weight[23..27] distance[63.4289..Inf) -> depot weight[46..50]
distance[85.4516..Inf)
truck4 :
 -> depot weight[0..16] distance[0..Inf) -> visit7 weight[0..16]
distance[26.4197..Inf) -> visit6 weight[19..35] distance[42.9121..Inf) -> depot
weight[34..50] distance[54.3139..Inf)
truck5 : Unused
```

The following output is generated for the nearest-to-depot heuristic:

```
Nearest to Depot
Number of fails              : 13
Number of choice points      : 1055
Number of variables          : 390
Number of constraints        : 93
Reversible stack (bytes)     : 60324
Solver heap (bytes)          : 337960
Solver global heap (bytes)   : 42744
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11160
Total memory used (bytes)    : 520600
Elapsed time since creation  : 0
Number of nodes              : 12
Number of visits             : 21
Number of vehicles           : 5
Number of dimensions         : 2
Number of accepted moves     : 0
===============
Cost         : 369.297
Number of vehicles used : 4
Solution     :
Unperformed visits : None
truck1 :
 -> depot weight[0] distance[0..Inf) -> visit6 weight[0] distance[11.4018..Inf)
-> visit11 weight[15] distance[33.2421..Inf) -> visit1 weight[34]
distance[45.3251..Inf) -> visit4 weight[41] distance[76.3896..Inf) -> depot
weight[50] distance[93.5942..Inf)
truck2 :
 -> depot weight[0..1] distance[0..Inf) -> visit5 weight[0..1]
distance[14.1421..Inf) -> visit8 weight[21..22] distance[47.3837..Inf) ->
visit10 weight[44..45] distance[93.0017..Inf) -> depot weight[49..50]
distance[121.321..Inf)
truck3 :
 -> depot weight[0..9] distance[0..Inf) -> visit2 weight[0..9]
distance[21.0238..Inf) -> visit9 weight[30..39] distance[37.3026..Inf) -> depot
weight[41..50] distance[60.3894..Inf)
truck4 :
 -> depot weight[0..15] distance[0..Inf) -> visit7 weight[0..15]
distance[26.4197..Inf) -> visit3 weight[19..34] distance[61.434..Inf) -> depot
weight[35..50] distance[93.9916..Inf)
truck5 : Unused
```

The following output is generated for the nearest-addition heuristic:

```
Nearest addition
Number of fails              : 21
Number of choice points      : 1052
Number of variables          : 390
Number of constraints        : 98
Reversible stack (bytes)     : 60324
Solver heap (bytes)          : 341980
Solver global heap (bytes)   : 42744
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11160
Total memory used (bytes)    : 524620
Elapsed time since creation  : 0.015
Number of nodes              : 12
Number of visits             : 21
Number of vehicles           : 5
Number of dimensions         : 2
Number of accepted moves     : 0
===============
Cost         : 285.232
Number of vehicles used : 4
Solution     :
Unperformed visits : None
truck1 :
 -> depot weight[0] distance[0..Inf) -> visit6 weight[0] distance[11.4018..Inf)
-> visit7 weight[15] distance[27.8942..Inf) -> visit1 weight[34]
distance[50.7196..Inf) -> visit4 weight[41] distance[81.7841..Inf) -> depot
weight[50] distance[98.9887..Inf)
truck2 :
 -> depot weight[0..1] distance[0..Inf) -> visit11 weight[0..1]
distance[12.0416..Inf) -> visit2 weight[19..20] distance[22.6717..Inf) -> depot
weight[49..50] distance[43.6955..Inf)
truck3 :
 -> depot weight[0..13] distance[0..Inf) -> visit5 weight[0..13]
distance[14.1421..Inf) -> visit9 weight[21..34] distance[26.5115..Inf) ->
visit10 weight[32..45] distance[38.553..Inf) -> depot weight[37..50]
distance[66.8727..Inf)
truck4 :
 -> depot weight[0..11] distance[0..Inf) -> visit8 weight[0..11]
distance[22.0227..Inf) -> visit3 weight[23..34] distance[43.1177..Inf) -> depot
weight[39..50] distance[75.6754..Inf)
truck5 : Unused
```

The following output is generated for the insertion heuristic:

```
Insertion
Number of fails              : 11
Number of choice points      : 13680
Number of variables          : 390
Number of constraints        : 93
Reversible stack (bytes)     : 60324
Solver heap (bytes)          : 341980
Solver global heap (bytes)   : 473476
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11160
Total memory used (bytes)    : 955352
Elapsed time since creation  : 0.078
Number of nodes              : 12
Number of visits             : 21
Number of vehicles           : 5
Number of dimensions         : 2
Number of accepted moves     : 0
===============
Cost         : 320.843
Number of vehicles used : 4
Solution     :
Unperformed visits : None
truck1 :
 -> depot weight[0..4] distance[0..Inf) -> visit4 weight[0..4]
distance[17.2047..Inf) -> visit2 weight[9..13] distance[54.2182..Inf) -> visit1
weight[39..43] distance[66.5875..Inf) -> depot weight[46..50]
distance[80.4799..Inf)
truck2 :
 -> depot weight[0..2] distance[0..Inf) -> visit5 weight[0..2]
distance[14.1421..Inf) -> visit9 weight[21..23] distance[26.5115..Inf) ->
visit3 weight[32..34] distance[57.5115..Inf) -> depot weight[48..50]
distance[90.0691..Inf)
truck3 :
 -> depot weight[0..16] distance[0..Inf) -> visit7 weight[0..16]
distance[26.4197..Inf) -> visit6 weight[19..35] distance[42.9121..Inf) -> depot
weight[34..50] distance[54.3139..Inf)
truck4 :
 -> depot weight[0..3] distance[0..Inf) -> visit10 weight[0..3]
distance[28.3196..Inf) -> visit11 weight[5..8] distance[50.2513..Inf) -> visit8
weight[24..27] distance[73.9579..Inf) -> depot weight[47..50]
distance[95.9806..Inf)
truck5 : Unused
```

Complete Program

The complete VRP program follows. You can also view it online in the `YourDispatcherHome/examples/src/vrp.cpp` file.

```
// --------------------------------------------------------------- -*- C++ -*-
// File: examples/src/vrp.cpp
// --------------------------------------------------------------------------


#include <ildispat/ilodispatcher.h>

ILOSTLBEGIN
////////////////////////////////////////////////////////////////////////////////
// Modeling

class RoutingModel {
  IloEnv             _env;
  IloModel           _mdl;
  IloDimension2      _time;
  IloDimension2      _distance;
  IloDimension1      _weight;

  void addDimensions();
  void createIloNodes(const char* nodeFileName);
  void createVehicles(const char* vehicleFileName);
  void createVisits(const char* visitsFileName);

public:
  RoutingModel(IloEnv env, int argc, char* argv[]);
  ~RoutingModel() {}
  IloEnv    getEnv() const { return _env; }
  IloModel  getModel() const { return _mdl; }
};

RoutingModel::RoutingModel( IloEnv env,
                            int argc,
                            char* argv[]):
  _env(env), _mdl(env) {
  addDimensions();

  //create IloNodes
  char * nodeFileName;
  if(argc < 2)  nodeFileName =
       (char*) “../../../examples/data/vrp20/vrp20nodes.csv”;
  else          nodeFileName = argv[1];
  createIloNodes(nodeFileName);

  //create vehicles
  char * vehiclesFileName;
  if(argc < 3)  vehiclesFileName =
       (char*) “../../../examples/data/vrp20/vrp20vehicles.csv”;
  else          vehiclesFileName = argv[2];
  createVehicles(vehiclesFileName);

  //create visits
  char * visitsFileName;
  if(argc < 4)  visitsFileName =
       (char*)  “../../../examples/data/vrp20/vrp20visits.csv”;
  else          visitsFileName = argv[3];
  createVisits(visitsFileName);
}

// create distance functions for dimensions, add dimensions to model
```

```
void RoutingModel::addDimensions() {

  _weight =IloDimension1 (_env, “weight”);
  _mdl.add(_weight);

  _time  =IloDimension2 (_env, IloEuclidean, “time”);
  _mdl.add(_time);

  _distance =IloDimension2 (_env, IloEuclidean, “distance”);
  _mdl.add(_distance);
}

//create IloNodes
void RoutingModel::createIloNodes(const char* nodeFileName) {
  IloCsvReader csvNodeReader(_env, nodeFileName);

  IloCsvReader::LineIterator  it(csvNodeReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char * name = line.getStringByHeader(“name”);

    IloNode node(_env, line.getFloatByHeader(“x”),
        line.getFloatByHeader(“y”), 0, name);

    node.setKey(name);

    ++it;
  }

  csvNodeReader.end();
}

//create vehicles
void RoutingModel::createVehicles(const char* vehicleFileName) {
  IloCsvReader csvVehicleReader(_env, vehicleFileName);
  IloCsvReader::LineIterator  it(csvVehicleReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char * namefirst = line.getStringByHeader(“first”);
    char * namelast = line.getStringByHeader(“last”);
    char * name = line.getStringByHeader(“name”);
    IloNum capacity = line.getFloatByHeader(“capacity”);

    IloNode node1 = IloNode::Find(_env, namefirst);
    IloNode node2 = IloNode::Find(_env, namelast);

    IloVisit first(node1, “depot”);

    _mdl.add(first.getCumulVar(_weight) == 0);
    _mdl.add(first.getCumulVar(_time) >= line.getFloatByHeader(“open”));

    IloVisit last(node2, “depot”);
    _mdl.add(last.getCumulVar(_time) <= line.getFloatByHeader(“close”));

    IloVehicle vehicle(first, last, name);

    vehicle.setCost(_distance, 1.0);
    vehicle.setCapacity(_weight, capacity);

    _mdl.add(vehicle);

    ++it;
  }
  csvVehicleReader.end();
}

//create visits
void RoutingModel::createVisits(const char* visitsFileName) {
  IloCsvReader csvVisitReader(_env, visitsFileName);
  IloCsvReader::LineIterator  it(csvVisitReader);
  while(it.ok()){
    IloCsvLine line = *it;
    //read visit data from files
    char * visitName =  line.getStringByHeader(“name”);
    char * nodeName = line.getStringByHeader(“node”);
    IloNum quantity = line.getFloatByHeader(“quantity”);
    IloNum minTime  = line.getFloatByHeader(“minTime”);
    IloNum maxTime  = line.getFloatByHeader(“maxTime”);
    IloNum dropTime = line.getFloatByHeader(“dropTime”);

    IloNode node = IloNode::Find(_env, nodeName);

    IloVisit visit(node, visitName);
    _mdl.add(visit.getDelayVar(_time) == dropTime);
    _mdl.add(visit.getTransitVar(_weight) == quantity);
    _mdl.add(minTime <= visit.getCumulVar(_time) <= maxTime);

    _mdl.add(visit);

    ++it;
  }
  csvVisitReader.end();
}

///////////////////////////////////////////////////////////////////////////////
// Solving
class RoutingSolver {
  IloEnv              _env;
  IloModel            _mdl;
  IloSolver           _solver;
  IloDispatcher       _dispatcher;
  IloRoutingSolution  _solution;
  IloGoal             _instantiateCost;
  IloGoal             _restoreSolution;
  IloGoal             _goal;

public:
  RoutingSolver(RoutingModel mdl);
  ~RoutingSolver() {}
  IloBool findFirstSolution();
  void improveWithNhood();
  void printInformation(const char* =0) const;
};

RoutingSolver::RoutingSolver(RoutingModel mdl):
  _env(mdl.getEnv()),
  _mdl(mdl.getModel()),
  _solver(mdl.getModel()),
  _dispatcher(_solver),
  _solution(mdl.getModel()){

    _instantiateCost =
          IloDichotomize(_env, _dispatcher.getCostVar(), IloFalse);

    _restoreSolution = IloRestoreSolution(_env, _solution);

    _goal = IloSavingsGenerate(_env) && _instantiateCost;
  }

// Solving : find first solution
IloBool RoutingSolver::findFirstSolution() {
  if (!_solver.solve(_goal)) {
    _solver.error() << “Infeasible Routing Plan” << endl;
    return IloFalse;
  }

  _solution.store(_solver);
  return IloTrue;
}

//Improve solution using nhood
void RoutingSolver::improveWithNhood() {
  IloNHood nhood =  IloTwoOpt(_env)
                  + IloOrOpt(_env)
                  + IloRelocate(_env)
                  + IloCross(_env)
                  + IloExchange(_env);
  _solver.out() << “Improving solution” << endl;

  IloGoal improve = IloSingleMove(_env,
                                  _solution,
                                  nhood,
                                  IloImprove(_env),
                                  _instantiateCost);
  while (_solver.solve(improve)) {
  }

   _solver.solve(_restoreSolution);
}

// Display Dispatcher information
void RoutingSolver::printInformation(const char* heading) const {
  if(heading)
      _solver.out()<<heading<<endl;
  _solver.printInformation();
  _dispatcher.printInformation();
  _solver.out() << “===============” << endl
    << “Cost         : “ << _dispatcher.getTotalCost() << endl
    << “Number of vehicles used : “
    << _dispatcher.getNumberOfVehiclesUsed() << endl
    << “Solution     : “ << endl
    << _dispatcher << endl;
}
```

```
////////////////////////////////////////////////////////////////////////////////
int main(int argc, char * argv[]) {
  IloEnv env;
  try {
    RoutingModel mdl(env, argc, argv);
    RoutingSolver solver(mdl);
    if (solver.findFirstSolution()) {
      solver.printInformation("***First Solution***");
      solver.improveWithNhood();
      solver.printInformation("***Improved Solution***");
    }
  } catch(IloException& ex) {
    cerr << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Complete Output

```
/**
***First Solution***
Number of fails              : 76
Number of choice points      : 1051
Number of variables          : 869
Number of constraints        : 153
Reversible stack (bytes)     : 92484
Solver heap (bytes)          : 478660
Solver global heap (bytes)   : 50860
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11160
Total memory used (bytes)    : 701576
Elapsed time since creation  : 0.06
Number of nodes              : 21
Number of visits             : 30
Number of vehicles           : 5
Number of dimensions         : 3
Number of accepted moves     : 0
===============
Cost         : 417.834
Number of vehicles used : 4
Solution     :
Unperformed visits : None
vehicle1 :
 -> depot weight[0] time[0..2.76114] distance[0..Inf) -> visit18 weight[0..87]
time[15.8114..18.5725] distance[15.8114..Inf) -> visit7 weight[12..99]
time[35.8114..38.5725] distance[25.8114..Inf) -> visit19 weight[17..104]
time[56.9917..59.7529] distance[36.9917..Inf) -> visit11 weight[34..121]
time[74.0628..76.8239] distance[44.0628..Inf) -> visit10 weight[46..133]
time[95.2431..98.0043] distance[55.2431..Inf) -> visit20 weight[62..149]
time[121.055..123.816] distance[71.0545..Inf) -> visit3 weight[71..158]
time[153.415..156.176] distance[93.4152..Inf) -> visit12 weight[84..171]
```

```
time[174.596..177.357] distance[104.596..Inf) -> visit1 weight[103..190]
time[201.239..204] distance[121.239..Inf) -> depot weight[113..200]
time[226.47..230] distance[136.47..Inf)
vehicle2 :
 -> depot weight[0] time[0..49.4126] distance[0..Inf) -> visit6 weight[0..114]
time[11.1803..60.5929] distance[11.1803..Inf) -> visit5 weight[3..117]
time[31.1803..80.5929] distance[21.1803..Inf) -> visit8 weight[29..143]
time[95..104.521] distance[35.1087..Inf) -> visit17 weight[38..152]
time[118.928..128.45] distance[49.0371..Inf) -> visit16 weight[40..154]
time[140.109..149.63] distance[60.2175..Inf) -> visit14 weight[59..173]
time[161.289..170.81] distance[71.3978..Inf) -> visit2 weight[79..193]
time[192.479..202] distance[92.5874..Inf) -> depot weight[86..200]
time[220.479..230] distance[110.587..Inf)
vehicle3 :
 -> depot weight[0] time[0..40.5862] distance[0..Inf) -> visit15 weight[0..173]
time[61..71] distance[30.4138..Inf) -> visit4 weight[8..181] time[149..159]
distance[59.5686..Inf) -> depot weight[27..200] time[184..230]
distance[84.5686..Inf)
vehicle4 :
 -> depot weight[0] time[0..74.9844] distance[0..Inf) -> visit9 weight[0..161]
time[97..107] distance[32.0156..Inf) -> visit13 weight[16..177] time[159..169]
distance[75.0272..Inf) -> depot weight[39..200] time[180.18..230]
distance[86.2076..Inf)
vehicle5 : Unused
Improving solution
***Improved Solution***
Number of fails               : 0
Number of choice points       : 0
Number of variables           : 869
Number of constraints         : 149
Reversible stack (bytes)      : 92484
Solver heap (bytes)           : 478660
Solver global heap (bytes)    : 62920
And stack (bytes)             : 20124
Or stack (bytes)              : 44244
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 11160
Total memory used (bytes)     : 713636
Elapsed time since creation   : 0
Number of nodes               : 21
Number of visits              : 30
Number of vehicles            : 5
Number of dimensions          : 3
Number of accepted moves      : 11
===============
Cost         : 366.447
Number of vehicles used : 4
Solution     :
Unperformed visits : None
vehicle1 :
 -> depot weight[0] time[0..25.8217] distance[0..Inf) -> visit10 weight[0..112]
time[25.4951..51.3168] distance[25.4951..Inf) -> visit11 weight[16..128]
time[67..72.4972] distance[36.6754..Inf) -> visit19 weight[28..140]
time[84.0711..89.5683] distance[43.7465..Inf) -> visit7 weight[45..157]
time[105.251..110.749] distance[54.9268..Inf) -> visit18 weight[50..162]
time[125.251..130.749] distance[64.9268..Inf) -> visit6 weight[62..174]
time[146.432..151.929] distance[76.1072..Inf) -> visit13 weight[65..177]
time[163.503..169] distance[83.1783..Inf) -> depot weight[88..200]
```

```
time[184.683..230] distance[94.3586..Inf)
vehicle2 :
 -> depot weight[0] time[0..60.4561] distance[0..Inf) -> visit5 weight[0..124]
time[20.6155..81.0716] distance[20.6155..Inf) -> visit8 weight[26..150]
time[95..105] distance[34.5439..Inf) -> visit17 weight[35..159]
time[118.928..144.639] distance[48.4723..Inf) -> visit16 weight[37..161]
time[140.109..165.82] distance[59.6526..Inf) -> visit14 weight[56..180]
time[161.289..187] distance[70.833..Inf) -> depot weight[76..200]
time[203.305..230] distance[102.849..Inf)
vehicle3 :
 -> depot weight[0] time[0..30] distance[0..Inf) -> visit2 weight[0..166]
time[18..48] distance[18..Inf) -> visit15 weight[7..173] time[61..71]
distance[31..Inf) -> visit4 weight[15..181] time[149..159]
distance[60.1548..Inf) -> depot weight[34..200] time[184..230]
distance[85.1548..Inf)
vehicle4 :
 -> depot weight[0] time[0..45.8197] distance[0..Inf) -> visit12 weight[0..133]
time[15..60.8197] distance[15..Inf) -> visit3 weight[19..152] time[36.1803..82]
distance[26.1803..Inf) -> visit9 weight[32..165] time[97..107]
distance[41.1803..Inf) -> visit20 weight[48..181] time[118.18..177.508]
distance[52.3607..Inf) -> visit1 weight[57..190] time[144.673..204]
distance[68.8531..Inf) -> depot weight[67..200] time[169.904..230]
distance[84.0846..Inf)
vehicle5 : Unused
Press any key to continue
*/
```

CHAPTER

4

Minimizing the Number of Vehicles

In this lesson, you will learn how to:

- minimize the number of vehicles in a routing problem
- create a road network graph and compute shortest paths
- use the classes and functions `IloDispatcherGraph`, `IloGraphDistance`, `IloMakePerformed`, `IloMakeUnperformed`, `IloSwapPerform`, `IloRoutingSolution::RouteIterator`, `IloRoutingSolution::getRouteSize`, and `IloVisit::setPenaltyCost`

You will learn how to solve a vehicle routing problem (VRP) where the main objective is to reduce the number of vehicles. Dispatcher’s graph functionality is also introduced in this lesson. You will learn how to create a graph representing a road network and to compute the shortest paths between visits from this graph. The graph functionality can be used to compute distance and time dimensions in any routing problem. Conversely, you could also use Euclidean, Manhattan, or another type of distance computation for the problem in this lesson.

Describe

This problem is the same as the problem presented in Chapter 2, *Modeling a Vehicle Routing Problem* except that the objective is to reduce the number of vehicles before minimizing distance or time.

Step 1 Describe the problem

The first step is to write a natural language description of the problem.

The components of the routing model for this problem are the same as for a standard VRP: vehicles, customers, and a depot.

The constraints in this problem are the same as those in a standard VRP: time windows, vehicle capacity, and visit quantities.

The objective is to minimize the number of vehicles used as well as minimizing the total length of the solution.

Model

Once you have written a description of your problem, you can use Dispatcher classes to model it.

Step 2 Open the example file

Open the example file `YourDispatcherHome/examples/src/tutorial/minvehcl_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 of this lesson, you will have completed the code for the problem and you will be able to compile and run it.

As in Chapter 2, *Modeling a Vehicle Routing Problem*, you create a `RoutingModel` class, which is used to model the problem.

Declare the RoutingModel class

The code for the declaration of the class `RoutingModel` is provided for you:

```
class RoutingModel {
  IloEnv             _env;
  IloModel           _mdl;
  IloDispatcherGraph  _graph;
  IloDimension2      _time;
  IloDimension1      _weight;

  void addDimensions();
  void loadGraphInformation (char* arcFileName, char* turnFileName);
  void lastMinuteGraphChanges ();
  void createIloNodes(char* nodeFileName, char* coordFileName);
  void createVehicles(char* vehicleFileName);
  void createVisits(char* visitsFileName);

public:
  RoutingModel(IloEnv env, int argc, char* argv[]);
  ~RoutingModel() {}
  IloEnv    getEnv() const { return _env; }
  IloModel  getModel() const { return _mdl; }
};
```

There are only a few differences between the `RoutingModel` class used in a standard VRP and the `RoutingModel` class used in this problem.

The dimensions in this problem are time and weight. Unlike in Chapter 2, *Modeling a Vehicle Routing Problem*, distance is not a dimension. In this example, only the dimensions representing time and weight are required to set side constraints and to compute the cost. However, you could add a distance dimension if you wanted to include this dimension in the model.

The graph, an instance of the class `IloDispatcherGraph`, allows you create a graph representing a road network topology on which instances of `IloNode` can be positioned. This graph is composed of graph nodes—which are different from `IloNodes`—and arcs connecting these graph nodes. The cost in time and distance for traversing an arc is loaded from a file, as well as any penalties associated with making turns. See “Define the RoutingModel constructor” on page 84.

`IloDispatcherGraph` computes the shortest paths between nodes for each vehicle. It is this shortest path that is used to create the dimension `_time` in this problem. See “Define the addDimensions function” on page 87.

You must associate the graph nodes with instances of `IloNode` that represent depot and visit locations. See “Define the createIloNodes function” on page 87.

The class `RoutingModel` introduces two member functions related to `IloDispatcherGraph` functionality: `loadGraphInformation` and `lastMinuteGraphChanges`. See “Define the loadGraphInformation function” on page 85 and “Define the lastMinuteGraphChanges function” on page 86.

Define the RoutingModel constructor

The constructor is defined as in Chapter 2, *Modeling a Vehicle Routing Problem*, except that it also allows you to specify the names of the input files relating to the road network graph. If you do not specify input files, the defaults will be used. In addition to files with node, vehicle, and visit data, as in Chapter 2, *Modeling a Vehicle Routing Problem,* this constructor loads information from an arc file, a turn file, and a node coordinates file. The arc file contains the arc name, the “from” graph node, the “to” graph node, the time to traverse the arc, and the distance between the two graph nodes that define the arc. The turn file contains two arcs and the time penalty associated with the turn between the two arcs. For example, a left turn may take longer than a right turn at certain intersections. By default, all turns are permitted with no penalty. The node coordinates file contains the name of a graph node and its x and y coordinates. It is used to associate a graph node and an `IloNode` representing a visit location.

This constructor will be called from the `main` function. It calls the following functions: `addDimensions`, `loadGraphInformation`, `createIloNodes`, `createVehicles`, and `createVisits`.

The following code is provided for you:

```
RoutingModel::RoutingModel( IloEnv env,
                            int argc,
                            char* argv[]):
  _env(env), _mdl(env), _graph(env) {
  addDimensions();

  // Load dispatcher graph information from file, and
  // add instance-specific features.
  char * arcFileName;
  if(argc < 2)  arcFileName =
     (char*) "../../../examples/data/dispatcherGraphData/gridNetwork.csv";
  else          arcFileName = argv[1];
  char * turnFileName;
  if(argc < 3)  turnFileName =
     (char*) "../../../examples/data/dispatcherGraphData/turnData.csv";
  else          turnFileName = argv[2];
  loadGraphInformation (arcFileName, turnFileName);

  // Create IloNodes and associate them to graph nodes
  char * nodeFileName;
  if(argc < 4)  nodeFileName =
     (char*) "../../../examples/data/vrp200/vrp200nodes.csv";
  else          nodeFileName = argv[3];
  char * nodeCoordsFile;
  if(argc < 5)  nodeCoordsFile =
     (char*) "../../../examples/data/dispatcherGraphData/coordTable.csv";
  else          nodeCoordsFile = argv[4];
  createIloNodes(nodeFileName, nodeCoordsFile);

  // Create vehicles
  char * vehiclesFileName;
  if(argc < 6)  vehiclesFileName =
     (char*) "../../../examples/data/vrp200/vrp200vehicles.csv";
  else          vehiclesFileName = argv[5];
  createVehicles(vehiclesFileName);

  // Create visits
  char * visitsFileName;
  if(argc < 7)  visitsFileName =
     (char*) "../../../examples/data/vrp200/vrp200visits.csv";
  else          visitsFileName = argv[6];
  createVisits(visitsFileName);
}
```

Define the loadGraphInformation function

First, you build the road network graph using the function `loadGraphInformation`. This function takes the arc file and turn file as parameters. The member function `IloDispatcherGraph::createArcsFromFile` loads the network topology from a csv file and creates all necessary arcs and nodes. The member function `IloDispatcheGraph::loadArcDimensionDataFromFile` loads the arc cost information relating to the dimension `_time` from the arc file. The member function

IloDispatcherGraph::loadTurnDimensionDataFromFile loads turn penalty information from the turn file. By default, all turns are allowed with no penalty. The loadGraphInformation function calls the lastMinuteGraphChanges function to allow for direct manipulation of the road network graph.

Step 3 Load the graph information

Add the following code after the comment //Load the graph information

```
void RoutingModel::loadGraphInformation ( char* arcFileName,
                                          char* turnFileName)    {
  _graph.createArcsFromFile (arcFileName);
  _graph.loadArcDimensionDataFromFile (arcFileName, _time);
  _graph.loadTurnDimensionDataFromFile(turnFileName, _time);
  lastMinuteGraphChanges();
}
```

Define the lastMinuteGraphChanges function

A road network graph may need to be modified directly at the last minute due to traffic, accidents, or construction work. IloDispatcherGraph includes functionality to allow you to modify the network topology and costs by direct manipulation. The member function IloDispatcherGraph::forbidArcUse sets the cost of the arc to infinity. The member function IloDispatcherGraph::setTurnPenalty sets the penalty cost of turning from one arc to another along the dimension. In this example, the penalty for turning from arc 1323 to arc 1544 is 12 time units.

Step 4 Make last minute graph modifications

Add the following code after the comment
//Make last minute graph modifications

```
void RoutingModel::lastMinuteGraphChanges () {
  _graph.forbidArcUse(_graph.getArcByEnds(2785-1, 2785));
  _graph.forbidArcUse(_graph.getArcByEnds(2785+1, 2785));
  _graph.forbidArcUse(_graph.getArcByEnds(2785-56, 2785));
  _graph.setTurnPenalty(_graph.getArc(1323), _graph.getArc(1544), _time, 12);
}
```

Define the addDimensions function

The member function `addDimensions` is used to create the two dimensions, `_weight` and `_time`, and add them to the model. The weight dimension is created in the same way as in Chapter 2, *Modeling a Vehicle Routing Problem*. This code is provided for you:

```
void RoutingModel::addDimensions() {
  _weight =IloDimension1 (_env, "weight");
  _mdl.add(_weight);
```

The time is computed from the road network graph. The dimension `_time` is an instance of the subclass `IloDimension2`. The constructor for `IloDimension2` takes three parameters: the environment, a distance function, and a name used for debug and trace purposes. In this lesson, the distance function is `SP_time`. This is the shortest path in time computed from the road network graph. To create `SP_time`, you use the function `IloGraphDistance`, which returns an instance of `IloDistance` for which the distance—or time—between two nodes for a specified vehicle is the value of the shortest path between the two nodes using the specified vehicle.

Step 5 Create the time dimension and add to model

Add the following code after the comment
`//Create the time dimension and add to model`

```
  IloDistance SP_time =   IloGraphDistance (_graph);
  _time  =IloDimension2 (_env, SP_time, "time");
  _mdl.add(_time);
}
```

In this example, you could also use the predefined distance functions `IloEuclidean` or `IloManhattan` or define your own distance function.

Define the createIloNodes function

You create the nodes in the same way you did in Chapter 2, *Modeling a Vehicle Routing Problem*, except that you must associate the graph nodes to the instances of `IloNode`. You use the csv reader functionality to input node data and node coordinate data from csv files. You use the member function `IloDispatcherGraph::associatebyCoordsInFile` to look up the coordinates of a given `IloNode` in a csv file and automatically associate it to the graph node with matching coordinates.

Step 6 Associate the graph nodes to the IloNodes

Add the following code after the comment
`//Associate the graph nodes to the IloNode`

```
    _graph.associateByCoordsInFile (node, coordFileName);
```

Here is the complete code for defining the `createIloNodes` function:

```
void RoutingModel::createIloNodes(char* nodeFileName, char* coordFileName) {
  IloCsvReader csvNodeReader(_env, nodeFileName);
  IloCsvReader::LineIterator  it(csvNodeReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char * name = line.getStringByHeader("name");
    IloNode node(_env, line.getFloatByHeader("x"),
        line.getFloatByHeader("y"), 0, name);
    node.setKey(name);

    _graph.associateByCoordsInFile (node, coordFileName);

    ++it;
  }
  csvNodeReader.end();
}
```

Define the createVehicles function

You create vehicles in the same way you did in Chapter 2, *Modeling a Vehicle Routing Problem*. You use csv reader functionality to input vehicle data from a csv file. The vehicles have start and end visits. You add side constraints that the vehicles must leave the depot after it opens and return to the depot before it closes. You set the capacities of the vehicles using `IloVehicle::setCapacity` and the dimension `_weight`. Using

`IloVehicle::setCost`, the cost of each vehicle is set to be directly proportional to the dimension `_time`. This code is provided for you:

```
void RoutingModel::createVehicles(char* vehicleFileName) {
  IloCsvReader csvVehicleReader(_env, vehicleFileName);
  IloCsvReader::LineIterator  it(csvVehicleReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char * namefirst = line.getStringByHeader("first");
    char * namelast = line.getStringByHeader("last");
    char * name = line.getStringByHeader("name");
    IloNum capacity = line.getFloatByHeader("capacity");
    IloNode node1 = IloNode::Find(_env, namefirst);
    IloNode node2 = IloNode::Find(_env, namelast);
    IloVisit first(node1, "depot");
    _mdl.add(first.getCumulVar(_weight) == 0);
    _mdl.add(first.getCumulVar(_time) >= line.getFloatByHeader("open"));
    IloVisit last(node2, "depot");
    _mdl.add(last.getCumulVar(_time) <= line.getFloatByHeader("close"));
    IloVehicle vehicle(first, last, name);
    vehicle.setCost(_time,   1.0);
    vehicle.setCapacity(_weight, capacity);
    _mdl.add(vehicle);
    ++it;
  }
  csvVehicleReader.end();
}
```

Define the createVisits function

You create visits in the same way you did in Chapter 2, *Modeling a Vehicle Routing Problem*, except that you set a penalty cost for not performing a visit. You use csv reader functionality to input visit data from a csv file. A visit must be associated to a node, its location. A visit also has a quantity—the amount of goods delivered to the location. A visit can have a minimum time and a maximum time during which it can be performed—a time window. Additionally, visits have a drop time—the amount time required to perform the visit. These side constraints are modeled using the dimensions `_time` and `_weight` and the delay, transit, and cumulative variables associated with the visit.

You use the member function `IloVisit::setPenaltyCost` to set the cost of not performing a visit to 1000 cost units. This allows the visit to not be performed. During the solution improvement phase of problem solving, you will want to be able to allow some visits to be temporarily unperformed. The penalty cost allows you to do this. See the section "Solve" on page 90 for more information.

Step 7 Set the penalty cost on unperformed visits

Add the following code after the comment
`//Set the penalty cost on unperformed visits`

```
    visit.setPenaltyCost(1000);
```

Here is the complete code defining the `createVisits` function:

```
void RoutingModel::createVisits(char* visitsFileName) {
  IloCsvReader csvVisitReader(_env, visitsFileName);
  IloCsvReader::LineIterator  it(csvVisitReader);
  while(it.ok()){
    IloCsvLine line = *it;
    //read visit data from files
    char * visitName =  line.getStringByHeader("name");
    char * nodeName = line.getStringByHeader("node");
    IloNum quantity = line.getFloatByHeader("quantity");
    IloNum minTime  = line.getFloatByHeader("minTime");
    IloNum maxTime  = line.getFloatByHeader("maxTime");
    IloNum dropTime = line.getFloatByHeader("dropTime");
    IloNode node = IloNode::Find(_env, nodeName);
    IloVisit visit(node, visitName);
    _mdl.add(visit.getDelayVar(_time) == dropTime);
    _mdl.add(visit.getTransitVar(_weight) == quantity);
    _mdl.add(minTime <= visit.getCumulVar(_time) <= maxTime);
    _mdl.add(visit);

    visit.setPenaltyCost(1000);

    ++it;
  }
  csvVisitReader.end();
}
```

Solve

Because the standard approach to vehicle routing problems described in *Lesson 3 Solving a Vehicle Routing Problem* can be inefficient when tackling the problem of minimizing the number of vehicles used to solve a VRP, this lesson presents another method, based on the use of instances of `IloRoutingSolution` and dynamic insertion of visits.

Here is a short description of the proposed heuristic:

1. Find a first solution.

2. Save the current solution in `_solution`, an instance of `IloRoutingSolution`.

3. Improve the first solution using neighborhoods.

4. Close any empty vehicles so that they cannot take part in any solution.
5. Let *v* be the non-empty vehicle with the fewest visits. Empty *v* by constraining the visits assigned to it so that they must be performed by some other vehicle, disabling *v*, and finding a new solution. (There will always be a solution if the visits have a finite penalty cost and can therefore be unperformed.)
6. Improve the new solution, minimizing its cost.
7. If there are no unperformed visits, go to 2.
8. If there are unperformed visits, enable *v*, and restore `solution`.
9. The solution with the minimal number of vehicles is `solution`.

As in Chapter 3, *Solving a Vehicle Routing Problem*, you create the `RoutingSolver` class, which is used to solve the vehicle routing problem.

Declare the RoutingSolver class

The code for the declaration of the class `RoutingSolver` is provided for you:

```
class RoutingSolver {
  IloEnv             _env;
  IloModel           _mdl;
  IloSolver          _solver;
  IloDispatcher      _dispatcher;
  IloRoutingSolution  _solution;
  IloGoal            _instantiateCost;
  IloGoal            _generateGoal;
  IloGoal            _restoreSolution;
  IloVehicle    getShortestRoute ();

public:
  RoutingSolver(RoutingModel mdl);
  ~RoutingSolver() {}
  IloBool findFirstSolution ();
  void improveWithNhood ();
  void closeEmptyVehicles ();
  void reduceActiveVehicles ();
  void printInformation(const char* =0) const;
};
```

There are only a few differences between the `RoutingSolver` class used in a standard VRP and the `RoutingSolver` class used in this problem. There are three additional member functions: `getShortestRoute`, `closeEmptyVehicles`, and `reduceActiveVehicles`. These member functions are explained in the following sections.

Define the RoutingSolver constructor

The RoutingSolver constructor is defined as in Chapter 3, *Solving a Vehicle Routing Problem*. It takes an instance of RoutingModel as a parameter. The environment, model, solver, dispatcher, and solution are initialized. The goal to instantiate cost is created using the function IloDichotomize. You use the predefined first solution generation heuristic IloSavingsGenerate to create the first solution. The goal to restore the solution is created using IloRestoreSolution. This constructor will be called from the main function. This code is provided for you:

```
 RoutingSolver::RoutingSolver(RoutingModel mdl):
  _env(mdl.getEnv()),
  _mdl(mdl.getModel()),
  _solver(mdl.getModel()),
  _dispatcher(_solver),
  _solution(mdl.getModel()){
    _instantiateCost =
        IloDichotomize(_env, _dispatcher.getCostVar(), IloFalse);
    _generateGoal = IloSavingsGenerate(_env) && _instantiateCost;
    _restoreSolution = IloRestoreSolution(_env, _solution);
  }
```

Define the findFirstSolution function

Now, you create the function that searches for the first solution. This function is the same as the one you created in Chapter 3, *Solving a Vehicle Routing Problem*. You search for a solution using _generateGoal and store the solution and its cost using the member function IloRoutingSolution::store. This code is provided for you:

```
IloBool RoutingSolver::findFirstSolution() {
  if (!_solver.solve(_generateGoal)) {
    _solver.error() << "Infeasible Routing Plan" << endl;
    return IloFalse;
  }
  _solution.store(_solver);
  return IloTrue;
}
```

Define the improveWithNHood function

After you have found a first solution, you create the neighborhoods you will use in the solution improvement phase. As in Chapter 3, *Solving a Vehicle Routing Problem*, you use the predefined neighborhoods IloRelocate, IloExchange, IloCross, IloTwoOpt, and IloOrOpt. Additionally, you use three other predefined neighborhoods: IloMakePerformed, IloMakeUnperformed, and IloSwapPerform. The function IloMakePerformed returns a neighborhood that modifies a solution by inserting an unperformed visit after a performed one. The function IloMakeUnperformed returns a neighborhood that modifies a solution by causing a performed visit to be unperformed. The function `IloSwapPerform` returns a neighborhood that modifies a solution by exchanging a performed visit with an unperformed one. These neighborhoods are used to improve a solution after emptying a vehicle and constraining the visits assigned to it to be performed by some other vehicle.

For more information about how predefined neighborhoods work, see Appendix B, *Predefined Neighborhoods*.

Step 8 Create the neighborhoods

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

```
void RoutingSolver::improveWithNhood() {
  IloNHood nhood = (IloRelocate(_env) + IloExchange(_env) + IloCross(_env))
                 + (IloTwoOpt(_env) + IloOrOpt(_env))
                 + (IloMakePerformed(_env)
                  + IloMakeUnperformed(_env)
                  + IloSwapPerform(_env));
```

The rest of the function is defined as in Chapter 3, *Solving a Vehicle Routing Problem*. You use the function `IloSingleMove` to return a goal that makes a single local move as defined by a neighborhood and a search heuristic. The following code is provided for you:

```
  _solver.out() << "Improving solution" << endl;
  IloGoal improve = IloSingleMove(_env,
                                  _solution,
                                  nhood,
                                  IloImprove(_env),
                                  _instantiateCost);
  while (_solver.solve(improve)) {
  }
  _solver.solve(_restoreSolution);
}
```

Define the closeEmptyVehicles function

Next, you define the `closeEmptyVehicles` function. You use an instance of `IloIterator` and the member function `IloRoutingSolution::getRouteSize` to locate any vehicles that have an empty route. Vehicles that are not used in the solution have a route size of 0 (zero). You constrain these vehicles to be closed by adding the constraint that the next-variable of their first visit is set equal to their last visit. In other words, they never leave the depot.

Step 9 Close empty vehicles

Add the following code after the comment `//Close empty vehicles`

```
void RoutingSolver::closeEmptyVehicles() {
  for (IloIterator<IloVehicle> iter(_env); iter.ok(); ++iter) {
    IloVehicle vehicle = *iter;
    IloInt size = _solution.getRouteSize(vehicle);
    if (size == 0) {
      _mdl.add(vehicle.getFirstVisit().getNextVar() == vehicle.getLastVisit());
    }
  }
}
```

Define the getShortestRoute function

Now you locate the vehicle with the smallest number of visits, since this is vehicle that you will want to empty. An `IloIterator` is used to search through the vehicles in the problem. The member function `IloRoutingSolution::getRouteSize` returns the number of visits in the stored route of each vehicle.

Step 10 Find the vehicle with the shortest route

Add the following code after the comment
`//Find the vehicle with the shortest route`

```
IloVehicle RoutingSolver::getShortestRoute() {
  IloInt bestSize = 1000000;
  IloVehicle bestVehicle;
  for (IloIterator<IloVehicle> iter(_env); iter.ok(); ++iter) {
    IloVehicle vehicle = *iter;
    IloInt size = _solution.getRouteSize(vehicle);
    if (size < bestSize && size != 0) {
      bestSize = size;
      bestVehicle = vehicle;
    }
  }
  return bestVehicle;
}
```

Define the reduceActiveVehicles function

To reduce the number of active vehicles, you first use the function `getShortestRoute` to find the vehicle with the smallest number of visits. You make a copy of the solution and use an `IloRoutingSolution::RouteIterator` to move through the route of the selected vehicle. You add a vehicle incompatibility constraint on each visit associated with the vehicle, stating that the visit should not be assigned to the vehicle you are trying to empty.

Step 11 Empty the vehicle

Add the following code after the comment `//Empty the vehicle`

```
void RoutingSolver::reduceActiveVehicles () {
  IloVehicle vehicle = getShortestRoute();
  IloRoutingSolution solCopy(_env);
  IloBool vehicleClosed = IloTrue;
  IloGoal sync = IloRestoreSolution(_env, _solution)
              && IloStoreSolution(_env, _solution);
  while (vehicleClosed && vehicle.getImpl()) {
    solCopy.copy(_solution);
    _solver.out() << "Emptying " << vehicle.getName() << " ..." << endl;
    IloAnd andCt(_env);
    _mdl.add(andCt);
    for (IloRoutingSolution::RouteIterator iter(solCopy, vehicle);
         iter.ok();
         ++iter) {
      IloVisit visit = *iter;
      if (!visit.isFirstVisit() && !visit.isLastVisit()) {
        andCt.add(vehicle != visit.getVehicleVar());
```

You then remove the visit from the solution, which sets the state of the visit to unperformed in the solution. You use the function `IloInsertVisit` to insert the unperformed visit back into the routing plan. This goal always succeeds because the visit has a finite penalty cost, which allows it be unperformed.

Step 12 Insert the unperformed visits into the routing plan

Add the following code after the comment
`//Insert the unperformed visits into the routing plan`

```
        _solution.remove(visit);
        _solver.solve(sync);
        IloGoal insert = IloInsertVisit(_env, visit, _solution);
        _solver.solve(insert);
        _solution.add(visit);
        _solution.store(_solver);
      }
    }
```

Then you improve the solution, using the neighborhoods `IloMakePerformed`, `IloMakeUnperformed`, and `IloSwapPerform` to try to perform the visits belonging to the emptied vehicle. If, after improvement, the number of unperformed visits is 0 (zero) and the vehicle is still empty, then this vehicle is closed by constraining the next-variable of the first visit to be set equal to the last visit of the vehicle. You then search for the next vehicle to empty, that is the one whose route size is now shortest. When you cannot empty any more vehicles and still perform all the visits, the loop breaks and the last vehicle that was unsuccessfully emptied is added back to the routing solution.

Step 13 Improve the solution

Add the following code after the comment `//Improve the solution`

```
    improveWithNhood();
    if (_solution.getNumberOfUnperformedVisits() == 0
        && _solution.getRouteSize(vehicle) == 0) {
      _mdl.add(vehicle.getFirstVisit().getNextVar()
              == vehicle.getLastVisit());
      vehicle = getShortestRoute();
    }
    else {
      _mdl.remove(andCt);
      _solution.copy(solCopy);
      vehicleClosed = IloFalse;
    }
  }
```

Finally, you restore the last solution found, which is the solution that uses the smallest number of vehicles.

Step 14 Restore the solution

Add the following code after the comment `//Restore the solution`

```
  _solver.solve(_restoreSolution);
}
```

Define the printInformation function

The `printInformation` function is the same as in Chapter 3, *Solving a Vehicle Routing Problem*. This code is provided for you:

```
void RoutingSolver::printInformation(const char* heading) const {
  if(heading)
      _solver.out()<<heading<<endl;
  _solver.printInformation();
  _dispatcher.printInformation();
  _solver.out() << "===============" << endl
    << "Cost         : " << _dispatcher.getTotalCost() << endl
    << "Number of vehicles used : "
    << _dispatcher.getNumberOfVehiclesUsed() << endl
    << "Solution     : " << endl
    << _dispatcher << endl;
}
```

Define the main function

After you finish creating the `RoutingModel` and `RoutingSolver` classes and the `printInformation` function, you use them in the `main` function. You can use command line syntax to pass the names of input files to the model. If you do not specify input files, the defaults will be used. In the `main` function, you first create an environment. Then you create an instance of the `RoutingModel` class, which takes the environment and input files as parameters. You create an instance of the `RoutingSolver` class. This takes one parameter, the model. You use an `if` loop to find a solution. If Dispatcher finds a first solution, you print the information and then improve the solution. Then you close all empty vehicles. You use the function `reduceActiveVehicles` to search for the vehicle with the fewest number of visits and empty it. You try to find a new solution without using this vehicle. If you find a new solution, you improve it. After solution improvement, you empty the vehicle that now has the fewest number of visits. When you cannot empty any more vehicles and still find a solution, you have minimized the number of vehicles. This solution is then printed. The following code is provided for you:

```
int main(int argc, char * argv[]) {
  IloEnv env;
  try {
    RoutingModel mdl(env, argc, argv);
    RoutingSolver solver(mdl);
    if (solver.findFirstSolution()) {
      solver.printInformation("***First Solution***");
      solver.improveWithNhood();
      solver.printInformation("***Solution after improvements with nhood***");
      solver.closeEmptyVehicles();
      solver.reduceActiveVehicles();
      solver.printInformation("***Solution after reducing active vehicles***");
    }
  } catch(IloException& ex) {
    cerr << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Step 15 Compile and run the program

Compile and run the program. You will get results that show the routing plan and information for the first solution, the initial improved solution, and the solution found after reducing the number of active vehicles. The first solution uses 20 vehicles. The initial improved solution uses 17 vehicles. The solution found after reducing the number of active vehicles uses 16 vehicles.

First solution information

The first solution phase finds a solution using 20 vehicles with a total cost of 2554.03 units:

```
***First Solution***
Number of fails              : 50659
Number of choice points      : 1077
Number of variables          : 5504
Number of constraints        : 881
Reversible stack (bytes)     : 578904
Solver heap (bytes)          : 2504836
Solver global heap (bytes)   : 1187724
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 13164
Total memory used (bytes)    : 4353040
Elapsed time since creation  : 9.483
Number of nodes              : 201
Number of visits             : 300
Number of vehicles           : 50
Number of dimensions         : 2
Number of accepted moves     : 0
===============
Cost         : 2554.03
Number of vehicles used : 20
```

Improved Solution Information

The solution improvement phase finds a solution using 17 vehicles with a total cost of 2073.11 units-:

```
***Solution after improvements with nhood***
Number of fails              : 0
Number of choice points      : 0
Number of variables          : 5504
Number of constraints        : 878
Reversible stack (bytes)     : 578904
Solver heap (bytes)          : 2089268
Solver global heap (bytes)   : 1203804
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 35196
Total memory used (bytes)    : 3975584
Elapsed time since creation  : 0.07
Number of nodes              : 201
Number of visits             : 300
Number of vehicles           : 50
Number of dimensions         : 2
Number of accepted moves     : 222
===============
Cost         : 2073.11
Number of vehicles used : 17
```

Solution information after reducing active vehicles

After solution improvement, vehicle 20 has the fewest number of visits. (See the section “Complete Output” on page 106.) The `reduceActiveVehicles` function empties vehicle 20, assigns its visits to other vehicles, and improves the solution. Now, vehicle 7 has the fewest number of visits. The `reduceActiveVehicles` function empties vehicle 7, tries to assign its visits to other vehicles, but is not able to do so. The last solution found is restored. The solution after reducing active vehicles uses 16 vehicles with a total cost of 2072.93 units:

```
Emptying vehicle20 ...
Improving solution
Emptying vehicle7 ...
Improving solution
***Solution after reducing active vehicles***
Number of fails               : 0
Number of choice points       : 0
Number of variables           : 5504
Number of constraints         : 917
Reversible stack (bytes)      : 578904
Solver heap (bytes)           : 2089592
Solver global heap (bytes)    : 1207824
And stack (bytes)             : 20124
Or stack (bytes)              : 44244
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 35196
Total memory used (bytes)     : 3979928
Elapsed time since creation   : 0.061
Number of nodes               : 201
Number of visits              : 300
Number of vehicles            : 50
Number of dimensions          : 2
Number of accepted moves      : 238
===============
Cost         : 2072.93
Number of vehicles used : 16
```

The complete program and output are listed in “Complete Program” on page 100. You can also view it online in the `YourDispatcherHome/examples/src/minvehcl.cpp` file.

Review Exercises

1. How do you associate the graph nodes created using `IloDispatcherGraph::Node` with the nodes created using `IloNode`?
2. How is the visit penalty cost used in this problem?

Suggested Answers

Exercise 1

How do you associate the graph nodes created using `IloDispatcherGraph::Node` with the nodes created using `IloNode`?

Suggested Answer

You use the member function `IloDispatcherGraph::associatebyCoordsInFile` to look up the coordinates of a given `IloNode` in a csv file and automatically associate it to the graph node with matching coordinates.

Exercise 2

How is the visit penalty cost used in this problem?

Suggested Answer

You use the member function `IloVisit::setPenaltyCost` to set the cost of not performing a visit to 1000 cost units. This allows the visit to not be performed. During the solution improvement phase of problem solving, you will want to be able to allow some visits to be temporarily unperformed. The penalty cost allows you to do this.

Complete Program

The complete program follows. You can also view it online in the `YourDispatcherHome/examples/src/minvehcl.cpp` file.

```
// -------------------------------------------------------------- -*- C++ -*-
// File: examples/src/minvehcl.cpp
// --------------------------------------------------------------------------


#include <ildispat/ilodispatcher.h>

ILOSTLBEGIN
///////////////////////////////////////////////////////////////////////////////
// Modeling
class RoutingModel {
  IloEnv              _env;
  IloModel            _mdl;
  IloDispatcherGraph  _graph;
  IloDimension2       _time;
  IloDimension1       _weight;

  void addDimensions();
  void loadGraphInformation (char* arcFileName, char* turnFileName);
  void lastMinuteGraphChanges ();
  void createIloNodes(char* nodeFileName, char* coordFileName);
  void createVehicles(char* vehicleFileName);
  void createVisits(char* visitsFileName);

public:
  RoutingModel(IloEnv env, int argc, char* argv[]);
  ~RoutingModel() {}
  IloEnv    getEnv() const { return _env; }
  IloModel  getModel() const { return _mdl; }
};

// Create distance functions for dimensions, add dimensions to model
void RoutingModel::addDimensions() {
  _weight =IloDimension1 (_env, “weight”);
  _mdl.add(_weight);

  IloDistance SP_time =   IloGraphDistance (_graph);
  _time  =IloDimension2 (_env, SP_time, “time”);
  _mdl.add(_time);
}

RoutingModel::RoutingModel( IloEnv env,
                            int argc,
                            char* argv[]):
  _env(env), _mdl(env), _graph(env) {
  addDimensions();

  // Load dispatcher graph information from file, and
  // add instance-specific features.
  char * arcFileName;
  if(argc < 2)  arcFileName =
     (char*) “../../../examples/data/dispatcherGraphData/gridNetwork.csv”;
  else          arcFileName = argv[1];
  char * turnFileName;
  if(argc < 3)  turnFileName =
     (char*) “../../../examples/data/dispatcherGraphData/turnData.csv”;
  else          turnFileName = argv[2];
  loadGraphInformation (arcFileName, turnFileName);

  // Create IloNodes and associate them to graph nodes
  char * nodeFileName;
  if(argc < 4)  nodeFileName =
     (char*) “../../../examples/data/vrp200/vrp200nodes.csv”;
  else          nodeFileName = argv[3];
  char * nodeCoordsFile;
  if(argc < 5)  nodeCoordsFile =
     (char*) “../../../examples/data/dispatcherGraphData/coordTable.csv”;
  else          nodeCoordsFile = argv[4];
  createIloNodes(nodeFileName, nodeCoordsFile);

  // Create vehicles
  char * vehiclesFileName;
  if(argc < 6)  vehiclesFileName =
     (char*) “../../../examples/data/vrp200/vrp200vehicles.csv”;
  else          vehiclesFileName = argv[5];
  createVehicles(vehiclesFileName);

  // Create visits
  char * visitsFileName;
  if(argc < 7)  visitsFileName =
     (char*) “../../../examples/data/vrp200/vrp200visits.csv”;
  else          visitsFileName = argv[6];
  createVisits(visitsFileName);
}

// Load network topology and travel costs from files.
void RoutingModel::loadGraphInformation ( char* arcFileName,
                                          char* turnFileName)    {
  _graph.createArcsFromFile (arcFileName);
  _graph.loadArcDimensionDataFromFile (arcFileName, _time);
  _graph.loadTurnDimensionDataFromFile(turnFileName, _time);
  lastMinuteGraphChanges();
}

// Make modifications to network conditions based on latest information.
void RoutingModel::lastMinuteGraphChanges () {
  _graph.forbidArcUse(_graph.getArcByEnds(2785-1, 2785));
  _graph.forbidArcUse(_graph.getArcByEnds(2785+1, 2785));
  _graph.forbidArcUse(_graph.getArcByEnds(2785-56, 2785));
  _graph.setTurnPenalty(_graph.getArc(1323), _graph.getArc(1544), _time, 12);
}

// Create IloNodes
void RoutingModel::createIloNodes(char* nodeFileName, char* coordFileName) {
  IloCsvReader csvNodeReader(_env, nodeFileName);
  IloCsvReader::LineIterator  it(csvNodeReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char * name = line.getStringByHeader(“name”);
    IloNode node(_env, line.getFloatByHeader(“x”),
        line.getFloatByHeader(“y”), 0, name);
    node.setKey(name);

    _graph.associateByCoordsInFile (node, coordFileName);

    ++it;
  }
  csvNodeReader.end();
}

// Create vehicles
void RoutingModel::createVehicles(char* vehicleFileName) {
  IloCsvReader csvVehicleReader(_env, vehicleFileName);
  IloCsvReader::LineIterator  it(csvVehicleReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char * namefirst = line.getStringByHeader(“first”);
    char * namelast = line.getStringByHeader(“last”);
    char * name = line.getStringByHeader(“name”);
    IloNum capacity = line.getFloatByHeader(“capacity”);
    IloNode node1 = IloNode::Find(_env, namefirst);
    IloNode node2 = IloNode::Find(_env, namelast);
    IloVisit first(node1, “depot”);
    _mdl.add(first.getCumulVar(_weight) == 0);
    _mdl.add(first.getCumulVar(_time) >= line.getFloatByHeader(“open”));
    IloVisit last(node2, “depot”);
    _mdl.add(last.getCumulVar(_time) <= line.getFloatByHeader(“close”));
    IloVehicle vehicle(first, last, name);
    vehicle.setCost(_time,   1.0);
    vehicle.setCapacity(_weight, capacity);
    _mdl.add(vehicle);
    ++it;
  }
  csvVehicleReader.end();
}

// Create visits
void RoutingModel::createVisits(char* visitsFileName) {
  IloCsvReader csvVisitReader(_env, visitsFileName);
  IloCsvReader::LineIterator  it(csvVisitReader);
  while(it.ok()){
    IloCsvLine line = *it;
    //read visit data from files
    char * visitName =  line.getStringByHeader(“name”);
    char * nodeName = line.getStringByHeader(“node”);
    IloNum quantity = line.getFloatByHeader(“quantity”);
    IloNum minTime  = line.getFloatByHeader(“minTime”);
    IloNum maxTime  = line.getFloatByHeader(“maxTime”);
    IloNum dropTime = line.getFloatByHeader(“dropTime”);
    IloNode node = IloNode::Find(_env, nodeName);
    IloVisit visit(node, visitName);
    _mdl.add(visit.getDelayVar(_time) == dropTime);
    _mdl.add(visit.getTransitVar(_weight) == quantity);
    _mdl.add(minTime <= visit.getCumulVar(_time) <= maxTime);
    _mdl.add(visit);

    visit.setPenaltyCost(1000);

    ++it;
  }
  csvVisitReader.end();
}

///////////////////////////////////////////////////////////////////////////////
// Solving
class RoutingSolver {
  IloEnv             _env;
  IloModel           _mdl;
  IloSolver          _solver;
  IloDispatcher      _dispatcher;
  IloRoutingSolution  _solution;
  IloGoal            _instantiateCost;
  IloGoal            _generateGoal;
  IloGoal            _restoreSolution;
  IloVehicle    getShortestRoute ();

public:
  RoutingSolver(RoutingModel mdl);
  ~RoutingSolver() {}
  IloBool findFirstSolution ();
  void improveWithNhood ();
  void closeEmptyVehicles ();
  void reduceActiveVehicles ();
  void printInformation(const char* =0) const;
```

```
};

RoutingSolver::RoutingSolver(RoutingModel mdl):
  _env(mdl.getEnv()),
  _mdl(mdl.getModel()),
  _solver(mdl.getModel()),
  _dispatcher(_solver),
  _solution(mdl.getModel()){
    _instantiateCost =
        IloDichotomize(_env, _dispatcher.getCostVar(), IloFalse);
    _generateGoal = IloSavingsGenerate(_env) && _instantiateCost;
    _restoreSolution = IloRestoreSolution(_env, _solution);
  }

// Solving : find first solution
IloBool RoutingSolver::findFirstSolution() {
  if (!_solver.solve(_generateGoal)) {
    _solver.error() << “Infeasible Routing Plan” << endl;
    return IloFalse;
  }
  _solution.store(_solver);
  return IloTrue;
}

//Improve solution using nhood
void RoutingSolver::improveWithNhood() {
  IloNHood nhood = (IloRelocate(_env) + IloExchange(_env) + IloCross(_env))
                 + (IloTwoOpt(_env) + IloOrOpt(_env))
                 + (IloMakePerformed(_env)
                  + IloMakeUnperformed(_env)
                  + IloSwapPerform(_env));

  _solver.out() << “Improving solution” << endl;
  IloGoal improve = IloSingleMove(_env,
                                  _solution,
                                  nhood,
                                  IloImprove(_env),
                                  _instantiateCost);
  while (_solver.solve(improve)) {
  }
  _solver.solve(_restoreSolution);
}

void RoutingSolver::closeEmptyVehicles() {
  for (IloIterator<IloVehicle> iter(_env); iter.ok(); ++iter) {
    IloVehicle vehicle = *iter;
    IloInt size = _solution.getRouteSize(vehicle);
    if (size == 0) {
      _mdl.add(vehicle.getFirstVisit().getNextVar() == vehicle.getLastVisit());
    }
  }
}

IloVehicle RoutingSolver::getShortestRoute() {
  IloInt bestSize = 1000000;
  IloVehicle bestVehicle;
  for (IloIterator<IloVehicle> iter(_env); iter.ok(); ++iter) {
    IloVehicle vehicle = *iter;
    IloInt size = _solution.getRouteSize(vehicle);
    if (size < bestSize && size != 0) {
      bestSize = size;
      bestVehicle = vehicle;
    }
  }
  return bestVehicle;
}

void RoutingSolver::reduceActiveVehicles () {
  IloVehicle vehicle = getShortestRoute();
  IloRoutingSolution solCopy(_env);
  IloBool vehicleClosed = IloTrue;
  IloGoal sync = IloRestoreSolution(_env, _solution)
              && IloStoreSolution(_env, _solution);
  while (vehicleClosed && vehicle.getImpl()) {
    solCopy.copy(_solution);
    _solver.out() << “Emptying “ << vehicle.getName() << “ ...” << endl;
    IloAnd andCt(_env);
    _mdl.add(andCt);
    for (IloRoutingSolution::RouteIterator iter(solCopy, vehicle);
         iter.ok();
         ++iter) {
      IloVisit visit = *iter;
      if (!visit.isFirstVisit() && !visit.isLastVisit()) {
        andCt.add(vehicle != visit.getVehicleVar());

        _solution.remove(visit);
        _solver.solve(sync);
        IloGoal insert = IloInsertVisit(_env, visit, _solution);
        _solver.solve(insert);
        _solution.add(visit);
        _solution.store(_solver);
      }
    }

    improveWithNhood();
    if (_solution.getNumberOfUnperformedVisits() == 0
        && _solution.getRouteSize(vehicle) == 0) {
      _mdl.add(vehicle.getFirstVisit().getNextVar()
              == vehicle.getLastVisit());
      vehicle = getShortestRoute();
    }
    else {
      _mdl.remove(andCt);
      _solution.copy(solCopy);
      vehicleClosed = IloFalse;
    }
  }

  _solver.solve(_restoreSolution);
}

// Display Dispatcher information
void RoutingSolver::printInformation(const char* heading) const {
  if(heading)
      _solver.out()<<heading<<endl;
  _solver.printInformation();
  _dispatcher.printInformation();
  _solver.out() << "==============" << endl
    << "Cost         : " << _dispatcher.getTotalCost() << endl
    << "Number of vehicles used : "
    << _dispatcher.getNumberOfVehiclesUsed() << endl
    << "Solution     : " << endl
    << _dispatcher << endl;
}

///////////////////////////////////////////////////////////////////////////////
int main(int argc, char * argv[]) {
  IloEnv env;
  try {
    RoutingModel mdl(env, argc, argv);
    RoutingSolver solver(mdl);
    if (solver.findFirstSolution()) {
      solver.printInformation("***First Solution***");
      solver.improveWithNhood();
      solver.printInformation("***Solution after improvements with nhood***");
      solver.closeEmptyVehicles();
      solver.reduceActiveVehicles();
      solver.printInformation("***Solution after reducing active vehicles***");
    }
  } catch(IloException& ex) {
    cerr << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Complete Output

```
/**
***First Solution***
Number of fails               : 50659
Number of choice points       : 1077
Number of variables           : 5504
Number of constraints         : 881
Reversible stack (bytes)      : 578904
Solver heap (bytes)           : 2504836
Solver global heap (bytes)    : 1187724
And stack (bytes)             : 20124
Or stack (bytes)              : 44244
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 13164
Total memory used (bytes)     : 4353040
Elapsed time since creation   : 9.483
Number of nodes               : 201
Number of visits              : 300
Number of vehicles            : 50
Number of dimensions          : 2
Number of accepted moves      : 0
===============
Cost         : 2554.03
Number of vehicles used : 20
Solution     :
```

```
Unperformed visits : None
vehicle1 :
 -> depot weight[0] time[0..95.8002] -> visit34 weight[0]
time[0.89247..96.6927] -> visit54 weight[20] time[10.9266..106.727] -> visit74
weight[40] time[20.9608..116.761] -> visit94 weight[60] time[30.9949..126.795]
-> visit114 weight[80] time[41.0291..136.829] -> visit134 weight[100]
time[51.0632..146.863] -> visit154 weight[120] time[61.0974..156.898] ->
visit174 weight[140] time[71.1315..166.932] -> visit194 weight[160]
time[81.1657..176.966] -> visit14 weight[180] time[91.1998..187] -> depot
weight[200] time[102.092..230]
vehicle2 :
 -> depot weight[0] time[0..48.4869] -> visit117 weight[0]
time[0.808264..49.2952] -> visit137 weight[2] time[10.8423..59.3293] ->
visit157 weight[4] time[20.8764..69.3633] -> visit177 weight[6]
time[30.9104..79.3974] -> visit197 weight[8] time[40.9445..89.4314] -> visit36
weight[10] time[51.2..99.687] -> visit56 weight[29] time[61.2348..109.722] ->
visit76 weight[48] time[71.2696..119.757] -> visit96 weight[67]
time[81.3044..129.791] -> visit116 weight[86] time[91.3391..139.826] ->
visit136 weight[105] time[101.374..149.861] -> visit156 weight[124]
time[111.409..159.896] -> visit176 weight[143] time[121.443..169.93] ->
visit196 weight[162] time[131.478..179.965] -> visit16 weight[181]
time[141.513..190] -> depot weight[200] time[152.371..230]
vehicle3 :
 -> depot weight[0] time[0..55.5455] -> visit97 weight[0..1]
time[0.808264..56.3537] -> visit191 weight[2..3] time[67..67.1946] -> visit39
weight[14..15] time[77.167..77.3617] -> visit59 weight[31..32]
time[87.2006..87.3953] -> visit79 weight[48..49] time[97.2342..97.4289] ->
visit99 weight[65..66] time[107.268..107.462] -> visit119 weight[82..83]
time[117.301..117.496] -> visit139 weight[99..100] time[127.335..127.53] ->
visit159 weight[116..117] time[137.369..137.563] -> visit179 weight[133..134]
time[147.402..147.597] -> visit199 weight[150..151] time[157.436..157.63] ->
visit19 weight[167..168] time[167.469..167.664] -> visit7 weight[184..185]
time[177.731..177.926] -> visit27 weight[189..190] time[187.768..187.963] ->
visit47 weight[194..195] time[197.805..198] -> depot weight[199..200]
time[208.427..230]
vehicle4 :
 -> depot weight[0] time[0..76.375] -> visit166 weight[0..1]
time[0.409416..76.7844] -> visit186 weight[3..4] time[10.4645..86.8395] ->
visit85 weight[6..7] time[20.7051..97.0801] -> visit105 weight[32..33]
time[30.746..107.121] -> visit125 weight[58..59] time[40.787..117.162] ->
visit145 weight[84..85] time[50.8279..127.203] -> visit165 weight[110..111]
time[60.8689..137.244] -> visit185 weight[136..137] time[70.9098..147.285] ->
visit37 weight[162..163] time[81.096..157.471] -> visit57 weight[164..165]
time[91.1301..167.505] -> visit77 weight[166..167] time[101.164..177.539] ->
visit17 weight[168..169] time[111.198..187.573] -> visit5 weight[170..171]
time[121.384..197.759] -> visit6 weight[196..197] time[131.625..208] -> depot
weight[199..200] time[142.034..230]
vehicle5 :
 -> depot weight[0] time[0..5.70798] -> visit40 weight[0]
time[0.741903..6.44988] -> visit60 weight[9] time[10.7752..16.4832] -> visit80
weight[18] time[20.8086..26.5166] -> visit100 weight[27] time[30.8419..36.5499]
-> visit120 weight[36] time[40.8753..46.5832] -> visit140 weight[45]
time[50.9086..56.6166] -> visit160 weight[54] time[60.9419..66.6499] ->
visit180 weight[63] time[70.9753..76.6832] -> visit200 weight[72]
time[81.0086..86.7166] -> visit20 weight[81] time[91.0419..96.7499] -> visit9
weight[90] time[101.292..107] -> visit3 weight[106] time[111.542..136.8] ->
visit23 weight[119] time[121.576..146.833] -> visit43 weight[132]
time[131.609..156.867] -> visit63 weight[145] time[141.642..166.9] -> visit83
```

```
weight[158] time[151.676..176.933] -> visit103 weight[171]
time[161.709..186.967] -> visit123 weight[184] time[171.742..197] -> visit26
weight[197] time[182.663..208] -> depot weight[200] time[193.073..230]
vehicle6 :
 -> depot weight[0] time[0..3.69563] -> visit158 weight[0]
time[0.490256..4.18588] -> visit178 weight[12] time[10.5339..14.2296] ->
visit198 weight[24] time[20.5776..24.2732] -> visit67 weight[36]
time[30.7804..34.4761] -> visit87 weight[41] time[40.8176..44.5133] -> visit107
weight[46] time[50.8548..54.5505] -> visit127 weight[51] time[60.892..64.5877]
-> visit147 weight[56] time[70.9292..74.6249] -> visit167 weight[61]
time[80.9664..84.662] -> visit187 weight[66] time[91.0036..94.6992] -> visit188
weight[71] time[101.304..105] -> visit45 weight[80] time[111.641..137.094] ->
visit65 weight[106] time[121.682..147.135] -> visit25 weight[132]
time[131.723..157.176] -> visit18 weight[158] time[142.035..167.488] -> visit38
weight[170] time[152.078..177.532] -> visit58 weight[182]
time[162.122..187.576] -> visit46 weight[194] time[172.491..197.945] -> visit66
weight[197] time[182.546..208] -> depot weight[200] time[192.956..230]
vehicle7 :
 -> depot weight[0] time[0..5.95457] -> visit70 weight[0..1]
time[0.589454..6.54403] -> visit90 weight[16..17] time[10.6236..16.5782] ->
visit110 weight[32..33] time[20.6578..26.6123] -> visit130 weight[48..49]
time[30.6919..36.6465] -> visit150 weight[64..65] time[40.7261..46.6806] ->
visit170 weight[80..81] time[50.7602..56.7148] -> visit190 weight[96..97]
time[60.7944..66.7489] -> visit11 weight[112..113] time[71.0454..77] -> visit10
weight[124..125] time[81.2965..150.78] -> visit30 weight[140..141]
time[91.3307..160.815] -> visit50 weight[156..157] time[101.365..170.849] ->
visit1 weight[172..173] time[111.742..181.226] -> visit21 weight[182..183]
time[121.778..191.262] -> visit2 weight[192..193] time[132.516..202] -> depot
weight[199..200] time[142.926..230]
vehicle8 :
 -> depot weight[0] time[0..5.71284] -> visit41 weight[0..11]
time[0.446525..6.15937] -> visit61 weight[10..21] time[10.4823..16.1952] ->
visit81 weight[20..31] time[20.5181..26.231] -> visit101 weight[30..41]
time[30.5539..36.2668] -> visit121 weight[40..51] time[40.5897..46.3026] ->
visit141 weight[50..61] time[50.6255..56.3384] -> visit161 weight[60..71]
time[60.6613..66.3742] -> visit181 weight[70..81] time[70.6971..76.4099] ->
visit163 weight[80..91] time[81.0037..86.7165] -> visit183 weight[93..104]
time[91.037..96.7498] -> visit189 weight[106..117] time[101.287..107] ->
visit143 weight[122..133] time[111.537..141.029] -> visit4 weight[135..146]
time[149..151.446] -> visit22 weight[154..165] time[159.401..161.847] ->
visit42 weight[161..172] time[169.439..171.886] -> visit62 weight[168..179]
time[179.478..181.924] -> visit82 weight[175..186] time[189.516..191.962] ->
visit102 weight[182..193] time[199.554..202] -> depot weight[189..200]
time[209.964..230]
vehicle9 :
 -> depot weight[0] time[0..3.12711] -> visit142 weight[0..23]
time[0.409943..3.53706] -> visit162 weight[7..30] time[10.4481..13.5752] ->
visit182 weight[14..37] time[20.4863..23.6134] -> visit86 weight[21..44]
time[30.999..34.1262] -> visit106 weight[24..47] time[41.0541..44.1812] ->
visit126 weight[27..50] time[51.1092..54.2363] -> visit146 weight[30..53]
time[61.1643..64.2914] -> visit98 weight[33..56] time[71.5335..74.6606] ->
visit118 weight[45..68] time[81.5772..84.7043] -> visit138 weight[57..80]
time[91.6209..94.748] -> visit168 weight[69..92] time[101.873..105] -> visit78
weight[78..101] time[112.125..153.476] -> visit13 weight[90..113]
time[159..164.073] -> visit122 weight[113..136] time[169.281..174.355] ->
visit12 weight[120..143] time[179.858..184.931] -> visit32 weight[139..162]
time[189.892..194.966] -> visit52 weight[158..181] time[199.927..205] -> depot
weight[177..200] time[210.235..230]
```

```
vehicle10 :
 -> depot weight[0] time[0..56.0521] -> visit92 weight[0..32]
time[0.308232..56.3604] -> visit112 weight[19..51] time[10.3427..66.3949] ->
visit132 weight[38..70] time[20.3772..76.4294] -> visit152 weight[57..89]
time[30.4117..86.4639] -> visit172 weight[76..108] time[40.4462..96.4984] ->
visit169 weight[95..127] time[97..107] -> visit192 weight[111..143]
time[107.502..148.665] -> visit24 weight[130..162] time[149..159] -> visit72
weight[149..181] time[159.335..205] -> depot weight[168..200]
time[169.643..230]
vehicle11 :
 -> depot weight[0] time[0..64.929] -> visit195 weight[0..113]
time[61..65.6378] -> visit171 weight[8..121] time[72.3622..77] -> visit148
weight[20..133] time[95..95.9483] -> visit29 weight[29..142] time[106.052..107]
-> visit44 weight[45..158] time[149..158.426] -> visit33 weight[64..177]
time[159.574..169] -> depot weight[87..200] time[169.937..230]
vehicle12 :
 -> depot weight[0] time[0..64.929] -> visit175 weight[0..113]
time[61..65.6378] -> visit151 weight[8..121] time[72.3622..77] -> visit128
weight[20..133] time[95..95.9483] -> visit49 weight[29..142] time[106.052..107]
-> visit64 weight[45..158] time[149..158.426] -> visit53 weight[64..177]
time[159.574..169] -> depot weight[87..200] time[169.937..230]
vehicle13 :
 -> depot weight[0] time[0..64.929] -> visit155 weight[0..113]
time[61..65.6378] -> visit131 weight[8..121] time[72.3622..77] -> visit108
weight[20..133] time[95..95.9483] -> visit69 weight[29..142] time[106.052..107]
-> visit84 weight[45..158] time[149..158.426] -> visit73 weight[64..177]
time[159.574..169] -> depot weight[87..200] time[169.937..230]
vehicle14 :
 -> depot weight[0] time[0..64.929] -> visit135 weight[0..113]
time[61..65.6378] -> visit111 weight[8..121] time[72.3622..77] -> visit88
weight[20..133] time[95..95.9483] -> visit89 weight[29..142] time[106.052..107]
-> visit104 weight[45..158] time[149..158.426] -> visit93 weight[64..177]
time[159.574..169] -> depot weight[87..200] time[169.937..230]
vehicle15 :
 -> depot weight[0] time[0..64.929] -> visit115 weight[0..113]
time[61..65.6378] -> visit91 weight[8..121] time[72.3622..77] -> visit68
weight[20..133] time[95..95.9483] -> visit109 weight[29..142]
time[106.052..107] -> visit124 weight[45..158] time[149..158.426] -> visit113
weight[64..177] time[159.574..169] -> depot weight[87..200] time[169.937..230]
vehicle16 :
 -> depot weight[0] time[0..64.929] -> visit95 weight[0..113] time[61..65.6378]
-> visit71 weight[8..121] time[72.3622..77] -> visit48 weight[20..133]
time[95..95.9483] -> visit129 weight[29..142] time[106.052..107] -> visit144
weight[45..158] time[149..158.426] -> visit133 weight[64..177]
time[159.574..169] -> depot weight[87..200] time[169.937..230]
vehicle17 :
 -> depot weight[0] time[0..64.929] -> visit75 weight[0..113] time[61..65.6378]
-> visit51 weight[8..121] time[72.3622..77] -> visit28 weight[20..133]
time[95..95.9483] -> visit149 weight[29..142] time[106.052..107] -> visit164
weight[45..158] time[149..158.426] -> visit153 weight[64..177]
time[159.574..169] -> depot weight[87..200] time[169.937..230]
vehicle18 :
 -> depot weight[0] time[0..64.929] -> visit55 weight[0..129] time[61..65.6378]
-> visit31 weight[8..137] time[72.3622..77] -> visit8 weight[20..149]
time[95..105] -> visit184 weight[29..158] time[149..158.426] -> visit173
weight[48..177] time[159.574..169] -> depot weight[71..200] time[169.937..230]
vehicle19 :
 -> depot weight[0] time[0..70.2912] -> visit35 weight[0..169] time[61..71] ->
```

```
visit193 weight[8..177] time[159..169] -> depot weight[31..200]
time[169.363..230]
vehicle20 :
 -> depot weight[0] time[0..70.2912] -> visit15 weight[0..192] time[61..71] ->
depot weight[8..200] time[71.7088..230]
vehicle21 : Unused
vehicle22 : Unused
vehicle23 : Unused
vehicle24 : Unused
vehicle25 : Unused
vehicle26 : Unused
vehicle27 : Unused
vehicle28 : Unused
vehicle29 : Unused
vehicle30 : Unused
vehicle31 : Unused
vehicle32 : Unused
vehicle33 : Unused
vehicle34 : Unused
vehicle35 : Unused
vehicle36 : Unused
vehicle37 : Unused
vehicle38 : Unused
vehicle39 : Unused
vehicle40 : Unused
vehicle41 : Unused
vehicle42 : Unused
vehicle43 : Unused
vehicle44 : Unused
vehicle45 : Unused
vehicle46 : Unused
vehicle47 : Unused
vehicle48 : Unused
vehicle49 : Unused
vehicle50 : Unused
Improving solution
***Solution after improvements with nhood***
Number of fails               : 0
Number of choice points       : 0
Number of variables           : 5504
Number of constraints         : 878
Reversible stack (bytes)      : 578904
Solver heap (bytes)           : 2089268
Solver global heap (bytes)    : 1203804
And stack (bytes)             : 20124
Or stack (bytes)              : 44244
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 35196
Total memory used (bytes)     : 3975584
Elapsed time since creation   : 0.07
Number of nodes               : 201
Number of visits              : 300
Number of vehicles            : 50
Number of dimensions          : 2
Number of accepted moves      : 222
===============
Cost         : 2073.11
Number of vehicles used : 17
```

```
Solution     :
Unperformed visits : None
vehicle1 : Unused
vehicle2 : Unused
vehicle3 : Unused
vehicle4 : Unused
vehicle5 :
 -> depot weight[0] time[0..5.70798] -> visit40 weight[0..4]
time[0.741903..6.44988] -> visit60 weight[9..13] time[10.7752..16.4832] ->
visit80 weight[18..22] time[20.8086..26.5166] -> visit100 weight[27..31]
time[30.8419..36.5499] -> visit120 weight[36..40] time[40.8753..46.5832] ->
visit140 weight[45..49] time[50.9086..56.6166] -> visit160 weight[54..58]
time[60.9419..66.6499] -> visit180 weight[63..67] time[70.9753..76.6832] ->
visit200 weight[72..76] time[81.0086..86.7166] -> visit20 weight[81..85]
time[91.0419..96.7499] -> visit9 weight[90..94] time[101.292..107] -> visit83
weight[106..110] time[111.542..118.483] -> visit103 weight[119..123]
time[121.576..128.516] -> visit123 weight[132..136] time[131.609..138.55] ->
visit3 weight[145..149] time[141.642..148.583] -> visit184 weight[158..162]
time[152.059..159] -> visit192 weight[177..181] time[162.394..205] -> depot
weight[196..200] time[172.702..230]
vehicle6 :
 -> depot weight[0] time[0..43.3555] -> visit30 weight[0..56]
time[0.589454..43.9449] -> visit119 weight[16..72] time[10.845..54.2005] ->
visit59 weight[33..89] time[20.8786..64.2341] -> visit31 weight[50..106]
time[67..74.4011] -> visit147 weight[62..118] time[77.261..84.662] -> visit127
weight[67..123] time[87.2981..94.6992] -> visit188 weight[72..128]
time[97.5989..105] -> visit17 weight[81..137] time[107.907..117.811] -> visit97
weight[83..139] time[117.942..127.845] -> visit157 weight[85..141]
time[127.976..137.879] -> visit196 weight[87..143] time[138.231..148.135] ->
visit4 weight[106..162] time[149.096..159] -> visit52 weight[125..181]
time[159.431..205] -> depot weight[144..200] time[169.739..230]
vehicle7 :
 -> depot weight[0] time[0..76.1618] -> visit11 weight[0..124] time[67..77] ->
visit10 weight[12..136] time[77.2511..96.5791] -> visit109 weight[28..152]
time[97..107] -> visit63 weight[44..168] time[107.25..194.749] -> visit32
weight[57..181] time[117.502..205] -> depot weight[76..200] time[127.81..230]
vehicle8 :
 -> depot weight[0] time[0..5.64133] -> visit41 weight[0..5]
time[0.446525..6.08785] -> visit61 weight[10..15] time[10.4823..16.1236] ->
visit81 weight[20..25] time[20.5181..26.1594] -> visit101 weight[30..35]
time[30.5539..36.1952] -> visit121 weight[40..45] time[40.5897..46.231] ->
visit141 weight[50..55] time[50.6255..56.2668] -> visit161 weight[60..65]
time[60.6613..66.3026] -> visit21 weight[70..75] time[70.6971..76.3384] ->
visit163 weight[80..85] time[81.0037..86.645] -> visit183 weight[93..98]
time[91.037..96.6783] -> visit189 weight[106..111] time[101.287..106.928] ->
visit99 weight[122..127] time[111.964..117.605] -> visit137 weight[139..144]
time[122.639..128.28] -> visit117 weight[141..146] time[132.673..138.314] ->
visit25 weight[143..148] time[142.859..148.5] -> visit86 weight[169..174]
time[153.099..158.741] -> visit13 weight[172..177] time[163.359..169] -> depot
weight[195..200] time[173.722..230]
vehicle9 :
 -> depot weight[0] time[0..56.2644] -> visit182 weight[0..4]
time[0.409943..56.6744] -> visit35 weight[7..11] time[61..66.9732] -> visit34
weight[15..19] time[71.3385..77.3117] -> visit36 weight[35..39]
time[81.5948..87.568] -> visit48 weight[54..58] time[95..97.986] -> visit85
weight[63..67] time[105.336..108.322] -> visit105 weight[89..93]
time[115.377..118.363] -> visit145 weight[115..119] time[125.418..128.404] ->
visit45 weight[141..145] time[135.459..138.445] -> visit146 weight[167..171]
```

```
time[145.7..148.686] -> visit26 weight[170..174] time[155.755..158.741] ->
visit173 weight[173..177] time[166.014..169] -> depot weight[196..200]
time[176.377..230]
vehicle10 :
 -> depot weight[0] time[0..45.5461] -> visit112 weight[0..1]
time[0.308232..45.8543] -> visit132 weight[19..20] time[10.3427..55.8888] ->
visit152 weight[38..39] time[20.3772..65.9233] -> visit172 weight[57..58]
time[30.4117..75.9578] -> visit12 weight[76..77] time[40.4462..85.9923] ->
visit43 weight[95..96] time[50.6977..96.2438] -> visit169 weight[108..109]
time[97..106.494] -> visit110 weight[124..125] time[107.421..116.915] ->
visit199 weight[140..141] time[117.676..127.17] -> visit197 weight[157..158]
time[128.351..137.845] -> visit177 weight[159..160] time[138.385..147.879] ->
visit24 weight[161..162] time[149.506..159] -> visit72 weight[180..181]
time[159.841..205] -> depot weight[199..200] time[170.149..230]
vehicle11 :
 -> depot weight[0] time[0..32.7593] -> visit122 weight[0..1]
time[0.409943..33.1692] -> visit54 weight[7..8] time[10.8925..43.6518] ->
visit74 weight[27..28] time[20.9266..53.6859] -> visit195 weight[47..48]
time[61..64.0244] -> visit171 weight[55..56] time[72.3622..75.3866] -> visit187
weight[67..68] time[82.6231..85.6475] -> visit148 weight[72..73]
time[95..95.9483] -> visit29 weight[81..82] time[106.052..107] -> visit134
weight[97..98] time[117.579..127.497] -> visit114 weight[117..118]
time[127.614..137.532] -> visit94 weight[137..138] time[137.648..147.566] ->
visit44 weight[157..158] time[149..158.426] -> visit33 weight[176..177]
time[159.574..169] -> depot weight[199..200] time[169.937..230]
vehicle12 :
 -> depot weight[0] time[0..65.6019] -> visit50 weight[0..21]
time[0.589454..66.1914] -> visit151 weight[16..37] time[67..76.4425] -> visit67
weight[28..49] time[77.261..86.7034] -> visit128 weight[33..54]
time[95..97.0042] -> visit56 weight[42..63] time[105.418..107.422] -> visit136
weight[61..82] time[115.453..117.457] -> visit116 weight[80..101]
time[125.488..127.492] -> visit96 weight[99..120] time[135.522..137.527] ->
visit76 weight[118..139] time[145.557..147.561] -> visit84 weight[137..158]
time[156.422..158.426] -> visit73 weight[156..177] time[166.996..169] -> depot
weight[179..200] time[177.359..230]
vehicle13 :
 -> depot weight[0] time[0..54.2267] -> visit22 weight[0..19]
time[0.409943..54.6366] -> visit155 weight[7..26] time[61..64.9355] -> visit14
weight[15..34] time[71.3385..75.274] -> visit16 weight[35..54]
time[81.5948..85.5303] -> visit108 weight[54..73] time[95..95.9483] -> visit69
weight[63..82] time[106.052..107] -> visit154 weight[79..98]
time[117.579..127.497] -> visit194 weight[99..118] time[127.614..137.532] ->
visit174 weight[119..138] time[137.648..147.566] -> visit64 weight[139..158]
time[149..158.426] -> visit53 weight[158..177] time[159.574..169] -> depot
weight[181..200] time[169.937..230]
vehicle14 :
 -> depot weight[0] time[0..63.3156] -> visit135 weight[0..57]
time[61..64.0244] -> visit111 weight[8..65] time[72.3622..75.3866] -> visit47
weight[20..77] time[82.6231..85.6475] -> visit88 weight[25..82]
time[95..95.9483] -> visit89 weight[34..91] time[106.052..107] -> visit70
weight[50..107] time[116.473..117.865] -> visit90 weight[66..123]
time[126.507..127.899] -> visit1 weight[82..139] time[136.884..138.276] ->
visit143 weight[92..149] time[147.191..148.583] -> visit124 weight[105..162]
time[157.608..159] -> visit92 weight[124..181] time[167.942..205] -> depot
weight[143..200] time[178.251..230]
vehicle15 :
 -> depot weight[0] time[0..54.929] -> visit42 weight[0..72]
time[0.409943..55.339] -> visit115 weight[7..79] time[61..65.6378] -> visit91
```

```
weight[15..87] time[72.3622..77] -> visit150 weight[27..99]
time[82.6132..96.5791] -> visit49 weight[43..115] time[97..107] -> visit170
weight[59..131] time[107.421..153.318] -> visit87 weight[75..147]
time[117.769..163.666] -> visit38 weight[80..152] time[127.972..173.869] ->
visit58 weight[92..164] time[138.016..183.913] -> visit158 weight[104..176]
time[148.059..193.956] -> visit118 weight[116..188] time[158.103..204] -> depot
weight[128..200] time[168.593..230]
vehicle16 :
 -> depot weight[0] time[0..54.929] -> visit62 weight[0..36]
time[0.409943..55.339] -> visit95 weight[7..43] time[61..65.6378] -> visit71
weight[15..51] time[72.3622..77] -> visit190 weight[27..63]
time[82.6132..95.9518] -> visit129 weight[43..79] time[97..106.373] -> visit130
weight[59..95] time[107.421..116.794] -> visit159 weight[75..111]
time[117.676..127.049] -> visit139 weight[92..128] time[127.71..137.083] ->
visit23 weight[109..145] time[138.637..148.009] -> visit144 weight[122..158]
time[149.054..158.426] -> visit133 weight[141..177] time[159.627..169] -> depot
weight[164..200] time[169.99..230]
vehicle17 :
 -> depot weight[0] time[0..52.7706] -> visit82 weight[0..64]
time[0.409943..53.1805] -> visit75 weight[7..71] time[61..63.4794] -> visit51
weight[15..79] time[72.3622..74.8416] -> visit179 weight[27..91]
time[82.5292..85.0086] -> visit28 weight[44..108] time[95..95.3838] -> visit149
weight[53..117] time[106.052..106.436] -> visit27 weight[69..133]
time[116.821..117.205] -> visit7 weight[74..138] time[126.858..127.242] ->
visit167 weight[79..143] time[136.895..137.279] -> visit181 weight[84..148]
time[147.319..147.703] -> visit164 weight[94..158] time[158.043..158.426] ->
visit153 weight[113..177] time[168.616..169] -> depot weight[136..200]
time[178.979..230]
vehicle18 :
 -> depot weight[0] time[0..52.3532] -> visit142 weight[0..46]
time[0.409943..52.7631] -> visit55 weight[7..53] time[61..63.062] -> visit131
weight[15..61] time[72.3622..74.4241] -> visit39 weight[27..73]
time[82.5292..84.5911] -> visit79 weight[44..90] time[92.5628..94.6248] ->
visit8 weight[61..107] time[102.938..105] -> visit65 weight[70..116]
time[113.274..118.349] -> visit125 weight[96..142] time[123.315..128.39] ->
visit106 weight[122..168] time[133.556..138.631] -> visit126 weight[125..171]
time[143.611..148.686] -> visit166 weight[128..174] time[153.666..158.741] ->
visit113 weight[131..177] time[163.925..169] -> depot weight[154..200]
time[174.288..230]
vehicle19 :
 -> depot weight[0] time[0..14.9558] -> visit66 weight[0..38]
time[0.409416..15.3652] -> visit186 weight[3..41] time[10.4645..25.4203] ->
visit6 weight[6..44] time[20.5196..35.4753] -> visit178 weight[9..47]
time[30.8888..45.8446] -> visit198 weight[21..59] time[40.9325..55.8883] ->
visit107 weight[33..71] time[51.1353..66.0911] -> visit191 weight[38..76]
time[67..76.3521] -> visit19 weight[50..88] time[77.167..86.5191] -> visit68
weight[67..105] time[95..96.8943] -> visit77 weight[76..114]
time[105.309..107.203] -> visit57 weight[78..116] time[115.343..117.237] ->
visit37 weight[80..118] time[125.377..127.271] -> visit176 weight[82..120]
time[135.632..137.527] -> visit156 weight[101..139] time[145.667..147.561] ->
visit104 weight[120..158] time[156.532..158.426] -> visit93 weight[139..177]
time[167.106..169] -> depot weight[162..200] time[177.469..230]
vehicle20 :
 -> depot weight[0] time[0..60.2912] -> visit162 weight[0..178]
time[0.409943..60.7011] -> visit15 weight[7..185] time[61..71] -> visit2
weight[15..193] time[71.2989..202] -> depot weight[22..200] time[81.7088..230]
vehicle21 :
 -> depot weight[0] time[0..53.0535] -> visit102 weight[0..24]
```

```
time[0.409943..53.4634] -> visit175 weight[7..31] time[61..63.7623] -> visit138
weight[15..39] time[71.8984..74.6606] -> visit78 weight[27..51]
time[81.9421..84.7043] -> visit98 weight[39..63] time[91.9857..94.748] ->
visit168 weight[51..75] time[102.238..105] -> visit18 weight[60..84]
time[112.49..118.106] -> visit5 weight[72..96] time[122.802..128.418] ->
visit185 weight[98..122] time[132.843..138.459] -> visit165 weight[124..148]
time[142.884..148.5] -> visit46 weight[150..174] time[153.124..158.741] ->
visit193 weight[153..177] time[163.384..169] -> depot weight[176..200]
time[173.747..230]
vehicle22 : Unused
vehicle23 : Unused
vehicle24 : Unused
vehicle25 : Unused
vehicle26 : Unused
vehicle27 : Unused
vehicle28 : Unused
vehicle29 : Unused
vehicle30 : Unused
vehicle31 : Unused
vehicle32 : Unused
vehicle33 : Unused
vehicle34 : Unused
vehicle35 : Unused
vehicle36 : Unused
vehicle37 : Unused
vehicle38 : Unused
vehicle39 : Unused
vehicle40 : Unused
vehicle41 : Unused
vehicle42 : Unused
vehicle43 : Unused
vehicle44 : Unused
vehicle45 : Unused
vehicle46 : Unused
vehicle47 : Unused
vehicle48 : Unused
vehicle49 : Unused
vehicle50 : Unused
Emptying vehicle20 ...
Improving solution
Emptying vehicle7 ...
Improving solution
***Solution after reducing active vehicles***
Number of fails              : 0
Number of choice points      : 0
Number of variables          : 5504
Number of constraints        : 917
Reversible stack (bytes)     : 578904
Solver heap (bytes)          : 2089592
Solver global heap (bytes)   : 1207824
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 35196
Total memory used (bytes)    : 3979928
Elapsed time since creation  : 0.061
Number of nodes              : 201
Number of visits             : 300
Number of vehicles           : 50
Number of dimensions         : 2
Number of accepted moves     : 238
===============
Cost         : 2072.93
Number of vehicles used : 16
Solution     :
Unperformed visits : None
vehicle1 : Unused
vehicle2 : Unused
vehicle3 : Unused
vehicle4 : Unused
vehicle5 :
 -> depot weight[0] time[0..5.70798] -> visit40 weight[0..4]
time[0.741903..6.44988] -> visit60 weight[9..13] time[10.7752..16.4832] ->
visit80 weight[18..22] time[20.8086..26.5166] -> visit100 weight[27..31]
time[30.8419..36.5499] -> visit120 weight[36..40] time[40.8753..46.5832] ->
visit140 weight[45..49] time[50.9086..56.6166] -> visit160 weight[54..58]
time[60.9419..66.6499] -> visit180 weight[63..67] time[70.9753..76.6832] ->
visit200 weight[72..76] time[81.0086..86.7166] -> visit20 weight[81..85]
time[91.0419..96.7499] -> visit9 weight[90..94] time[101.292..107] -> visit83
weight[106..110] time[111.542..118.483] -> visit103 weight[119..123]
time[121.576..128.516] -> visit123 weight[132..136] time[131.609..138.55] ->
visit3 weight[145..149] time[141.642..148.583] -> visit184 weight[158..162]
time[152.059..159] -> visit192 weight[177..181] time[162.394..205] -> depot
weight[196..200] time[172.702..230]
vehicle6 :
 -> depot weight[0] time[0..43.3555] -> visit30 weight[0..56]
time[0.589454..43.9449] -> visit119 weight[16..72] time[10.845..54.2005] ->
visit59 weight[33..89] time[20.8786..64.2341] -> visit31 weight[50..106]
time[67..74.4011] -> visit147 weight[62..118] time[77.261..84.662] -> visit127
weight[67..123] time[87.2981..94.6992] -> visit188 weight[72..128]
time[97.5989..105] -> visit17 weight[81..137] time[107.907..117.811] -> visit97
weight[83..139] time[117.942..127.845] -> visit157 weight[85..141]
time[127.976..137.879] -> visit196 weight[87..143] time[138.231..148.135] ->
visit4 weight[106..162] time[149.096..159] -> visit52 weight[125..181]
time[159.431..205] -> depot weight[144..200] time[169.739..230]
vehicle7 :
 -> depot weight[0] time[0..54.929] -> visit2 weight[0..109]
time[0.409943..55.339] -> visit15 weight[7..116] time[61..65.6378] -> visit11
weight[15..124] time[72.3622..77] -> visit10 weight[27..136]
time[82.6132..96.5791] -> visit109 weight[43..152] time[97..107] -> visit63
weight[59..168] time[107.25..194.749] -> visit32 weight[72..181]
time[117.502..205] -> depot weight[91..200] time[127.81..230]
vehicle8 :
 -> depot weight[0] time[0..5.64133] -> visit41 weight[0..5]
time[0.446525..6.08785] -> visit61 weight[10..15] time[10.4823..16.1236] ->
visit81 weight[20..25] time[20.5181..26.1594] -> visit101 weight[30..35]
time[30.5539..36.1952] -> visit121 weight[40..45] time[40.5897..46.231] ->
visit141 weight[50..55] time[50.6255..56.2668] -> visit161 weight[60..65]
time[60.6613..66.3026] -> visit21 weight[70..75] time[70.6971..76.3384] ->
visit163 weight[80..85] time[81.0037..86.645] -> visit183 weight[93..98]
time[91.037..96.6783] -> visit189 weight[106..111] time[101.287..106.928] ->
visit99 weight[122..127] time[111.964..117.605] -> visit137 weight[139..144]
time[122.639..128.28] -> visit117 weight[141..146] time[132.673..138.314] ->
visit25 weight[143..148] time[142.859..148.5] -> visit86 weight[169..174]
time[153.099..158.741] -> visit13 weight[172..177] time[163.359..169] -> depot
weight[195..200] time[173.722..230]
```

```
vehicle9 :
 -> depot weight[0] time[0..56.2644] -> visit182 weight[0..4]
time[0.409943..56.6744] -> visit35 weight[7..11] time[61..66.9732] -> visit34
weight[15..19] time[71.3385..77.3117] -> visit36 weight[35..39]
time[81.5948..87.568] -> visit48 weight[54..58] time[95..97.986] -> visit85
weight[63..67] time[105.336..108.322] -> visit105 weight[89..93]
time[115.377..118.363] -> visit145 weight[115..119] time[125.418..128.404] ->
visit45 weight[141..145] time[135.459..138.445] -> visit146 weight[167..171]
time[145.7..148.686] -> visit26 weight[170..174] time[155.755..158.741] ->
visit173 weight[173..177] time[166.014..169] -> depot weight[196..200]
time[176.377..230]
vehicle10 :
 -> depot weight[0] time[0..45.5461] -> visit112 weight[0..1]
time[0.308232..45.8543] -> visit132 weight[19..20] time[10.3427..55.8888] ->
visit152 weight[38..39] time[20.3772..65.9233] -> visit172 weight[57..58]
time[30.4117..75.9578] -> visit12 weight[76..77] time[40.4462..85.9923] ->
visit43 weight[95..96] time[50.6977..96.2438] -> visit169 weight[108..109]
time[97..106.494] -> visit110 weight[124..125] time[107.421..116.915] ->
visit199 weight[140..141] time[117.676..127.17] -> visit197 weight[157..158]
time[128.351..137.845] -> visit177 weight[159..160] time[138.385..147.879] ->
visit24 weight[161..162] time[149.506..159] -> visit72 weight[180..181]
time[159.841..205] -> depot weight[199..200] time[170.149..230]
vehicle11 :
 -> depot weight[0] time[0..32.7593] -> visit122 weight[0..1]
time[0.409943..33.1692] -> visit54 weight[7..8] time[10.8925..43.6518] ->
visit74 weight[27..28] time[20.9266..53.6859] -> visit195 weight[47..48]
time[61..64.0244] -> visit171 weight[55..56] time[72.3622..75.3866] -> visit187
weight[67..68] time[82.6231..85.6475] -> visit148 weight[72..73]
time[95..95.9483] -> visit29 weight[81..82] time[106.052..107] -> visit134
weight[97..98] time[117.579..127.497] -> visit114 weight[117..118]
time[127.614..137.532] -> visit94 weight[137..138] time[137.648..147.566] ->
visit44 weight[157..158] time[149..158.426] -> visit33 weight[176..177]
time[159.574..169] -> depot weight[199..200] time[169.937..230]
vehicle12 :
 -> depot weight[0] time[0..65.6019] -> visit50 weight[0..21]
time[0.589454..66.1914] -> visit151 weight[16..37] time[67..76.4425] -> visit67
weight[28..49] time[77.261..86.7034] -> visit128 weight[33..54]
time[95..97.0042] -> visit56 weight[42..63] time[105.418..107.422] -> visit136
weight[61..82] time[115.453..117.457] -> visit116 weight[80..101]
time[125.488..127.492] -> visit96 weight[99..120] time[135.522..137.527] ->
visit76 weight[118..139] time[145.557..147.561] -> visit84 weight[137..158]
time[156.422..158.426] -> visit73 weight[156..177] time[166.996..169] -> depot
weight[179..200] time[177.359..230]
vehicle13 :
 -> depot weight[0] time[0..54.2267] -> visit22 weight[0..19]
time[0.409943..54.6366] -> visit155 weight[7..26] time[61..64.9355] -> visit14
weight[15..34] time[71.3385..75.274] -> visit16 weight[35..54]
time[81.5948..85.5303] -> visit108 weight[54..73] time[95..95.9483] -> visit69
weight[63..82] time[106.052..107] -> visit154 weight[79..98]
time[117.579..127.497] -> visit194 weight[99..118] time[127.614..137.532] ->
visit174 weight[119..138] time[137.648..147.566] -> visit64 weight[139..158]
time[149..158.426] -> visit53 weight[158..177] time[159.574..169] -> depot
weight[181..200] time[169.937..230]
vehicle14 :
 -> depot weight[0] time[0..53.3156] -> visit162 weight[0..50]
time[0.409943..53.7256] -> visit135 weight[7..57] time[61..64.0244] -> visit111
weight[15..65] time[72.3622..75.3866] -> visit47 weight[27..77]
time[82.6231..85.6475] -> visit88 weight[32..82] time[95..95.9483] -> visit89
```

```
weight[41..91] time[106.052..107] -> visit70 weight[57..107]
time[116.473..117.865] -> visit90 weight[73..123] time[126.507..127.899] ->
visit1 weight[89..139] time[136.884..138.276] -> visit143 weight[99..149]
time[147.191..148.583] -> visit124 weight[112..162] time[157.608..159] ->
visit92 weight[131..181] time[167.942..205] -> depot weight[150..200]
time[178.251..230]
vehicle15 :
 -> depot weight[0] time[0..54.929] -> visit42 weight[0..72]
time[0.409943..55.339] -> visit115 weight[7..79] time[61..65.6378] -> visit91
weight[15..87] time[72.3622..77] -> visit150 weight[27..99]
time[82.6132..96.5791] -> visit49 weight[43..115] time[97..107] -> visit170
weight[59..131] time[107.421..153.318] -> visit87 weight[75..147]
time[117.769..163.666] -> visit38 weight[80..152] time[127.972..173.869] ->
visit58 weight[92..164] time[138.016..183.913] -> visit158 weight[104..176]
time[148.059..193.956] -> visit118 weight[116..188] time[158.103..204] -> depot
weight[128..200] time[168.593..230]
vehicle16 :
 -> depot weight[0] time[0..54.929] -> visit62 weight[0..36]
time[0.409943..55.339] -> visit95 weight[7..43] time[61..65.6378] -> visit71
weight[15..51] time[72.3622..77] -> visit190 weight[27..63]
time[82.6132..95.9518] -> visit129 weight[43..79] time[97..106.373] -> visit130
weight[59..95] time[107.421..116.794] -> visit159 weight[75..111]
time[117.676..127.049] -> visit139 weight[92..128] time[127.71..137.083] ->
visit23 weight[109..145] time[138.637..148.009] -> visit144 weight[122..158]
time[149.054..158.426] -> visit133 weight[141..177] time[159.627..169] -> depot
weight[164..200] time[169.99..230]
vehicle17 :
 -> depot weight[0] time[0..52.7706] -> visit82 weight[0..64]
time[0.409943..53.1805] -> visit75 weight[7..71] time[61..63.4794] -> visit51
weight[15..79] time[72.3622..74.8416] -> visit179 weight[27..91]
time[82.5292..85.0086] -> visit28 weight[44..108] time[95..95.3838] -> visit149
weight[53..117] time[106.052..106.436] -> visit27 weight[69..133]
time[116.821..117.205] -> visit7 weight[74..138] time[126.858..127.242] ->
visit167 weight[79..143] time[136.895..137.279] -> visit181 weight[84..148]
time[147.319..147.703] -> visit164 weight[94..158] time[158.043..158.426] ->
visit153 weight[113..177] time[168.616..169] -> depot weight[136..200]
time[178.979..230]
vehicle18 :
 -> depot weight[0] time[0..52.3532] -> visit142 weight[0..46]
time[0.409943..52.7631] -> visit55 weight[7..53] time[61..63.062] -> visit131
weight[15..61] time[72.3622..74.4241] -> visit39 weight[27..73]
time[82.5292..84.5911] -> visit79 weight[44..90] time[92.5628..94.6248] ->
visit8 weight[61..107] time[102.938..105] -> visit65 weight[70..116]
time[113.274..118.349] -> visit125 weight[96..142] time[123.315..128.39] ->
visit106 weight[122..168] time[133.556..138.631] -> visit126 weight[125..171]
time[143.611..148.686] -> visit166 weight[128..174] time[153.666..158.741] ->
visit113 weight[131..177] time[163.925..169] -> depot weight[154..200]
time[174.288..230]
vehicle19 :
 -> depot weight[0] time[0..14.9558] -> visit66 weight[0..38]
time[0.409416..15.3652] -> visit186 weight[3..41] time[10.4645..25.4203] ->
visit6 weight[6..44] time[20.5196..35.4753] -> visit178 weight[9..47]
time[30.8888..45.8446] -> visit198 weight[21..59] time[40.9325..55.8883] ->
visit107 weight[33..71] time[51.1353..66.0911] -> visit191 weight[38..76]
time[67..76.3521] -> visit19 weight[50..88] time[77.167..86.5191] -> visit68
weight[67..105] time[95..96.8943] -> visit77 weight[76..114]
time[105.309..107.203] -> visit57 weight[78..116] time[115.343..117.237] ->
visit37 weight[80..118] time[125.377..127.271] -> visit176 weight[82..120]
```

```
time[135.632..137.527] -> visit156 weight[101..139] time[145.667..147.561] ->
visit104 weight[120..158] time[156.532..158.426] -> visit93 weight[139..177]
time[167.106..169] -> depot weight[162..200] time[177.469..230]
vehicle20 : Unused
vehicle21 :
 -> depot weight[0] time[0..53.0535] -> visit102 weight[0..24]
time[0.409943..53.4634] -> visit175 weight[7..31] time[61..63.7623] -> visit138
weight[15..39] time[71.8984..74.6606] -> visit78 weight[27..51]
time[81.9421..84.7043] -> visit98 weight[39..63] time[91.9857..94.748] ->
visit168 weight[51..75] time[102.238..105] -> visit18 weight[60..84]
time[112.49..118.106] -> visit5 weight[72..96] time[122.802..128.418] ->
visit185 weight[98..122] time[132.843..138.459] -> visit165 weight[124..148]
time[142.884..148.5] -> visit46 weight[150..174] time[153.124..158.741] ->
visit193 weight[153..177] time[163.384..169] -> depot weight[176..200]
time[173.747..230]
vehicle22 : Unused
vehicle23 : Unused
vehicle24 : Unused
vehicle25 : Unused
vehicle26 : Unused
vehicle27 : Unused
vehicle28 : Unused
vehicle29 : Unused
vehicle30 : Unused
vehicle31 : Unused
vehicle32 : Unused
vehicle33 : Unused
vehicle34 : Unused
vehicle35 : Unused
vehicle36 : Unused
vehicle37 : Unused
vehicle38 : Unused
vehicle39 : Unused
vehicle40 : Unused
vehicle41 : Unused
vehicle42 : Unused
vehicle43 : Unused
vehicle44 : Unused
vehicle45 : Unused
vehicle46 : Unused
vehicle47 : Unused
vehicle48 : Unused
vehicle49 : Unused
vehicle50 : Unused

*/
```

C H A P T E R

5

Adding Visit Disjunctions

In this lesson, you will learn how to:

- add visit disjunctions
- modify an existing routing plan by adding or removing visits
- use the classes and functions `IloInsertVisit`, `IloRoutingSolution::remove`, `IloRoutingSolution::add`, `IloNHoodArray`, `IloConcatenate`, `IloVisit::performed`, `IloVisit::unperformed`, and `IloVisitArray`

You will learn how to model and solve a vehicle routing problem (VRP) where only one of a pair of visits can be performed or where there are alternate sites available for a visit. For example, a customer may have more than one site available to receive deliveries. Dispatcher provides *visit disjunctions* to model these types of problems.

You will also learn how to add and remove visits from an existing routing plan in a dynamic problem. In many situations, it may be necessary to take an order and add a visit to an existing routing plan. It may also be the case that an order is cancelled by a customer. Recomputing the plan from scratch would be far too time-consuming, so Dispatcher allows you to add or remove a visit by modifying an existing solution.

Describe

This problem presented here is similar to a standard VRP, except that each delivery to a customer is represented by two visits, only one of which can be performed. Each of these visits has a penalty cost for not being performed, and the total cost of the solution includes the penalty cost of the unperformed visits.

Step 1 Describe the problem

The first step is to write a natural language description of the problem.

The components of the routing model for this problem are the same as for a standard VRP: vehicles, customers, and a depot. However, each delivery to a customer is represented by two alternative visits.

The constraints in this problem are the same as those in a standard VRP: time windows, vehicle capacity, and visit quantities.

The objective is to minimize the total cost of the solution. This cost includes the penalty costs for not performing the visit in each pair that is unperformed.

Model

Once you have written a description of your problem, you can use Dispatcher classes to model it.

Step 2 Open the example file

Open the example file `YourDispatcherHome/examples/src/tutorial/disjunct_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 of this lesson, you will have completed the code for the problem and you will be able to compile and run it.

As in Chapter 2, *Modeling a Vehicle Routing Problem*, you create a `RoutingModel` class, which is used to model the problem.

Declare the RoutingModel class

The code for the declaration of the class `RoutingModel` is provided for you:

```
class RoutingModel {
  IloEnv              _env;
  IloModel            _mdl;
  IloDispatcherGraph  _graph;
  IloDimension1       _weight;
  IloDimension2       _time;
  IloDimension2       _distance;
  IloNHoodArray       _swapArray;

  void addDimensions();
  void loadGraphInformation (const char * arcFileName,
                                  const char* turnFileName);
  void lastMinuteGraphChanges ();
  void createIloNodes(const char * nodeFileName, const char* coordFileName);
  void createVehicles(const char * vehicleFileName);
  void createVisits(const char * visitsFileName);

public:
  RoutingModel(IloEnv env, int argc, char* argv[]);
  ~RoutingModel() {}
  IloEnv    getEnv() const { return _env; }
  IloModel  getModel() const { return _mdl; }
  IloNHoodArray getSwapArray() const {return _swapArray;}
  IloVisit  createAdditionalVisits (int argc, char* argv[]);
  void      removeVisit (IloVisit);
};
```

There are only a few differences between the `RoutingModel` class used in a standard VRP and the `RoutingModel` class used in this problem.

The dimensions in this problem are time, weight, and distance.

An instance of the class `IloDispatcherGraph` allows you create a graph representing a road network topology on which instances of `IloNode` can be positioned. `IloDispatcherGraph` computes the shortest paths between nodes for each vehicle. It is this shortest path that is used to create the dimensions `_time` and `_distance` in this problem. See “Define the addDimensions function” on page 124. Dispatcher’s graph functionality is introduced in Chapter 4, *Minimizing the Number of Vehicles*.

The function `getSwapArray` is used to access a neighborhood created to swap alternative visits and implement visit disjunctions. See “Define the createVisits function” on page 125. The member functions `createAdditionalVisits` and `removeVisit` are used to add and remove visits from the model. See “Define the createAdditionalVisits function” on page 128 and “Define the function removeVisit” on page 128.

Define the RoutingModel constructor

The constructor is defined as in Chapter 4, *Minimizing the Number of Vehicles*. If you do not specify input files, the defaults will be used. This constructor will be called from the `main` function. It calls the following functions: `addDimensions`, `loadGraphInformation`, `createIloNodes`, `createVehicles`, and `createVisits`.

The following code is provided for you:

```
RoutingModel::RoutingModel( IloEnv env,
                              int argc,
                              char* argv[]):
    _env(env), _mdl(env), _graph(env) {
    addDimensions();

    // Load dispatcher graph information and add instance-specific features.
    char * arcFileName;
    if(argc < 2)  arcFileName =
        (char*) "../../../examples/data/dispatcherGraphData/gridNetwork.csv";
    else          arcFileName = argv[1];
    char * turnFileName;
    if(argc < 3)  turnFileName =
        (char*) "../../../examples/data/dispatcherGraphData/turnData.csv";
    else          turnFileName = argv[2];
    loadGraphInformation (arcFileName, turnFileName);

    //create IloNodes and associate them to graph nodes
    char * nodeFileName;
    if(argc < 4)  nodeFileName =
        (char*) "../../../examples/data/vrp50/vrp50nodes.csv";
    else          nodeFileName = argv[3];
    char * nodeCoordsFile;
    if(argc < 5)  nodeCoordsFile =
        (char*) "../../../examples/data/dispatcherGraphData/coordTable.csv";
    else          nodeCoordsFile = argv[4];
    createIloNodes(nodeFileName, nodeCoordsFile);

    //create vehicles
    char * vehiclesFileName;
    if(argc < 6)  vehiclesFileName =
        (char*) "../../../examples/data/vrp50/vrp50vehicles.csv";
    else          vehiclesFileName = argv[5];
    createVehicles(vehiclesFileName);

    //create visits
    char * visitsFileName;
    if(argc < 7)  visitsFileName =
        (char*) "../../../examples/data/vrp50/vrp50visits.csv";
    else          visitsFileName = argv[6];
    createVisits(visitsFileName);
}
```

Define the loadGraphInformation function

The `loadGraphInformation` function is defined as in Chapter 4, *Minimizing the Number of Vehicles*. First, you build the road network graph using the function `loadGraphInformation`. Then you load the network topology from a csv file and create all necessary arcs and nodes. The member function `IloDispatcherGraph::loadArcDimensionDataFromFile` loads the arc cost information relating to the dimensions `_time` and `_distance`. The member function `IloDispatcherGraph::loadTurnDimensionDataFromFile` loads turn penalty information from the turn file. By default, all turns are allowed with no penalty. The `loadGraphInformation` function calls the `lastMinuteGraphChanges` function to allow for direct manipulation of the road network graph. The following code is provided for you:

```
void RoutingModel::loadGraphInformation ( const char* arcFileName,
                                          const char* turnFileName)    {
  _graph.createArcsFromFile (arcFileName);
  _graph.loadArcDimensionDataFromFile (arcFileName, _time);
  _graph.loadArcDimensionDataFromFile (arcFileName, _distance);
  _graph.loadTurnDimensionDataFromFile(turnFileName, _time);
  lastMinuteGraphChanges();
}
```

Define the lastMinuteGraphChanges function

The `lastMinuteGraphChanges` function is defined as in Chapter 4, *Minimizing the Number of Vehicles*. The following code is provided for you:

```
void RoutingModel::lastMinuteGraphChanges () {
  _graph.forbidArcUse(_graph.getArcByEnds(2785-1, 2785));
  _graph.forbidArcUse(_graph.getArcByEnds(2785+1, 2785));
  _graph.forbidArcUse(_graph.getArcByEnds(2785-56, 2785));
  _graph.setTurnPenalty(_graph.getArc(1323), _graph.getArc(1544), _time, 12);
}
```

Define the addDimensions function

The `addDimensions` function is defined as in *Chapter 4, Minimizing the Number of Vehicles*. The `_time` and `_distance` dimensions are computed using the shortest path from the road network graph. This code is provided for you:

```
void RoutingModel::addDimensions() {
  _weight =IloDimension1 (_env, "weight");
  _mdl.add(_weight);

  IloDistance SP_time =   IloGraphDistance (_graph);
  _time  =IloDimension2 (_env, SP_time, "time");
  _mdl.add(_time);

  IloDistance SP_distance =   IloGraphDistance (_graph);
  _distance  =IloDimension2 (_env, SP_distance, "distance");
  _mdl.add(_distance);
}
```

In this example, you could also use the predefined distance functions `IloEuclidean` or `IloManhattan` or define your own distance function.

Define the createIloNodes function

The `createIloNodes` function is defined as in Chapter 4, *Minimizing the Number of Vehicles*. You use IBM® ILOG® Concert Technology's csv functionality to input node data and node coordinate data from csv files. You use the member function `IloDispatcherGraph::associatebyCoordsInFile` to look up the coordinates of a given `IloNode` in a csv file and automatically associate it to the graph node with matching coordinates. Here is the complete code for defining the `createIloNodes` function:

```
void RoutingModel::createIloNodes(const char * nodeFileName,
                                  const char* coordFileName) {
  IloCsvReader csvNodeReader(_env, nodeFileName);
  IloCsvReader::LineIterator  it(csvNodeReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char * name = line.getStringByHeader("name");
    IloNode node(_env, line.getFloatByHeader("x"),
        line.getFloatByHeader("y"), 0, name);
    node.setKey(name);
    _graph.associateByCoordsInFile (node, coordFileName);
    ++it;
  }
  csvNodeReader.end();
}
```

Define the createVehicles function

The `createVehicles` function is defined as in Chapter 4, *Minimizing the Number of Vehicles*. You use csv reader functionality to input vehicle data from a csv file. The vehicles have start and end visits. You add side constraints that the vehicles must leave the depot after it opens and return to the depot before it closes. You set the capacities of the vehicles using `IloVehicle::setCapacity` and the dimension `_weight`. Using `IloVehicle::setCost`, the cost of each vehicle is set to be directly proportional to the dimensions `_time` and `_distance`. This code is provided for you:

```
 void RoutingModel::createVehicles(const char * vehicleFileName) {
  IloCsvReader csvVehicleReader(_env, vehicleFileName);
  IloCsvReader::LineIterator  it(csvVehicleReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char * namefirst = line.getStringByHeader("first");
    char * namelast = line.getStringByHeader("last");
    char * name = line.getStringByHeader("name");
    IloNum capacity = line.getFloatByHeader("capacity");
    IloNode node1 = IloNode::Find(_env, namefirst);
    IloNode node2 = IloNode::Find(_env, namelast);
    IloVisit first(node1, "depot");
    _mdl.add(first.getCumulVar(_weight) == 0);
    IloVisit last(node2, "depot");
    _mdl.add(last.getCumulVar(_distance) >= line.getFloatByHeader("open"));
    _mdl.add(last.getCumulVar(_distance) <= line.getFloatByHeader("close"));
    IloVehicle vehicle(first, last, name);
    vehicle.setCapacity(_weight, capacity);
    vehicle.setCost(_time, 1.0);
    vehicle.setCost(_distance, 1.0);
    _mdl.add(vehicle);
    ++it;
  }
  csvVehicleReader.end();
}
```

Define the createVisits function

Now you create the visits. In this problem, since each visit can be performed at either of two alternative sites, the visits are created in pairs. These pairs are themselves each a neighborhood, since they represent alternative moves that can be taken in a search for a solution—they can be swapped or exchanged with each other. The array of neighborhoods `_swapArray` is created using the constructor `IloNHoodArray`. This constructor creates an array of neighborhoods associated with environment `_env`. The size of the array is the value returned by `csvVisitReader.getNbOfItems` divided by 2. There are half as many neighborhood pairs as there are visits.

Step 3 Create the visit swap array

Add the following code after the comment `//Create the visit swap array`

```
void RoutingModel::createVisits( const char* visitsFileName) {
  IloCsvReader csvVisitReader(_env, visitsFileName);
  _swapArray = IloNHoodArray (_env, csvVisitReader.getNumberOfItems()/2);
  IloInt i=0;
  IloCsvReader::LineIterator  it(csvVisitReader);
  while(it.ok()){
```

Now the pairs of visits are created. Each visit of a pair is created in the same way. You use csv reader functionality to input visit data from a csv file. A visit must be associated to a node, its location. A visit also has a quantity—the amount of goods delivered to the location. A visit can have a minimum time and a maximum time during which it can be performed—a time window. Additionally, visits have a drop time—the amount time required to perform the visit. These side constraints are modeling using the dimensions `_time` and `_weight` and the delay, transit, and cumulative variables associated with the visit.

You use the member function `IloVisit::setPenaltyCost` to set the cost of not performing `visit1` to 20 units. This allows the visit to not be performed, since only one visit of each pair will be performed.

Step 4 Create the first visit of the pair

Add the following code after the comment `// Create the first visit of the pair`

```
    IloCsvLine line1 = *it;
    //read visit data from files
    char * visitName1 = line1.getStringByHeader("name");
    char * nodeName1 = line1.getStringByHeader("node");
    IloNum quantity1 = line1.getFloatByHeader("quantity");
    IloNum minTime1 = line1.getFloatByHeader("minTime");
    IloNum maxTime1 = line1.getFloatByHeader("maxTime");
    IloNode node1 = IloNode::Find(_env, nodeName1);
    IloVisit visit1(node1, visitName1);
    _mdl.add(visit1.getTransitVar(_weight) == quantity1);
    visit1.setPenaltyCost(20);
    _mdl.add(minTime1 <= visit1.getCumulVar(_time) <= maxTime1);
    _mdl.add(visit1);
    ++it;
```

The second visit of the pair is created in the same way as the first, except that it has a different penalty cost. You use the member function `IloVisit::setPenaltyCost` to set the cost of not performing `visit2` to 10 units. This means that the solution will be more likely to include the `visit1` of each pair, rather than the `visit2` of each pair, since the penalty for not performing `visit1` of each pair is higher than the penalty for not performing `visit2`. The following code is provided for you:

```
    IloCsvLine line2 = *it;
    char * visitName2 = line2.getStringByHeader("name");
    char * nodeName2 = line2.getStringByHeader("node");
    IloNum quantity2 = line2.getFloatByHeader("quantity");
    IloNum minTime2 = line2.getFloatByHeader("minTime");
    IloNum maxTime2 = line2.getFloatByHeader("maxTime");
    IloNode node2 = IloNode::Find(_env, nodeName2);
    IloVisit visit2(node2, visitName2);
    _mdl.add(visit2.getTransitVar(_weight) == quantity2);
    visit2.setPenaltyCost(10);
    _mdl.add(minTime2 <= visit2.getCumulVar(_time) <= maxTime2);
    _mdl.add(visit2);
    ++it;
```

Once both visits are created, you post a constraint that ensures that only one of the two visits is performed. You use the member function `IloVisit::unperformed` to add this constraint.

Step 5 Add the visit disjunction constraint

Add the following code after the comment
`//Add the visit disjunction constraint`

```
    _mdl.add(visit1.unperformed() + visit2.unperformed() == 1);
```

You then create an array of visits that includes `visit1` and `visit2`. You create a neighborhood using the function `IloSwapPerform` that will modify the solution by exchanging the performed visit of the pair with the unperformed visit of the pair. This is used in the solution improvement phase.

Step 6 Create the swap neighborhood

Add the following code after the comment `//Create the swap neighborhood`

```
    IloVisitArray visits(_env, 2);
    visits[0] = visit1; visits[1] = visit2;
    _swapArray[i] = IloSwapPerform(_env, visits);
    ++i;
  }
  csvVisitReader.end();
}
```

Define the createAdditionalVisits function

To create a new visit in the model, you create an instance of `IloNode` corresponding to the customer location. You use the member function `IloDispatcherGraph::associateByCoordsInFile` to look up the coordinates of this `IloNode` in a csv file and automatically associate it to the graph node with matching coordinates. You then create an instance of `IloVisit` located at the customer node. This visit has a delay time of 10 units, a quantity of 12 units, and a time window. You use the member function `IloModel::add(IloVisit)` to add the visit from the model.

Step 7 Create an additional visit

Add the following code after the comment `//Create an additional visit`

```
IloVisit RoutingModel::createAdditionalVisits (int argc, char * argv[]) {
  char * nodeCoordsFile;
  if(argc < 5)  nodeCoordsFile =
      (char*) "../../../examples/data/dispatcherGraphData/coordTable.csv";
  else          nodeCoordsFile = argv[4];
  IloNode customer(_env, 43, 63);
  _graph.associateByCoordsInFile (customer, nodeCoordsFile);
  IloVisit visit(customer, "Extra visit");
  _mdl.add(visit.getDelayVar(_time) == 10);
  _mdl.add(visit.getTransitVar(_weight) == 12);
  _mdl.add(70 <= visit.getCumulVar(_time) <= 120);
  _mdl.add(visit);
  return visit;
}
```

Define the function removeVisit

You use the member function `IloModel::remove(IloVisit)` to remove a visit from the model.

Step 8 Remove the visit from the model

Add the following code after the comment `//Remove the visit from the model`

```
void RoutingModel::removeVisit (IloVisit visit) {
  _mdl.remove(visit);
}
```

Solve

You search for a solution using the two-phase approach shown in Chapter 3, *Solving a Vehicle Routing Problem*. The only difference is that you use the swap neighborhoods you created from the visit pairs. You also dynamically modify the solution by adding and removing a visit from an existing routing plan.

As in Chapter 3, *Solving a Vehicle Routing Problem*, you create the `RoutingSolver` class, which is used to solve the vehicle routing problem.

Declare the RoutingSolver class

The code for the declaration of the class `RoutingSolver` is provided for you:

```
class RoutingSolver {
  IloEnv             _env;
  IloModel           _mdl;
  IloSolver          _solver;
  IloDispatcher      _dispatcher;
  IloRoutingSolution  _solution;
  IloGoal            _instantiateCost;
  IloGoal            _generalGoal;
  IloGoal            _restoreSolution;

public:
  RoutingSolver(RoutingModel mdl);
  ~RoutingSolver() {}
  IloBool findFirstSolution ();
  void improveWithNhood (IloNHoodArray swap);
  IloBool addNewVisit (IloVisit visit);
  IloBool removeVisitAndResolve (IloVisit visit);
  void printInformation(const char* =0) const;
};
```

There are only a few differences between the `RoutingSolver` class used in a standard VRP and the `RoutingSolver` class used in this problem. There are two additional member functions: `addNewVisit` and `removeVisitAndResolve`. These member functions are explained in the following sections.

Define the RoutingSolver constructor

The `RoutingSolver` constructor is defined as in Chapter 3, *Solving a Vehicle Routing Problem*. It takes an instance of `RoutingModel` as a parameter. The environment, model, solver, dispatcher, and solution are initialized. The goal to instantiate cost is created using the function `IloDichotomize`. You use the predefined first solution generation heuristic `IloSavingsGenerate` to create the first solution. The goal to restore the solution is created using IloRestoreSolution. This constructor will be called from the main function. This code is provided for you:

```
 RoutingSolver::RoutingSolver(RoutingModel mdl):
  _env(mdl.getEnv()),
  _mdl(mdl.getModel()),
  _solver(mdl.getModel()),
  _dispatcher(_solver),
  _solution(mdl.getModel()){
    _instantiateCost =
        IloDichotomize(_env, _dispatcher.getCostVar(), IloFalse);
    _generalGoal = IloSavingsGenerate(_env) && _instantiateCost;
    _restoreSolution = IloRestoreSolution(_env, _solution);
  }
```

Define the findFirstSolution function

Now, you create the function that searches for the first solution. This function is the same as the one you created in Chapter 3, *Solving a Vehicle Routing Problem*. You search for a solution using _generalGoal and store the solution and its cost using the member function IloRoutingSolution::store. This code is provided for you:

```
IloBool RoutingSolver::findFirstSolution() {
  if (!_solver.solve(_generalGoal)) {
    _solver.error() << "Infeasible Routing Plan" << endl;
    return IloFalse;
  }
  _solution.store(_solver);
  return IloTrue;
}
```

Define the improveWithNHood function

After you have found a first solution, you create the neighborhoods you will use in the solution improvement phase. You use the constructor IloNHoodArray to create an array of six neighborhoods called nhoodArray. Five of these neighborhoods are the predefined neighborhoods IloRelocate, IloExchange, IloCross, IloTwoOpt, and IloOrOpt. The neighborhood nhoodArray[2] is created using the function IloConcatenate to join together the array of swap neighborhoods you created in the section “Define the createVisits function” on page 125. You also use the function IloConcatenate to join together all the neighborhoods in nHoodArray and create nhood.

For more information about how predefined neighborhoods work, see Appendix B, *Predefined Neighborhoods*.

Step 9 Create the neighborhoods

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

```
void RoutingSolver::improveWithNhood(IloNHoodArray swap) {
  IloNHoodArray nhoodArray(_env, 6);
  nhoodArray[0] = IloTwoOpt(_env);
  nhoodArray[1] = IloOrOpt(_env);
  nhoodArray[2] = IloConcatenate(_env, swap);
  nhoodArray[3] = IloExchange(_env);
  nhoodArray[4] = IloRelocate(_env);
  nhoodArray[5] = IloCross(_env);
  IloNHood nhood = IloConcatenate(_env, nhoodArray);
```

The rest of the function is defined as in Chapter 3, *Solving a Vehicle Routing Problem*. You use the function `IloSingleMove` to return a goal that makes a single local move as defined by a neighborhood and a search heuristic. The following code is provided for you:

```
  _solver.out() << "Improving solution" << endl;
  IloGoal improve = IloSingleMove(_env,
                                  _solution,
                                  nhood,
                                  IloImprove(_env),
                                  _instantiateCost);
  while (_solver.solve(improve)) {
  }
  _solver.solve(_restoreSolution);
}
```

Define the addNewVisit function

Once new visits have been added to an existing routing plan using the function `createAdditionalVisits`, they can be inserted into the existing solution using the function `addNewVisit`. This function uses `IloRoutingSolution::add` to directly add the new visit to the solution. When it is added, its saved state is unperformed. To make the visit performed, you create a goal using `IloInsertVisit`, which eliminates the need to recompute the routing plan from scratch. If the visit cannot be inserted in the solution, the goal fails. If the visit is inserted, the new solution is stored.

Step 10 Add a new visit

Add the following code after the comment `//Add a new visit`

```
IloBool RoutingSolver::addNewVisit (IloVisit visit) {
  IloGoal insert = IloInsertVisit(_env, visit, _solution, _instantiateCost);
  if (!_solver.solve(insert)) {
    _solver.out() << "Cannot insert new visit in solution" << endl;
    return IloFalse;
  }
  _solution.add(visit);
  _solution.store(_solver);
  return IloTrue;
}
```

Define the removeVisitAndResolve function

To remove that same new visit from the solution, you use the function `removeVisitAndResolve`. This function uses `IloRoutingSolution::remove` to directly remove the new visit to the solution. You then restore the solution.

Step 11 Remove the visit and resolve

Add the following code after the comment `//Remove the visit and resolve`

```
IloBool RoutingSolver::removeVisitAndResolve (IloVisit visit) {
  _solution.remove(visit);
  if (_solver.solve(_restoreSolution))
      return IloTrue;
  else return IloFalse;
}
```

Define the printInformation function

The `printInformation` function is the same as in Chapter 3, *Solving a Vehicle Routing Problem*. This code is provided for you:

```
void RoutingSolver::printInformation(const char* heading) const {
  if(heading)
      _solver.out()<<heading<<endl;
  _solver.printInformation();
  _dispatcher.printInformation();
  _solver.out() << "===============" << endl
    << "Cost         : " << _dispatcher.getTotalCost() << endl
    << "Number of vehicles used : "
    << _dispatcher.getNumberOfVehiclesUsed() << endl
    << "Solution     : " << endl
    << _dispatcher << endl;
}
```

Define the main function

After you finish creating the `RoutingModel` and `RoutingSolver` classes and the `printInformation` function, you use them in the `main` function. You can use command line syntax to pass the names of input files to the model. If you do not specify input files, the defaults will be used. In the `main` function, you first create an environment. Then you create an instance of the `RoutingModel` class, which takes the environment and input files as parameters. You create an instance of the `RoutingSolver` class. This takes one parameter, the model. You use an `if` loop to find a solution. If Solver finds a first solution, you improve the solution. Then, you use the function `createAdditionalVisits` to add a visit to the model and the function `addNewVisit` to dynamically add the visit to the routing plan without recomputing the solution from scratch. You print this solution and then remove the visit. The following code is provided for you:

```
int main(int argc, char * argv[]) {
  IloEnv env;
  try {
    RoutingModel mdl(env, argc, argv);
    RoutingSolver solver(mdl);
    if (solver.findFirstSolution()) {
      solver.printInformation("***First Solution***");
      solver.improveWithNhood(mdl.getSwapArray());
      solver.printInformation("***Solution after improvements with nhood***");
      IloVisit visit = mdl.createAdditionalVisits (argc, argv);
      if (solver.addNewVisit (visit)) {
        solver.printInformation("***Solution including new visit***");
        mdl.removeVisit(visit);
        if (solver.removeVisitAndResolve (visit)) {
          solver.printInformation("***Solution after removing visit***");
        }
      }
    }
  } catch(IloException& ex) {
    cerr << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Step 12 Compile and run the program

Compile and run the program. You will get results that show the routing plan and information for the first solution and the improved solution. You will also get results that show how the solution changes when you add an additional visit and then remove it.

First solution information

The first solution phase finds a solution using 2 vehicles with a total cost of 1012.11 units:

```
***First Solution***
Number of fails              : 245
Number of choice points      : 1057
Number of variables          : 2064
Number of constraints        : 348
Reversible stack (bytes)     : 156804
Solver heap (bytes)          : 892720
Solver global heap (bytes)   : 132324
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11160
Total memory used (bytes)    : 1261420
Elapsed time since creation  : 1.593
Number of nodes              : 51
Number of visits             : 70
Number of vehicles           : 10
Number of dimensions         : 3
Number of accepted moves     : 0
===============
Cost         : 1012.11
Number of vehicles used : 2
```

Improved Solution Information

The solution improvement phase finds a solution using 2 vehicles with a total cost of 716.945 units after making 24 cost-decreasing moves:

```
***Solution after improvements with nhood***
Number of fails              : 0
Number of choice points      : 0
Number of variables          : 2064
Number of constraints        : 344
Reversible stack (bytes)     : 156804
Solver heap (bytes)          : 892720
Solver global heap (bytes)   : 148404
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11160
Total memory used (bytes)    : 1277500
Elapsed time since creation  : 0.01
Number of nodes              : 51
Number of visits             : 70
Number of vehicles           : 10
Number of dimensions         : 3
Number of accepted moves     : 24
===============
Cost         : 716.945
Number of vehicles used : 2
```

Solution information including new visit

After including a new visit, the total cost is now 763.538:

```
Number of fails              : 1
Number of choice points      : 1049
Number of variables          : 2091
Number of constraints        : 345
Reversible stack (bytes)     : 156804
Solver heap (bytes)          : 892720
Solver global heap (bytes)   : 559036
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11160
Total memory used (bytes)    : 1688132
Elapsed time since creation  : 1.462
Number of nodes              : 52
Number of visits             : 71
Number of vehicles           : 10
Number of dimensions         : 3
Number of accepted moves     : 24
===============
Cost          : 763.538
Number of vehicles used : 2
```

Solution information after removing visit

After removing the visit, the routing plan again has a total cost of 716.945:

```
***Solution after removing visit***
Number of fails              : 0
Number of choice points      : 0
Number of variables          : 2068
Number of constraints        : 345
Reversible stack (bytes)     : 156804
Solver heap (bytes)          : 892720
Solver global heap (bytes)   : 559036
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11160
Total memory used (bytes)    : 1688132
Elapsed time since creation  : 1.482
Number of nodes              : 52
Number of visits             : 70
Number of vehicles           : 10
Number of dimensions         : 3
Number of accepted moves     : 24
===============
Cost          : 716.945
Number of vehicles used : 2
```

The complete program and output are listed in “Complete Program” on page 137. You can also view it online in the `YourDispatcherHome/examples/src/disjunct.cpp` file.

Review Exercises

1. How is the visit penalty cost used in this problem?
2. How do you add a visit without recomputing the routing plan from scratch?

Suggested Answers

Exercise 1

How is the visit penalty cost used in this problem?

Suggested Answer

You use the member function `IloVisit::setPenaltyCost` to set the cost of not performing `visit1` to 20 units. This allows the visit to not be performed, since only one visit of each pair will be performed. You use the member function `IloVisit::setPenaltyCost` to set the cost of not performing `visit2` to 10 units. This means that the solution will be more likely to include the `visit1` of each pair, rather than the `visit2` of each pair, since the penalty for not performing `visit1` of each pair is higher than the penalty for not performing `visit2`.

Exercise 2

How do you add a visit without recomputing the routing plan from scratch?

Suggested Answer

You use `IloRoutingSolution::add` to directly add the new visit to the solution. When it is added, its saved state is unperformed. To make the visit performed, you create a goal using `IloInsertVisit`, which eliminates the need to recompute the routing plan from scratch. If the visit cannot be inserted in the solution, the goal fails. If the visit is inserted, the new solution is stored.

Complete Program

The complete program follows. You can also view it online in the `YourDispatcherHome/examples/src/disjunct.cpp` file.

```
// -------------------------------------------------------------- -*- C++ -*-
// File: examples/src/disjunct.cpp
// --------------------------------------------------------------------------
```

```
#include <ildispat/ilodispatcher.h>

ILOSTLBEGIN
////////////////////////////////////////////////////////////////////////////////
// Modeling
class RoutingModel {
  IloEnv             _env;
  IloModel           _mdl;
  IloDispatcherGraph  _graph;
  IloDimension1      _weight;
  IloDimension2      _time;
  IloDimension2      _distance;
  IloNHoodArray      _swapArray;

  void addDimensions();
  void loadGraphInformation (const char * arcFileName,
                               const char* turnFileName);
  void lastMinuteGraphChanges ();
  void createIloNodes(const char * nodeFileName, const char* coordFileName);
  void createVehicles(const char * vehicleFileName);
  void createVisits(const char * visitsFileName);

public:
  RoutingModel(IloEnv env, int argc, char* argv[]);
  ~RoutingModel() {}
  IloEnv    getEnv() const { return _env; }
  IloModel  getModel() const { return _mdl; }
  IloNHoodArray getSwapArray() const {return _swapArray;}
  IloVisit  createAdditionalVisits (int argc, char* argv[]);
  void      removeVisit (IloVisit);
};

RoutingModel::RoutingModel( IloEnv env,
                              int argc,
                              char* argv[]):
    _env(env), _mdl(env), _graph(env) {
    addDimensions();

    // Load dispatcher graph information and add instance-specific features.
    char * arcFileName;
    if(argc < 2)  arcFileName =
        (char*) "../../../examples/data/dispatcherGraphData/gridNetwork.csv";
    else          arcFileName = argv[1];
    char * turnFileName;
    if(argc < 3)  turnFileName =
        (char*) "../../../examples/data/dispatcherGraphData/turnData.csv";
    else          turnFileName = argv[2];
    loadGraphInformation (arcFileName, turnFileName);

    //create IloNodes and associate them to graph nodes
    char * nodeFileName;
    if(argc < 4)  nodeFileName =
        (char*) "../../../examples/data/vrp50/vrp50nodes.csv";
    else          nodeFileName = argv[3];
    char * nodeCoordsFile;
    if(argc < 5)  nodeCoordsFile =
        (char*) "../../../examples/data/dispatcherGraphData/coordTable.csv";
    else          nodeCoordsFile = argv[4];
    createIloNodes(nodeFileName, nodeCoordsFile);

    //create vehicles
    char * vehiclesFileName;
    if(argc < 6)  vehiclesFileName =
        (char*) “../../../examples/data/vrp50/vrp50vehicles.csv”;
    else          vehiclesFileName = argv[5];
    createVehicles(vehiclesFileName);

    //create visits
    char * visitsFileName;
    if(argc < 7)  visitsFileName =
        (char*) “../../../examples/data/vrp50/vrp50visits.csv”;
    else          visitsFileName = argv[6];
    createVisits(visitsFileName);
}

// create distance functions for dimensions, add dimensions to model
void RoutingModel::addDimensions() {
  _weight =IloDimension1 (_env, “weight”);
  _mdl.add(_weight);

  IloDistance SP_time =   IloGraphDistance (_graph);
  _time  =IloDimension2 (_env, SP_time, “time”);
  _mdl.add(_time);

  IloDistance SP_distance =   IloGraphDistance (_graph);
  _distance  =IloDimension2 (_env, SP_distance, “distance”);
  _mdl.add(_distance);
}

// load network topology and travel costs from files.
void RoutingModel::loadGraphInformation ( const char* arcFileName,
                                          const char* turnFileName)    {
  _graph.createArcsFromFile (arcFileName);
  _graph.loadArcDimensionDataFromFile (arcFileName, _time);
  _graph.loadArcDimensionDataFromFile (arcFileName, _distance);
  _graph.loadTurnDimensionDataFromFile(turnFileName, _time);
  lastMinuteGraphChanges();
}

// Make modifications to network conditions based on latest information.
void RoutingModel::lastMinuteGraphChanges () {
  _graph.forbidArcUse(_graph.getArcByEnds(2785-1, 2785));
  _graph.forbidArcUse(_graph.getArcByEnds(2785+1, 2785));
  _graph.forbidArcUse(_graph.getArcByEnds(2785-56, 2785));
  _graph.setTurnPenalty(_graph.getArc(1323), _graph.getArc(1544), _time, 12);
}

//create IloNodes
void RoutingModel::createIloNodes(const char * nodeFileName,
                                  const char* coordFileName) {
  IloCsvReader csvNodeReader(_env, nodeFileName);
  IloCsvReader::LineIterator  it(csvNodeReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char * name = line.getStringByHeader(“name”);
    IloNode node(_env, line.getFloatByHeader(“x”),
        line.getFloatByHeader(“y”), 0, name);
    node.setKey(name);
    _graph.associateByCoordsInFile (node, coordFileName);
    ++it;
  }
  csvNodeReader.end();
}

//create vehicles
void RoutingModel::createVehicles(const char * vehicleFileName) {
  IloCsvReader csvVehicleReader(_env, vehicleFileName);
  IloCsvReader::LineIterator  it(csvVehicleReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char * namefirst = line.getStringByHeader(“first”);
    char * namelast = line.getStringByHeader(“last”);
    char * name = line.getStringByHeader(“name”);
    IloNum capacity = line.getFloatByHeader(“capacity”);
    IloNode node1 = IloNode::Find(_env, namefirst);
    IloNode node2 = IloNode::Find(_env, namelast);
    IloVisit first(node1, “depot”);
    _mdl.add(first.getCumulVar(_weight) == 0);
    IloVisit last(node2, “depot”);
    _mdl.add(last.getCumulVar(_distance) >= line.getFloatByHeader(“open”));
    _mdl.add(last.getCumulVar(_distance) <= line.getFloatByHeader(“close”));
    IloVehicle vehicle(first, last, name);
    vehicle.setCapacity(_weight, capacity);
    vehicle.setCost(_time, 1.0);
    vehicle.setCost(_distance, 1.0);
    _mdl.add(vehicle);
    ++it;
  }
  csvVehicleReader.end();
}

//create visits
void RoutingModel::createVisits( const char* visitsFileName) {
  IloCsvReader csvVisitReader(_env, visitsFileName);
  _swapArray = IloNHoodArray (_env, csvVisitReader.getNumberOfItems()/2);
  IloInt i=0;
  IloCsvReader::LineIterator  it(csvVisitReader);
  while(it.ok()){

    IloCsvLine line1 = *it;
    //read visit data from files
    char * visitName1 = line1.getStringByHeader(“name”);
    char * nodeName1 = line1.getStringByHeader(“node”);
    IloNum quantity1 = line1.getFloatByHeader(“quantity”);
    IloNum minTime1 = line1.getFloatByHeader(“minTime”);
    IloNum maxTime1 = line1.getFloatByHeader(“maxTime”);
    IloNode node1 = IloNode::Find(_env, nodeName1);
    IloVisit visit1(node1, visitName1);
    _mdl.add(visit1.getTransitVar(_weight) == quantity1);
    visit1.setPenaltyCost(20);
    _mdl.add(minTime1 <= visit1.getCumulVar(_time) <= maxTime1);
    _mdl.add(visit1);
    ++it;
    IloCsvLine line2 = *it;
    char * visitName2 = line2.getStringByHeader(“name”);
    char * nodeName2 = line2.getStringByHeader(“node”);
    IloNum quantity2 = line2.getFloatByHeader(“quantity”);
    IloNum minTime2 = line2.getFloatByHeader(“minTime”);
    IloNum maxTime2 = line2.getFloatByHeader(“maxTime”);
    IloNode node2 = IloNode::Find(_env, nodeName2);
    IloVisit visit2(node2, visitName2);
    _mdl.add(visit2.getTransitVar(_weight) == quantity2);
    visit2.setPenaltyCost(10);
    _mdl.add(minTime2 <= visit2.getCumulVar(_time) <= maxTime2);
    _mdl.add(visit2);
    ++it;

    _mdl.add(visit1.unperformed() + visit2.unperformed() == 1);

    IloVisitArray visits(_env, 2);
    visits[0] = visit1; visits[1] = visit2;
    _swapArray[i] = IloSwapPerform(_env, visits);
    ++i;
  }
  csvVisitReader.end();
}

 // Modify problem set-up by adding a new visit.
 // Associate new visit to graph node.
IloVisit RoutingModel::createAdditionalVisits (int argc, char * argv[]) {
  char * nodeCoordsFile;
  if(argc < 5)  nodeCoordsFile =
      (char*) “../../../examples/data/dispatcherGraphData/coordTable.csv”;
  else          nodeCoordsFile = argv[4];
  IloNode customer(_env, 43, 63);
  _graph.associateByCoordsInFile (customer, nodeCoordsFile);
  IloVisit visit(customer, “Extra visit”);
  _mdl.add(visit.getDelayVar(_time) == 10);
  _mdl.add(visit.getTransitVar(_weight) == 12);
  _mdl.add(70 <= visit.getCumulVar(_time) <= 120);
  _mdl.add(visit);
  return visit;
}

void RoutingModel::removeVisit (IloVisit visit) {
  _mdl.remove(visit);
}

///////////////////////////////////////////////////////////////////////////////
// Solving
class RoutingSolver {
  IloEnv              _env;
  IloModel            _mdl;
  IloSolver           _solver;
  IloDispatcher       _dispatcher;
  IloRoutingSolution  _solution;
  IloGoal             _instantiateCost;
  IloGoal             _generalGoal;
  IloGoal             _restoreSolution;

public:
  RoutingSolver(RoutingModel mdl);
  ~RoutingSolver() {}
  IloBool findFirstSolution ();
  void improveWithNhood (IloNHoodArray swap);
  IloBool addNewVisit (IloVisit visit);
  IloBool removeVisitAndResolve (IloVisit visit);
  void printInformation(const char* =0) const;
};

RoutingSolver::RoutingSolver(RoutingModel mdl):
  _env(mdl.getEnv()),
  _mdl(mdl.getModel()),
  _solver(mdl.getModel()),
  _dispatcher(_solver),
  _solution(mdl.getModel()){
    _instantiateCost =
        IloDichotomize(_env, _dispatcher.getCostVar(), IloFalse);
    _generalGoal = IloSavingsGenerate(_env) && _instantiateCost;
    _restoreSolution = IloRestoreSolution(_env, _solution);
  }

// Solving : find first solution
IloBool RoutingSolver::findFirstSolution() {
  if (!_solver.solve(_generalGoal)) {
    _solver.error() << “Infeasible Routing Plan” << endl;
    return IloFalse;
  }
  _solution.store(_solver);
  return IloTrue;
}

//Improve solution using nhood
void RoutingSolver::improveWithNhood(IloNHoodArray swap) {
  IloNHoodArray nhoodArray(_env, 6);
  nhoodArray[0] = IloTwoOpt(_env);
  nhoodArray[1] = IloOrOpt(_env);
  nhoodArray[2] = IloConcatenate(_env, swap);
  nhoodArray[3] = IloExchange(_env);
  nhoodArray[4] = IloRelocate(_env);
  nhoodArray[5] = IloCross(_env);
  IloNHood nhood = IloConcatenate(_env, nhoodArray);

  _solver.out() << “Improving solution” << endl;
  IloGoal improve = IloSingleMove(_env,
                                  _solution,
                                  nhood,
                                  IloImprove(_env),
                                  _instantiateCost);
  while (_solver.solve(improve)) {
  }
  _solver.solve(_restoreSolution);
}

IloBool RoutingSolver::addNewVisit (IloVisit visit) {
  IloGoal insert = IloInsertVisit(_env, visit, _solution, _instantiateCost);
  if (!_solver.solve(insert)) {
    _solver.out() << “Cannot insert new visit in solution” << endl;
    return IloFalse;
  }
  _solution.add(visit);
  _solution.store(_solver);
  return IloTrue;
}

IloBool RoutingSolver::removeVisitAndResolve (IloVisit visit) {
  _solution.remove(visit);
  if (_solver.solve(_restoreSolution))
      return IloTrue;
  else return IloFalse;
}

// Display Dispatcher information
void RoutingSolver::printInformation(const char* heading) const {
  if(heading)
      _solver.out()<<heading<<endl;
  _solver.printInformation();
  _dispatcher.printInformation();
  _solver.out() << "===============" << endl
    << "Cost         : " << _dispatcher.getTotalCost() << endl
    << "Number of vehicles used : "
    << _dispatcher.getNumberOfVehiclesUsed() << endl
    << "Solution     : " << endl
    << _dispatcher << endl;
}

////////////////////////////////////////////////////////////////////////////////
int main(int argc, char * argv[]) {
  IloEnv env;
  try {
    RoutingModel mdl(env, argc, argv);
    RoutingSolver solver(mdl);
    if (solver.findFirstSolution()) {
      solver.printInformation("***First Solution***");
      solver.improveWithNhood(mdl.getSwapArray());
      solver.printInformation("***Solution after improvements with nhood***");
      IloVisit visit = mdl.createAdditionalVisits (argc, argv);
      if (solver.addNewVisit (visit)) {
        solver.printInformation("***Solution including new visit***");
        mdl.removeVisit(visit);
        if (solver.removeVisitAndResolve (visit)) {
          solver.printInformation("***Solution after removing visit***");
        }
      }
    }
  } catch(IloException& ex) {
    cerr << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Complete Output

```
/**
```

```
***First Solution***
Number of fails              : 245
Number of choice points      : 1057
Number of variables          : 2064
Number of constraints        : 348
Reversible stack (bytes)     : 156804
Solver heap (bytes)          : 892720
Solver global heap (bytes)   : 132324
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11160
Total memory used (bytes)    : 1261420
Elapsed time since creation  : 1.593
Number of nodes              : 51
Number of visits             : 70
Number of vehicles           : 10
Number of dimensions         : 3
Number of accepted moves     : 0
===============
Cost          : 1012.11
Number of vehicles used : 2
Solution      :
Unperformed visits : visit1 visit3 visit5 visit7 visit9 visit12 visit13 visit15
visit18 visit20 visit21 visit23 visit26 visit27 visit29 visit31 visit33 visit35
visit37 visit40 visit41 visit44 visit45 visit48 visit50
vehicle1 :
 -> depot weight[0] time[0..103.367] distance[0..2] -> visit34 weight[0..1]
time[0.741619..104.109] distance[38..40] -> visit19 weight[14..15]
time[0.86081..104.228] distance[45..47] -> visit47 weight[31..32]
time[1.04517..104.412] distance[56..58] -> visit36 weight[58..59]
time[1.21219..104.58] distance[66..68] -> visit46 weight[63..64]
time[1.41234..104.78] distance[78..80] -> visit8 weight[64..65] time[95..105]
distance[91..93] -> visit17 weight[73..74] time[95.3086..175.83]
distance[109..111] -> visit49 weight[75..76] time[95.4785..176]
distance[119..121] -> visit16 weight[105..106] time[95.5642..180.936]
distance[124..126] -> visit14 weight[124..125] time[95.8205..181.193]
distance[139..141] -> visit43 weight[144..145] time[96.0725..181.445]
distance[154..156] -> visit42 weight[151..152] time[96.2458..181.618]
distance[164..166] -> visit39 weight[156..157] time[96.3014..181.674]
distance[167..169] -> visit25 weight[187..188] time[172..182]
distance[185..187] -> visit24 weight[193..194] time[172.316..190]
distance[200..202] -> visit6 weight[196..197] time[172.627..208]
distance[213..215] -> depot weight[199..200] time[173.036..Inf)
distance[228..230]
vehicle2 :
 -> depot weight[0] time[0..76.2884] distance[0..10] -> visit32 weight[0..52]
time[0.309653..76.5981] distance[13..23] -> visit2 weight[23..75]
time[0.409943..76.6984] distance[18..28] -> visit11 weight[30..82] time[67..77]
distance[33..43] -> visit38 weight[42..94] time[83..93] distance[60..70] ->
visit22 weight[58..110] time[83.7578..158.666] distance[105..115] -> visit4
weight[76..128] time[149..159] distance[125..135] -> visit28 weight[95..147]
time[149.536..193.391] distance[156..166] -> visit30 weight[111..163]
time[149.975..193.83] distance[180..190] -> visit10 weight[132..184]
time[150.145..194] distance[190..200] -> depot weight[148..200]
time[150.734..Inf) distance[220..230]
vehicle3 : Unused
vehicle4 : Unused
```

```
vehicle5 : Unused
vehicle6 : Unused
vehicle7 : Unused
vehicle8 : Unused
vehicle9 : Unused
vehicle10 : Unused
Improving solution
***Solution after improvements with nhood***
Number of fails              : 0
Number of choice points      : 0
Number of variables          : 2064
Number of constraints        : 344
Reversible stack (bytes)     : 156804
Solver heap (bytes)          : 892720
Solver global heap (bytes)   : 148404
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11160
Total memory used (bytes)    : 1277500
Elapsed time since creation  : 0.01
Number of nodes              : 51
Number of visits             : 70
Number of vehicles           : 10
Number of dimensions         : 3
Number of accepted moves     : 24
===============
Cost         : 716.945
Number of vehicles used : 2
Solution     :
Unperformed visits : visit1 visit4 visit5 visit8 visit9 visit12 visit13 visit15
visit18 visit20 visit22 visit23 visit26 visit28 visit30 visit31 visit33 visit35
visit38 visit40 visit41 visit44 visit46 visit48 visit50
vehicle1 :
 -> depot weight[0] time[0..76.2884] distance[0..54] -> visit32 weight[0..46]
time[0.309653..76.5981] distance[13..67] -> visit2 weight[23..69]
time[0.409943..76.6984] distance[18..72] -> visit11 weight[30..76] time[67..77]
distance[33..87] -> visit14 weight[42..88] time[67.2136..184.748]
distance[45..99] -> visit43 weight[62..108] time[67.4656..185]
distance[60..114] -> visit42 weight[69..115] time[67.6389..185.944]
distance[70..124] -> visit39 weight[74..120] time[67.6945..186]
distance[73..127] -> visit21 weight[105..151] time[68.1551..188.853]
distance[99..153] -> visit3 weight[116..162] time[68.7423..189.441]
distance[134..188] -> visit29 weight[129..175] time[69.3016..190]
distance[163..217] -> visit27 weight[138..184] time[69.4863..215]
distance[171..225] -> depot weight[154..200] time[69.6058..Inf)
distance[176..230]
vehicle2 :
 -> depot weight[0] time[0..174.972] distance[0..46] -> visit6 weight[0..29]
time[0.409416..175.382] distance[15..61] -> visit37 weight[3..32]
time[0.756408..175.729] distance[30..76] -> visit25 weight[11..40]
time[172..175.787] distance[33..79] -> visit16 weight[17..46]
time[172.127..175.914] distance[40..86] -> visit49 weight[36..65]
time[172.213..176] distance[45..91] -> visit17 weight[66..95]
time[172.383..177.34] distance[55..101] -> visit45 weight[68..97]
time[172.537..177.494] distance[64..110] -> visit47 weight[84..113]
time[172.876..177.833] distance[84..130] -> visit36 weight[111..140]
time[173.043..178] distance[94..140] -> visit19 weight[116..145]
```

```
time[173.26..182.437] distance[107..153] -> visit10 weight[133..162]
time[173.516..182.693] distance[122..168] -> visit34 weight[149..178]
time[173.823..183] distance[140..186] -> visit7 weight[163..192]
time[173.966..189.645] distance[148..194] -> visit24 weight[168..197]
time[174.321..190] distance[166..212] -> depot weight[171..200]
time[174.813..Inf) distance[184..230]
vehicle3 : Unused
vehicle4 : Unused
vehicle5 : Unused
vehicle6 : Unused
vehicle7 : Unused
vehicle8 : Unused
vehicle9 : Unused
vehicle10 : Unused
***Solution including new visit***
Number of fails              : 1
Number of choice points      : 1049
Number of variables          : 2091
Number of constraints        : 345
Reversible stack (bytes)     : 156804
Solver heap (bytes)          : 892720
Solver global heap (bytes)   : 559036
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11160
Total memory used (bytes)    : 1688132
Elapsed time since creation  : 1.462
Number of nodes              : 52
Number of visits             : 71
Number of vehicles           : 10
Number of dimensions         : 3
Number of accepted moves     : 24
===============
Cost         : 763.538
Number of vehicles used : 2
Solution     :
Unperformed visits : visit1 visit4 visit5 visit8 visit9 visit12 visit13 visit15
visit18 visit20 visit22 visit23 visit26 visit28 visit30 visit31 visit33 visit35
visit38 visit40 visit41 visit44 visit46 visit48 visit50
vehicle1 :
 -> depot weight[0] time[0..76.2884] distance[0..18] -> visit32 weight[0..34]
time[0.309653..76.5981] distance[13..31] -> visit2 weight[23..57]
time[0.409943..76.6984] distance[18..36] -> visit11 weight[30..64] time[67..77]
distance[33..51] -> visit14 weight[42..76] time[67.2136..117.971]
distance[45..63] -> visit43 weight[62..96] time[67.4656..118.223]
distance[60..78] -> visit42 weight[69..103] time[67.6389..118.396]
distance[70..88] -> visit39 weight[74..108] time[67.6945..118.452]
distance[73..91] -> visit21 weight[105..139] time[68.1551..118.913]
distance[99..117] -> visit3 weight[116..150] time[68.7423..119.5]
distance[134..152] -> Extra visit weight[129..163] time[70..120]
distance[164..182] -> visit29 weight[141..175] time[80.6518..190]
distance[199..217] -> visit27 weight[150..184] time[80.8365..215]
distance[207..225] -> depot weight[166..200] time[80.9559..Inf)
distance[212..230]
vehicle2 :
 -> depot weight[0] time[0..174.972] distance[0..46] -> visit6 weight[0..29]
time[0.409416..175.382] distance[15..61] -> visit37 weight[3..32]
```

```
time[0.756408..175.729] distance[30..76] -> visit25 weight[11..40]
time[172..175.787] distance[33..79] -> visit16 weight[17..46]
time[172.127..175.914] distance[40..86] -> visit49 weight[36..65]
time[172.213..176] distance[45..91] -> visit17 weight[66..95]
time[172.383..177.34] distance[55..101] -> visit45 weight[68..97]
time[172.537..177.494] distance[64..110] -> visit47 weight[84..113]
time[172.876..177.833] distance[84..130] -> visit36 weight[111..140]
time[173.043..178] distance[94..140] -> visit19 weight[116..145]
time[173.26..182.437] distance[107..153] -> visit10 weight[133..162]
time[173.516..182.693] distance[122..168] -> visit34 weight[149..178]
time[173.823..183] distance[140..186] -> visit7 weight[163..192]
time[173.966..189.645] distance[148..194] -> visit24 weight[168..197]
time[174.321..190] distance[166..212] -> depot weight[171..200]
time[174.813..Inf) distance[184..230]
vehicle3 : Unused
vehicle4 : Unused
vehicle5 : Unused
vehicle6 : Unused
vehicle7 : Unused
vehicle8 : Unused
vehicle9 : Unused
vehicle10 : Unused
***Solution after removing visit***
Number of fails              : 0
Number of choice points      : 0
Number of variables          : 2068
Number of constraints        : 345
Reversible stack (bytes)     : 156804
Solver heap (bytes)          : 892720
Solver global heap (bytes)   : 559036
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11160
Total memory used (bytes)    : 1688132
Elapsed time since creation  : 1.482
Number of nodes              : 52
Number of visits             : 70
Number of vehicles           : 10
Number of dimensions         : 3
Number of accepted moves     : 24
===============
Cost         : 716.945
Number of vehicles used : 2
Solution     :
Unperformed visits : visit1 visit4 visit5 visit8 visit9 visit12 visit13 visit15
visit18 visit20 visit22 visit23 visit26 visit28 visit30 visit31 visit33 visit35
visit38 visit40 visit41 visit44 visit46 visit48 visit50
vehicle1 :
 -> depot weight[0] time[0..76.2884] distance[0..54] -> visit32 weight[0..46]
time[0.309653..76.5981] distance[13..67] -> visit2 weight[23..69]
time[0.409943..76.6984] distance[18..72] -> visit11 weight[30..76] time[67..77]
distance[33..87] -> visit14 weight[42..88] time[67.2136..184.748]
distance[45..99] -> visit43 weight[62..108] time[67.4656..185]
distance[60..114] -> visit42 weight[69..115] time[67.6389..185.944]
distance[70..124] -> visit39 weight[74..120] time[67.6945..186]
distance[73..127] -> visit21 weight[105..151] time[68.1551..188.853]
distance[99..153] -> visit3 weight[116..162] time[68.7423..189.441]
```

```
distance[134..188] -> visit29 weight[129..175] time[69.3016..190]
distance[163..217] -> visit27 weight[138..184] time[69.4863..215]
distance[171..225] -> depot weight[154..200] time[69.6058..Inf)
distance[176..230]
vehicle2 :
 -> depot weight[0] time[0..174.972] distance[0..46] -> visit6 weight[0..29]
time[0.409416..175.382] distance[15..61] -> visit37 weight[3..32]
time[0.756408..175.729] distance[30..76] -> visit25 weight[11..40]
time[172..175.787] distance[33..79] -> visit16 weight[17..46]
time[172.127..175.914] distance[40..86] -> visit49 weight[36..65]
time[172.213..176] distance[45..91] -> visit17 weight[66..95]
time[172.383..177.34] distance[55..101] -> visit45 weight[68..97]
time[172.537..177.494] distance[64..110] -> visit47 weight[84..113]
time[172.876..177.833] distance[84..130] -> visit36 weight[111..140]
time[173.043..178] distance[94..140] -> visit19 weight[116..145]
time[173.26..182.437] distance[107..153] -> visit10 weight[133..162]
time[173.516..182.693] distance[122..168] -> visit34 weight[149..178]
time[173.823..183] distance[140..186] -> visit7 weight[163..192]
time[173.966..189.645] distance[148..194] -> visit24 weight[168..197]
time[174.321..190] distance[166..212] -> depot weight[171..200]
time[174.813..Inf) distance[184..230]
vehicle3 : Unused
vehicle4 : Unused
vehicle5 : Unused
vehicle6 : Unused
vehicle7 : Unused
vehicle8 : Unused
vehicle9 : Unused
vehicle10 : Unused

*/
```

C H A P T E R

6

Multiple Tours per Vehicle

In this lesson, you will learn how to:

- ◆ solve a problem that allows vehicles to make multiple tours
- ◆ use a submodel to generate a first solution
- ◆ use the classes and functions `IloNearestAdditionGenerate`, `IloVehicle::setCost(fixedCost)`, and `IloRoutingSolution::RouteIterator`

In a VRP, the goal is to build a set of routes with each vehicle leaving from the depot and returning there at the end of the route. However, in some cases it can be advantageous to have vehicles make a return visit to the depot during the work period to pick up or drop off goods, for example when vehicles have limited capacity. It may be more cost-efficient to have one truck make several additional loading and unloading stops at the depot (tours) than to use several vehicles.

In this lesson, you will model and solve a VRP with multiple tours per vehicle. You will also use a submodel to generate the first solution.

Describe

The problem presented here is similar to a standard VRP, except that each vehicle can make multiple tours.

Step 1 Describe the problem

The first step is to write a natural language description of the problem.

The components of the routing model for this problem are the same as for a standard VRP: vehicles, customers, and a depot. In addition, return visits to the depot are modeled.

Some of the constraints in this problem are the same as those in a standard VRP: vehicle capacity and visit quantities. There are no time windows for visits and no opening or closing times at the depot. However, there is a constraint on the total amount of distance each vehicle can travel.

The objective is to minimize the total cost of the solution. The total cost of the routing plan is the total cost for all vehicles added to any costs related to unperformed visits. In this model, vehicle fixed costs and negative penalty costs for return visits are used to encourage multiple tours per vehicle:

◆ In addition to a cost proportional to the distance traveled, each vehicle also has a fixed cost associated with it. Thus, using fewer vehicles will lower the total cost of the solution.

◆ By default, the penalty cost for not performing a visit is `IloInfinity`, which forces the visit to be performed. In this problem, the return visits to the depot have a small negative penalty cost. If the return visit is performed, there is no additional cost added to the solution. If the return visit is not performed, then there is a small negative penalty cost. This makes not performing the return visit a slightly more favorable decision than performing the return visit, since you are minimizing the total cost. However, the negative penalty is so small that not performing the visit does not greatly affect the overall cost of the solution. This enables Dispatcher to make optimization decisions as to whether additional return visits to the depot are desirable.

Model

Once you have written a description of your problem, you can use Dispatcher classes to model it.

Step 2 Open the example file

Open the example file `YourDispatcherHome/examples/src/tutorial/multitr_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 of this lesson, you will have completed the code for the problem and you will be able to compile and run it.

As in Chapter 2, *Modeling a Vehicle Routing Problem*, you create a `RoutingModel` class, which is used to model the problem.

Declare the RoutingModel class

The code for the declaration of the class `RoutingModel` is provided for you:

```
class RoutingModel {
  IloEnv             _env;
  IloModel           _mdl;
  IloDispatcherGraph  _graph;
  IloDimension2       _distance;
  IloDimension1       _weight;
  IloVisitArray       _unorderedVisitArray;

  void addDimensions();
  void loadGraphInformation (const char* arcFileName);
  void lastMinuteGraphChanges ();
  void createIloNodes(const char* nodeFileName, const char* coordFileName);
  void createVehicles(const char* vehicleFileName);
  void createVisits(const char* visitsFileName);

public:
  RoutingModel(IloEnv env, int argc, char* argv[]);
  ~RoutingModel() {}
  IloEnv    getEnv() const { return _env; }
  IloModel  getModel() const { return _mdl; }
  IloDispatcherGraph getGraph() const {return _graph;}
  IloVisitArray getUnorderedVisitArray () const {return _unorderedVisitArray;}
};
```

There are only a few differences between the `RoutingModel` class used in a standard VRP and the `RoutingModel` class used in this problem.

The dimensions in this problem are weight and distance. As in Chapter 4, *Minimizing the Number of Vehicles*, you use an instance of the class `IloDispatcherGraph` to compute the shortest paths between nodes for each vehicle. It is this shortest path that is used to create the dimension `_distance` in this model. See “Define the addDimensions function” on page 152.

You create `_unorderedVisitArray`, an instance of `IloVisitArray` that will be used in the submodel to generate a first solution.

The function getGraph is used to access the road network graph in the submodel. The member function getUnorderedVisitArray is used in the submodel to access the array of unordered visits.

Define the RoutingModel constructor

The constructor is defined as in Chapter 4, *Minimizing the Number of Vehicles*. If you do not specify input files, the defaults will be used. This constructor is invoked from the main function. It calls the following functions: addDimensions, loadGraphInformation, createIloNodes, createVehicles, and createVisits.

Define the loadGraphInformation and lastMinuteGraphChanges functions

These functions are defined as in Chapter 4, *Minimizing the Number of Vehicles*.

Define the addDimensions function

The member function addDimensions is used to create the two dimensions, _weight and _distance, and add them to the model. The _weight dimension is defined as in *Chapter 2, Modeling a Vehicle Routing Problem.* The _distance dimension is computed using the shortest path from the road network graph, as in Chapter 4, *Minimizing the Number of Vehicles*. This code is provided for you:

```
void RoutingModel::addDimensions() {
  _weight =IloDimension1 (_env, "weight");
  _mdl.add(_weight);

  IloDistance SP_distance = IloGraphDistance (_graph);
  _distance =IloDimension2 (_env, SP_distance, "distance");
  _mdl.add(_distance);

}
```

In this example, you could also use the predefined distance functions IloEuclidean or IloManhattan or define your own distance function.

Define the createIloNodes function

The createIloNodes function is defined as in Chapter 4, *Minimizing the Number of Vehicles*. You use csv reader functionality to input node data and node coordinate data from csv files. You use the member function IloDispatcherGraph::associatebyCoordsInFile to look up the coordinates of a given `IloNode` in a csv file and automatically associate it to the graph node with matching coordinates. Here is the complete code for defining the `createIloNodes` function:

```
void RoutingModel::createIloNodes(const char* nodeFileName,
                                  const char* coordFileName) {
  IloCsvReader csvNodeReader(_env, nodeFileName);
  IloCsvReader::LineIterator  it(csvNodeReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char * name = line.getStringByHeader("name");
    IloNode node(_env, line.getFloatByHeader("x"),
        line.getFloatByHeader("y"), 0, name);
    node.setKey(name);
    _graph.associateByCoordsInFile (node, coordFileName);
    ++it;
  }
  csvNodeReader.end();
}
```

Define the createVehicles function

The `createVehicles` function is defined as in Chapter 4, *Minimizing the Number of Vehicles*, except that you add a fixed cost to each vehicle and you add an upper bound to the distance traveled by each vehicle.

You use `IloVehicle::setCost(IloDimension dim, IloNum unitCost)` to set the cost of each vehicle to be directly proportional to the dimension `_distance`. This is the way that you have set costs on vehicles in the lessons you have worked on so far.

Step 3 Set the vehicle cost proportional to the dimension

Add the following code after the comment
`//Set the vehicle cost proportional to the dimension`

```
vehicle.setCost(_distance, 1.0);
```

To model this problem, each vehicle also has a fixed cost associated with it in addition to a cost proportional to the distance traveled. You use `IloVehicle::setCost(fixedCost)` to set a fixed cost of 250 units on each vehicle. Thus, using fewer vehicles will lower the total cost of the solution.

Step 4 Set the fixed vehicle cost

Add the following code after the comment `//Set the fixed vehicle cost`

```
vehicle.setCost(250.0);
```

You also add a constraint using `IloVisit::getCumulVar` that the total distance traveled by the vehicle has an upper bound of 350 distance units. Since vehicles can return to the depot and there is a fixed cost for using a vehicle, Dispatcher will try to find a solution with as few vehicles as possible. However, a vehicle can only travel a certain distance during a work period. This constraint makes sure that the route of the vehicle is not longer than that specified distance.

Step 5 Add the constraint on total distance traveled by vehicle

Add the following code after the comment
`//Add the constraint on total distance traveled by vehicle`

```
    last.getCumulVar(_distance).setUb(350);
```

As in Chapter 2, *Modeling a Vehicle Routing Problem*, you use csv reader functionality to input vehicle data from a csv file. The vehicles have start and end visits. You set the capacities of the vehicles using `IloVehicle::setCapacity` and the dimension `_weight`. Here is the complete code for defining the `createVehicles` function:

```
void RoutingModel::createVehicles(const char* vehicleFileName) {
  IloCsvReader csvVehicleReader(_env, vehicleFileName);
  IloCsvReader::LineIterator  it(csvVehicleReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char * namefirst = line.getStringByHeader("first");
    char * namelast = line.getStringByHeader("last");
    char * name = line.getStringByHeader("name");
    IloNum capacity = line.getFloatByHeader("capacity");
    IloNode node1 = IloNode::Find(_env, namefirst);
    IloNode node2 = IloNode::Find(_env, namelast);
    IloVisit first(node1, "depot");
    _mdl.add(first.getCumulVar(_weight) == 0);
    IloVisit last(node2, "depot");
    IloVehicle vehicle(first, last, name);

    vehicle.setCost(250.0);

    vehicle.setCost(_distance, 1.0);

    vehicle.setCapacity(_weight, capacity);
    last.getCumulVar(_distance).setUb(350);

    _mdl.add(vehicle);
    ++it;
  }
  csvVehicleReader.end();
}
```

Define the createVisits function

The `createVisits` function is defined as in Chapter 4, *Minimizing the Number of Vehicles*, except that instead of directly adding each visit to the model, you add each visit to an array of unordered visits. This array of unordered array of visits will be ordered in the submodel and used to create a first solution. The visits created by this function are the customer visits. The return visits to the depot are created by a member function of the class `RoutingSolver`. See “Define the insertAllReturnVisits function” on page 158.

Step 6 Add the visit to the array of unordered visits

Add the following code after the comment
`//Add the visit to the array of unordered visits`

```
    _unorderedVisitArray.add(visit);
```

Likewise, the constraint on visit quantity is not added directly to the model. It is added by setting both the upper and lower bounds of the transit variable associated with the dimension `_weight` equal to `quantity`.

Step 7 Add the constraint on visit quantity

Add the following code after the comment
`//Add the constraint on visit quantity`

```
    visit.getTransitVar(_weight).setBounds(quantity, quantity);
```

Here is the complete code for defining the `createVisits` function:

```
 void RoutingModel::createVisits(const char* visitsFileName) {
  IloCsvReader csvVisitReader(_env, visitsFileName);
  IloCsvReader::LineIterator  it(csvVisitReader);
  while(it.ok()){
    IloCsvLine line = *it;
    //read visit data from files
    char * visitName =  line.getStringByHeader("name");
    char * nodeName = line.getStringByHeader("node");
    IloNum quantity = line.getFloatByHeader("quantity");
    IloNode node = IloNode::Find(_env, nodeName);
    IloVisit visit(node, visitName);

    visit.getTransitVar(_weight).setBounds(quantity, quantity);

    _unorderedVisitArray.add(visit);

    ++it;
  }
  csvVisitReader.end();
}
```

Solve

As in Chapter 3, *Solving a Vehicle Routing Problem*, you search for a solution using the two-phase approach of generating a first solution and then improving it. However, in this lesson you create a first solution in several steps.

Here is a short description of the proposed heuristic:

1. Create the return visits and insert them into the first solution.
2. Use a submodel to find a logical order for the customer visits using the predefined first solution heuristic `IloNearestAdditionGenerate`.
3. Insert the ordered customer visits into the first solution.
4. Improve the first solution using neighborhoods.

As in Chapter 3, *Solving a Vehicle Routing Problem*, you create the `RoutingSolver` class, which is used to solve the vehicle routing problem.

Declare the RoutingSolver class

The code for the declaration of the class `RoutingSolver` is provided for you:

```
class RoutingSolver {
  IloEnv              _env;
  IloModel            _mdl;
  IloSolver           _solver;
  IloDispatcher       _dispatcher;
  IloRoutingSolution  _solution;
  IloGoal             _instantiateCost;
  IloGoal             _restoreSolution;
  IloVisitArray       _orderedVisitArray;

public:
  RoutingSolver(RoutingModel mdl);
  ~RoutingSolver() {}
  void    insertAllReturnVisits ();
  void    orderVisits (IloVisitArray visitArray,
                       IloDispatcherGraph graph);
  bool    insertCustomerVisits ();
  void    improveWithNhood();
  void    printInformation(const char* =0) const;

};
```

There are several important differences between the `RoutingSolver` class used in a standard VRP and the `RoutingSolver` class used in this problem. There is an additional data member, `_orderedVisitArray`, an instance of `IloVisitArray` that will be used in the submodel. In the submodel, the visits in the `_unorderedVisitArray` created in the `RoutingModel` class will be ordered and added to the `_orderedVisitArray` created in the `RoutingSolver` class.

You do not use a single function to find the first solution, as you have in the previous lessons in this part. Instead, the first solution is generated using three functions: `insertAllReturnVisits`, `orderVisits`, and `insertCustomerVisits`. These functions are explained in the following sections.

Define the RoutingSolver constructor

The `RoutingSolver` constructor is defined as in *Chapter 3, Solving a Vehicle Routing Problem*, except that you do not create a goal for finding the first solution. The `RoutingSolver` constructor takes an instance of `RoutingModel` as a parameter. The environment, model, solver, dispatcher, and solution are initialized. The goal to instantiate cost is created using the function `IloDichotomize`. The goal to restore the solution is created using `IloRestoreSolution`. This constructor will be called from the `main` function. This code is provided for you:

```
RoutingSolver::RoutingSolver(RoutingModel mdl):
  _env(mdl.getEnv()),
  _mdl(mdl.getModel()),
  _solver(mdl.getModel()),
  _dispatcher(_solver),
  _solution(mdl.getModel()){
    _instantiateCost =
        IloDichotomize(_env, _dispatcher.getCostVar(), IloFalse);
    _restoreSolution = IloRestoreSolution (_env, _solution);
  }
```

Define the insertAllReturnVisits function

Next, you define the `insertAllReturnVisits` function which does the following:

- creates the return visits
- sets a penalty cost on them
- adds them to the model
- inserts them into the solution

You loop on each vehicle and create two return visits to the depot for each vehicle. This allows each vehicle to perform three tours. During the solution improvement phase, the number of tours can be rebalanced between vehicles.

Step 8 Create the return visits

Add the following code after the comment `//Create the return visits`

```
void RoutingSolver::insertAllReturnVisits () {
  _solver.out() << "Inserting return visits" << endl;
  IloNode depot = IloNode::Find(_env, "depot");
  for (IloVehicleIterator vehIt(_mdl); vehIt.ok(); ++vehIt) {
    IloVehicle vehicle = *vehIt;
    for (IloInt i = 0; i < 2; i++) {
      IloVisit visit(depot, "depot");
```

Now, you use `IloVisit::setPenaltyCost` to set a negative penalty cost on the return visit. By default, the penalty cost for not performing a visit is `IloInfinity`, which forces the visit to be performed. If the return visit is performed, there is no additional cost added to the solution. If the return visit is not performed, then there is a small negative penalty cost. This makes not performing the return visit a slightly more favorable decision than performing the return visit, since you are minimizing the total cost. However, the negative penalty is so small that not performing the visit does not greatly affect the overall cost of the solution.

Step 9 Set the return visit negative penalty cost

Add the following code after the comment
`//Set the return visit negative penalty cost`

```
visit.setPenaltyCost(-0.1);
```

Next, you add the return visit to the model using `IloModel::add`.

Step 10 Add the return visit to the model

Add the following code after the comment `//Add the return visit to the model`

```
_mdl.add    (visit);
```

You insert the return visit directly into the route of the vehicle by creating a goal using `IloInsertVisit`. This function makes `visit` performed without the need to recompute the routing plan from scratch. If the visit cannot be inserted, the goal fails and the visit is removed from the model. If the visit is inserted in the vehicle route, you use the function `IloRoutingSolution::add` to directly add the return visit to the solution. The new solution is then stored.

Step 11 Insert the return visits into the first solution

Add the following code after the comment
`//Insert the return visits into the first solution`

```
      IloGoal insert =
          IloInsertVisit(_env, visit, vehicle, _solution, _instantiateCost);
      if (!_solver.solve(insert)) {
        _solver.out() << "Cannot insert new visit in solution" << endl;
        _mdl.remove(visit);
      }
      else {
        _solution.add(visit);
        _solution.store(_solver);
      }
    }
  }
}
```

Define the orderVisits function

Now you create the submodel that is used to order the customer visits so that they can be inserted into the first solution along with the return visits. This submodel is a version of a basic routing problem, the Traveling Salesperson Problem (TSP). The description of this problem is fairly simple: a set of locations are given, the goal is to build a closed route (closed in the sense that it ends where it begins) that goes through each of these locations exactly once, while minimizing the cost. Unlike a standard VRP, a TSP uses only one vehicle.

The TSP submodel takes the unordered array of customer visits from the VRP model and a single vehicle and finds a solution. You then iterate over the route of the vehicle and create the array of ordered visits. These ordered visits can then be inserted in the first solution of the VRP model. If you had just inserted the visits from the unordered visit array into the first solution, this would have likely resulted in a more costly first solution. Inserting the visits that have already been ordered into the first solution will create a lower cost first solution.

Here is the heuristic used in the submodel to create the ordered array of customer visits:

1. Create the submodel.
2. Add one vehicle.
3. Add the customer visits from the unordered visit array.
4. Find a first solution using the predefined heuristic `IloNearestAdditionGenerate`.
5. Improve the solution.
6. Iterate over the route of the vehicle, adding the customer visits to the ordered visit array.

First, you define the `orderVisits` function to take two parameters: an instance of the unordered visit array and an instance of `IloDispatcherGraph`. Then, you create the submodel `tspModel` using an instance of `IloModel`. With Concert Technology, you can create more than one model in a given environment.

Step 12 Create the submodel

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

```
void RoutingSolver::orderVisits(IloVisitArray unorderedVisitArray,
                                IloDispatcherGraph graph) {
  IloModel tspModel(_env);
```

Next, you add the dimension and vehicle to the submodel. You create the first and last visits the vehicle will make—these two visits are to the same location, the first visit in the array of unordered customer visits. You create the vehicle and add it to the TSP submodel. Then you create the distance dimension using a distance function that is the shortest path in distance computed from the road network graph. You set the cost of the vehicle to be directly proportional to the distance dimension `dim` and add the dimension `dim` to the TSP submodel.

Step 13 Add the dimension and vehicle to the submodel

Add the following code after the comment
`//Add the dimension and vehicle to the submodel`

```
  IloInt nbOfVisits = unorderedVisitArray.getSize();
  IloVisit first(unorderedVisitArray[0].getNode(), "first");
  IloVisit last(unorderedVisitArray[0].getNode(), "last");
  IloVehicle vehicle(first, last, "TSP");
  tspModel.add(vehicle);
  IloDistance dist =  IloGraphDistance (graph);
  IloDimension2 dim(_env, dist, IloFalse);
  vehicle.setCost(dim, 1.0);
  tspModel.add(dim);
```

You add the visits to the TSP submodel by looping through `unorderedVisitArray` and adding each visit to the model. Since you are only concerned with generating an order based on location, you do not add any constraints on visit quantity.

Step 14 Add the customer visits to the submodel

Add the following code after the comment
`//Add the customer visits to the submodel`

```
  for (IloInt i = 1; i < nbOfVisits; i++)
    tspModel.add(unorderedVisitArray[i]);
```

As in a standard VRP, you create an instance of `IloSolver`, an instance of `IloDispatcher`, and a goal to instantiate cost.

Step 15 Create the solver, dispatcher, and cost goal in the submodel

Add the following code after the comment
`//Create the solver, dispatcher, and cost goal in the submodel`

```
  IloSolver solver(tspModel);
  IloDispatcher dispatcher(solver);
  IloGoal instCost = IloDichotomize(_env, dispatcher.getCostVar(), IloFalse);
  solver.out() << "Producing insertion order" << endl;
```

Next, you find a first solution and store it. To find the first solution to the TSP, you use the predefined first solution heuristic `IloNearestAdditionGenerate`. The nearest addition heuristic builds a first solution route by adding visits to the route. The visit added to the route is the visit closest—that is, the least costly to get to—to the end of the current partial route of the vehicle. After this visit is added, the heuristic finds the visit that is closest to the visit just added to the end of the current partial route, and so on, until all visits are added. For more information about `IloNearestAdditionGenerate`, see *Appendix A, Predefined First Solution Heuristics.*

Step 16 Find the first solution in the submodel and store it

Add the following code after the comment
`//Find the first solution in the submodel and store it`

```
  solver.solve(IloNearestAdditionGenerate(_env) && instCost);
  IloRoutingSolution rsolution(tspModel);
  rsolution.store(solver);
```

Now you improve the first solution using the `IloTwoOpt` neighborhood. For more information about this neighborhood, see Appendix B, *Predefined Neighborhoods*.

Step 17 Improve the first solution in the submodel

Add the following code after the comment
`//Improve the first solution in the submodel`

```
  IloNHood nhood = IloTwoOpt(_env);
  IloMetaHeuristic improve = IloImprove(_env);
  IloGoal move = IloSingleMove(_env, rsolution, nhood, improve, instCost);
  while (solver.solve(move)) {
    }
```

Now, you create `_orderedVisitArray`, an instance of `IloVisitArray`. You iterate over the route of the vehicle, adding visits from the route to `_orderedVisitArray`.

Step 18 Create the array of ordered customer visits in the submodel

Add the following code after the comment
`//Create the array of ordered customer visits in the submodel`

```
  _orderedVisitArray = IloVisitArray (_env, nbOfVisits);
  IloRoutingSolution::RouteIterator rit(rsolution, vehicle);
  ++rit;
  _orderedVisitArray[0] = unorderedVisitArray[0];
  for (IloInt k = 1; k < nbOfVisits; ++k, ++rit)
    _orderedVisitArray[k] = *rit;
```

Finally, you use the member functions `IloNHood::end`, `IloSolver::end`, and `IloRoutingSolution::end` to deallocate the memory used by these objects in the submodel.

Step 19 End the objects in the submodel

Add the following code after the comment `//End the objects in the submodel`

```
  nhood.end();
  solver.end();
  rsolution.end();
}
```

Define the insertCustomerVisits function

You use the `insertCustomerVisits` function to add the ordered visits from `_orderedVisitArray` to the first solution in the multitour model. As you remember, you already added the return visits to the first solution in the section "Define the insertAllReturnVisits function" on page 158.

Step 20 Add the ordered customer visits to the model

Add the following code after the comment
`//Add the ordered customer visits to the model`

```
bool RoutingSolver::insertCustomerVisits () {
  _solver.out() << "Inserting customer visits" << endl;
  for (IloInt i = 0; i < _orderedVisitArray.getSize(); i++) {
    IloVisit visit = _orderedVisitArray[i];
    _mdl.add(visit);
```

You insert the visit directly into the first solution by creating a goal using `IloInsertVisit`. This function makes `visit` performed without the need to recompute the routing plan from scratch. If the visit cannot be inserted, the goal fails. If the visit can be inserted, you use the function `IloRoutingSolution::add` to directly add the visit to the solution. The new solution is then stored.

Step 21 Insert the ordered customer visits into the first solution

Add the following code after the comment
`//Insert the ordered customer visits into the first solution`

```
      IloGoal insert =
          IloInsertVisit(_env, visit, vehicle, _solution, _instantiateCost);
      if (!_solver.solve(insert)) {
        _solver.out() << "Cannot insert new visit in solution" << endl;
        _mdl.remove(visit);
      }
      else {
        _solution.add(visit);
        _solution.store(_solver);
      }
    }
  }
}
```

Define the improveWithNHood function

After you have created a first solution by inserting the return visits and ordered visits from the submodel into the first solution, you create the neighborhoods you will use in the solution improvement phase. As in Chapter 3, *Solving a Vehicle Routing Problem*, you use the predefined neighborhoods `IloRelocate`, `IloExchange`, `IloCross`, `IloTwoOpt`, and `IloOrOpt`. Additionally, you use two other predefined neighborhoods: `IloMakePerformed` and `IloMakeUnperformed`. The function `IloMakePerformed` returns a neighborhood that modifies a solution by inserting an unperformed visit after a performed one. The function `IloMakeUnperformed` returns a neighborhood that modifies a solution by causing a performed visit to be unperformed. These neighborhoods are used to improve a solution by making return visits performed and unperformed.

For more information about how predefined neighborhoods work, see Appendix B, *Predefined Neighborhoods*.

This code is provided for you:

```
void RoutingSolver::improveWithNhood() {
  IloNHood nhood = IloRelocate(_env)
                   + IloTwoOpt(_env)
                   + IloOrOpt(_env)
                   + IloCross(_env)
                   + IloExchange(_env)
                   + IloMakeUnperformed(_env)
                   + IloMakePerformed(_env);
  _solver.out() << "Improving solution" << endl;
  IloGoal improve = IloSingleMove(_env,
                                  _solution,
                                  nhood,
                                  IloImprove(_env),
                                  _instantiateCost);
  while (_solver.solve(improve)) {
  }
   _solver.solve(_restoreSolution);
}
```

Define the printInformation function

The `printInformation` function is the same as in Chapter 3, *Solving a Vehicle Routing Problem*.

Define the main function

After you finish creating the `RoutingModel` and `RoutingSolver` classes and the `printInformation` function, you use them in the `main` function. You can use command line syntax to pass the names of input files to the model. If you do not specify input files, the defaults will be used. In the `main` function, you first create an environment. Then you create an instance of the `RoutingModel` class, which takes the environment and input files as parameters. You create an instance of the `RoutingSolver` class, which takes the model as a parameter. You call the member function `RoutingSolver::insertAllReturnVisits` to add the return visits to the first solution. You call the member function `RoutingSolver::orderVisits` to create a submodel and order the customer visits. This member function takes two parameters: the unordered visit array and the graph from the routing model `mdl`. You call the member function `RoutingSolver::insertCustomerVisits` to insert the ordered customer visits into the first solution. You use Solver to improve the first solution and print this solution. The following code is provided for you:

```
int main(int argc, char * argv[]) {
  IloEnv env;
  try {
    RoutingModel mdl(env, argc, argv);
    RoutingSolver solver(mdl);
    solver.insertAllReturnVisits();
    solver.orderVisits(mdl.getUnorderedVisitArray(), mdl.getGraph());
    if (solver.insertCustomerVisits()) {
      solver.improveWithNhood();
      solver.printInformation("***Solution after improvements with nhood***");
    }
  } catch(IloException& ex) {
    cerr << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Step 22 Compile and run the program

Compile and run the program. You will get results that show the routing plan and information for the improved solution. The solution uses 4 vehicles. Each vehicle makes one return visit to the depot, that is, each vehicle performs two tours. There are 12 unused return visits. See “Complete Output” on page 174 for details. The solution improvement phase finds a solution using 4 vehicles with a total cost of 2114.8 units after making 139 cost-decreasing moves:

```
Inserting return visits
Producing insertion order
Inserting customer visits
Improving solution
***Solution after improvements with nhood***
Number of fails              : 0
Number of choice points      : 0
Number of variables          : 2196
Number of constraints        : 132
Reversible stack (bytes)     : 192984
Solver heap (bytes)          : 687828
Solver global heap (bytes)   : 1021680
And stack (bytes)            : 20124
Or stack (bytes)             : 48264
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 13164
Total memory used (bytes)    : 1988088
Elapsed time since creation  : 0.016
Number of nodes              : 101
Number of visits             : 132
Number of vehicles           : 8
Number of dimensions         : 2
Number of accepted moves     : 139
===============
Cost         : 2114.8
Number of vehicles used : 4
```

The complete program and output are listed in “Complete Program” on page 168. You can also view it online in the `YourDispatcherHome/examples/src/multitr.cpp` file.

Review Exercises

1. How are vehicle fixed costs used to encourage multiple tours per vehicle?
2. How are negative penalty costs for return visits used to encourage multiple tours per vehicle?

Suggested Answers

Exercise 1

How are vehicle fixed costs used to encourage multiple tours per vehicle?

Suggested Answer

In addition to a cost proportional to the distance traveled, each vehicle also has a fixed cost associated with it. Thus, using fewer vehicles will lower the total cost of the solution.

Exercise 2

How are negative penalty costs for return visits used to encourage multiple tours per vehicle?

Suggested Answer

By default, the penalty cost for not performing a visit is `IloInfinity`, which forces the visit to be performed. In this problem, the return visits to the depot have a small negative penalty cost. If the return visit is performed, there is no additional cost added to the solution. If the return visit is not performed, then there is a small negative penalty cost. This makes not performing the return visit a slightly more favorable decision than performing the return visit, since you are minimizing the total cost. However, the negative penalty is so small that not performing the visit does not greatly affect the overall cost of the solution. This enables Dispatcher to make optimization decisions as to whether additional return visits to the depot are desirable.

Complete Program

The complete program follows. You can also view it online in the `YourDispatcherHome/examples/src/multitr.cpp` file.

```
// -------------------------------------------------------------- -*- C++ -*-
// File: examples/src/multitr.cpp
// --------------------------------------------------------------------------


#include <ildispat/ilodispatcher.h>

ILOSTLBEGIN
////////////////////////////////////////////////////////////////////////////////
// Modeling
class RoutingModel {
  IloEnv              _env;
  IloModel            _mdl;
  IloDispatcherGraph  _graph;
  IloDimension2       _distance;
  IloDimension1       _weight;
  IloVisitArray       _unorderedVisitArray;

  void addDimensions();
  void loadGraphInformation (const char* arcFileName);
  void lastMinuteGraphChanges ();
  void createIloNodes(const char* nodeFileName, const char* coordFileName);
  void createVehicles(const char* vehicleFileName);
  void createVisits(const char* visitsFileName);

public:
  RoutingModel(IloEnv env, int argc, char* argv[]);
  ~RoutingModel() {}
  IloEnv    getEnv() const { return _env; }
  IloModel  getModel() const { return _mdl; }
  IloDispatcherGraph getGraph() const {return _graph;}
  IloVisitArray getUnorderedVisitArray () const {return _unorderedVisitArray;}
};

RoutingModel::RoutingModel( IloEnv env,
                            int argc,
                            char* argv[]):
  _env(env),
  _mdl(env),
  _graph(env),
  _unorderedVisitArray(env){
  addDimensions();

  // Load dispatcher graph information from file .
  // Add instance-specific features to network
  char * arcFileName;
  if(argc < 2)  arcFileName =
       (char*) "../../../examples/data/dispatcherGraphData/gridNetwork.csv";
  else          arcFileName = argv[1];
  char * turnFileName;
  if(argc < 3)  turnFileName =
       (char*) "../../../examples/data/dispatcherGraphData/turnData.csv";
  else          turnFileName = argv[2];
  loadGraphInformation (arcFileName);

  //create IloNodes and associate them to graph nodes
  char * nodeFileName;
  if(argc < 4)  nodeFileName =
       (char*) "../../../examples/data/vrp100/vrp100nodes.csv";
  else          nodeFileName = argv[3];
  char * nodeCoordsFile;
  if(argc < 5)  nodeCoordsFile =
       (char*) "../../../examples/data/dispatcherGraphData/coordTable.csv";
  else          nodeCoordsFile = argv[4];
  createIloNodes(nodeFileName, nodeCoordsFile);

  //create vehicles
  char * vehiclesFileName;
  if(argc < 6)  vehiclesFileName =
       (char*) "../../../examples/data/vrp100/vrp100vehicles.csv";
  else          vehiclesFileName = argv[5];
  createVehicles(vehiclesFileName);

  //create visits
  char * visitsFileName;
  if(argc < 7)  visitsFileName =
       (char*) "../../../examples/data/vrp100/vrp100visits.csv";
  else          visitsFileName = argv[6];
  createVisits(visitsFileName);
}
```

```
// create distance functions for dimensions, add dimensions to model
void RoutingModel::addDimensions() {
  _weight =IloDimension1 (_env, “weight”);
  _mdl.add(_weight);

  IloDistance SP_distance = IloGraphDistance (_graph);
  _distance =IloDimension2 (_env, SP_distance, “distance”);
  _mdl.add(_distance);

}

// load network topology and travel costs from files.
// note that by default, all turns are allowed with no penalty.
void RoutingModel::loadGraphInformation (const char* arcFileName)    {
  _graph.createArcsFromFile (arcFileName);
  _graph.loadArcDimensionDataFromFile (arcFileName, _distance);
  lastMinuteGraphChanges();
}

// Make modifications to network conditions based on latest information.
void RoutingModel::lastMinuteGraphChanges () {
  _graph.forbidArcUse(_graph.getArcByEnds(2785-1, 2785));
  _graph.forbidArcUse(_graph.getArcByEnds(2785+1, 2785));
  _graph.forbidArcUse(_graph.getArcByEnds(2785-56, 2785));
}

//create IloNodes
void RoutingModel::createIloNodes(const char* nodeFileName,
                                  const char* coordFileName) {
  IloCsvReader csvNodeReader(_env, nodeFileName);
  IloCsvReader::LineIterator  it(csvNodeReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char * name = line.getStringByHeader(“name”);
    IloNode node(_env, line.getFloatByHeader(“x”),
        line.getFloatByHeader(“y”), 0, name);
    node.setKey(name);
    _graph.associateByCoordsInFile (node, coordFileName);
    ++it;
  }
  csvNodeReader.end();
}

//create vehicles
void RoutingModel::createVehicles(const char* vehicleFileName) {
  IloCsvReader csvVehicleReader(_env, vehicleFileName);
  IloCsvReader::LineIterator  it(csvVehicleReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char * namefirst = line.getStringByHeader(“first”);
    char * namelast = line.getStringByHeader(“last”);
    char * name = line.getStringByHeader(“name”);
    IloNum capacity = line.getFloatByHeader(“capacity”);
    IloNode node1 = IloNode::Find(_env, namefirst);
    IloNode node2 = IloNode::Find(_env, namelast);
    IloVisit first(node1, “depot”);
    _mdl.add(first.getCumulVar(_weight) == 0);
    IloVisit last(node2, “depot”);
    IloVehicle vehicle(first, last, name);

    vehicle.setCost(250.0);

    vehicle.setCost(_distance, 1.0);

    vehicle.setCapacity(_weight, capacity);
    last.getCumulVar(_distance).setUb(350);

    _mdl.add(vehicle);
    ++it;
  }
  csvVehicleReader.end();
}

//create visits
void RoutingModel::createVisits(const char* visitsFileName) {
  IloCsvReader csvVisitReader(_env, visitsFileName);
  IloCsvReader::LineIterator  it(csvVisitReader);
  while(it.ok()){
    IloCsvLine line = *it;
    //read visit data from files
    char * visitName =  line.getStringByHeader(“name”);
    char * nodeName = line.getStringByHeader(“node”);
    IloNum quantity = line.getFloatByHeader(“quantity”);
    IloNode node = IloNode::Find(_env, nodeName);
    IloVisit visit(node, visitName);

    visit.getTransitVar(_weight).setBounds(quantity, quantity);

    _unorderedVisitArray.add(visit);

    ++it;
  }
  csvVisitReader.end();
}

///////////////////////////////////////////////////////////////////////////////
// Solving
class RoutingSolver {
  IloEnv             _env;
  IloModel           _mdl;
  IloSolver          _solver;
  IloDispatcher      _dispatcher;
  IloRoutingSolution  _solution;
  IloGoal            _instantiateCost;
  IloGoal            _restoreSolution;
  IloVisitArray      _orderedVisitArray;

public:
  RoutingSolver(RoutingModel mdl);
  ~RoutingSolver() {}
  void    insertAllReturnVisits ();
  void    orderVisits (IloVisitArray visitArray,
                       IloDispatcherGraph graph);
  bool    insertCustomerVisits ();
  void    improveWithNhood();
  void    printInformation(const char* =0) const;
```

```
};

RoutingSolver::RoutingSolver(RoutingModel mdl):
  _env(mdl.getEnv()),
  _mdl(mdl.getModel()),
  _solver(mdl.getModel()),
  _dispatcher(_solver),
  _solution(mdl.getModel()){
    _instantiateCost =
        IloDichotomize(_env, _dispatcher.getCostVar(), IloFalse);
    _restoreSolution = IloRestoreSolution (_env, _solution);
  }

void RoutingSolver::insertAllReturnVisits () {
  _solver.out() << “Inserting return visits” << endl;
  IloNode depot = IloNode::Find(_env, “depot”);
  for (IloVehicleIterator vehIt(_mdl); vehIt.ok(); ++vehIt) {
    IloVehicle vehicle = *vehIt;
    for (IloInt i = 0; i < 2; i++) {
      IloVisit visit(depot, “depot”);

      visit.setPenaltyCost(-0.1);

      _mdl.add    (visit);

      IloGoal insert =
          IloInsertVisit(_env, visit, vehicle, _solution, _instantiateCost);
      if (!_solver.solve(insert)) {
        _solver.out() << “Cannot insert new visit in solution” << endl;
        _mdl.remove(visit);
      }
      else {
        _solution.add(visit);
        _solution.store(_solver);
      }
    }
  }
}

void RoutingSolver::orderVisits(IloVisitArray unorderedVisitArray,
                                IloDispatcherGraph graph) {
  IloModel tspModel(_env);

  IloInt nbOfVisits = unorderedVisitArray.getSize();
  IloVisit first(unorderedVisitArray[0].getNode(), “first”);
  IloVisit last(unorderedVisitArray[0].getNode(), “last”);
  IloVehicle vehicle(first, last, “TSP”);
  tspModel.add(vehicle);
  IloDistance dist =  IloGraphDistance (graph);
  IloDimension2 dim(_env, dist, IloFalse);
  vehicle.setCost(dim, 1.0);
  tspModel.add(dim);

  for (IloInt i = 1; i < nbOfVisits; i++)
    tspModel.add(unorderedVisitArray[i]);

  IloSolver solver(tspModel);
  IloDispatcher dispatcher(solver);
  IloGoal instCost = IloDichotomize(_env, dispatcher.getCostVar(), IloFalse);
  solver.out() << “Producing insertion order” << endl;

  solver.solve(IloNearestAdditionGenerate(_env) && instCost);
  IloRoutingSolution rsolution(tspModel);
  rsolution.store(solver);

  IloNHood nhood = IloTwoOpt(_env);
  IloMetaHeuristic improve = IloImprove(_env);
  IloGoal move = IloSingleMove(_env, rsolution, nhood, improve, instCost);
  while (solver.solve(move)) {
    }

  _orderedVisitArray = IloVisitArray (_env, nbOfVisits);
  IloRoutingSolution::RouteIterator rit(rsolution, vehicle);
  ++rit;
  _orderedVisitArray[0] = unorderedVisitArray[0];
  for (IloInt k = 1; k < nbOfVisits; ++k, ++rit)
    _orderedVisitArray[k] = *rit;

  nhood.end();
  solver.end();
  rsolution.end();
}

bool RoutingSolver::insertCustomerVisits () {
  _solver.out() << “Inserting customer visits” << endl;
  for (IloInt i = 0; i < _orderedVisitArray.getSize(); i++) {
    IloVisit visit = _orderedVisitArray[i];
    _mdl.add(visit);

    IloGoal insert = IloInsertVisit(_env, visit, _solution, _instantiateCost);
    if (!_solver.solve(insert)) {
      _solver.error() << “Could not generate an initial solution” << endl;
      return IloFalse;
    }
    else {
      _solution.add(visit);
      _solution.store(_solver);
    }
  }
  return IloTrue;
}

//Improve solution using nhood
void RoutingSolver::improveWithNhood() {
  IloNHood nhood = IloRelocate(_env)
                   + IloTwoOpt(_env)
                   + IloOrOpt(_env)
                   + IloCross(_env)
                   + IloExchange(_env)
                   + IloMakeUnperformed(_env)
                   + IloMakePerformed(_env);
  _solver.out() << “Improving solution” << endl;
  IloGoal improve = IloSingleMove(_env,
                                  _solution,
                                  nhood,
                                  IloImprove(_env),
                                  _instantiateCost);
  while (_solver.solve(improve)) {
  }
   _solver.solve(_restoreSolution);
}

// Display Dispatcher information
void RoutingSolver::printInformation(const char* heading) const {
  if(heading)
      _solver.out()<<heading<<endl;
  _solver.printInformation();
  _dispatcher.printInformation();
  _solver.out() << “===============” << endl
    << “Cost         : “ << _dispatcher.getTotalCost() << endl
    << “Number of vehicles used : “
    << _dispatcher.getNumberOfVehiclesUsed() << endl
    << “Solution     : “ << endl
    << _dispatcher << endl;
}

///////////////////////////////////////////////////////////////////////////////
int main(int argc, char * argv[]) {
  IloEnv env;
  try {
    RoutingModel mdl(env, argc, argv);
    RoutingSolver solver(mdl);
    solver.insertAllReturnVisits();
    solver.orderVisits(mdl.getUnorderedVisitArray(), mdl.getGraph());
    if (solver.insertCustomerVisits()) {
      solver.improveWithNhood();
      solver.printInformation(“***Solution after improvements with nhood***”);
    }
  } catch(IloException& ex) {
    cerr << “Error: “ << ex << endl;
  }
  env.end();
  return 0;
}
```

Complete Output

```
/**
Inserting return visits
Producing insertion order
Inserting customer visits
Improving solution
***Solution after improvements with nhood***
Number of fails              : 0
Number of choice points      : 0
Number of variables          : 2196
Number of constraints        : 132
Reversible stack (bytes)     : 192984
Solver heap (bytes)          : 687828
Solver global heap (bytes)   : 1021680
And stack (bytes)            : 20124
```

```
Or stack (bytes)              : 48264
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 13164
Total memory used (bytes)     : 1988088
Elapsed time since creation   : 0.016
Number of nodes               : 101
Number of visits              : 132
Number of vehicles            : 8
Number of dimensions          : 2
Number of accepted moves      : 139
===============
Cost         : 2114.8
Number of vehicles used : 4
Solution     :
Unperformed visits : depot depot depot depot depot depot depot depot depot
depot depot depot
vehicle1 : Unused
vehicle2 : Unused
vehicle3 :
 -> depot weight[0] distance[0..78] -> visit32 weight[0..7] distance[1..79] ->
visit28 weight[23..30] distance[8..86] -> visit80 weight[39..46]
distance[15..93] -> visit68 weight[45..52] distance[17..95] -> visit54
weight[81..88] distance[32..110] -> visit58 weight[99..106] distance[42..120] -
> visit87 weight[117..124] distance[58..136] -> visit100 weight[143..150]
distance[68..146] -> visit98 weight[160..167] distance[72..150] -> visit99
weight[170..177] distance[78..156] -> visit96 weight[179..186]
distance[81..159] -> visit6 weight[190..197] distance[87..165] -> depot
weight[193..200] distance[102..180] -> visit24 weight[0..4] distance[120..198]
-> visit60 weight[3..7] distance[121..199] -> visit5 weight[6..10]
distance[127..205] -> visit16 weight[32..36] distance[142..220] -> visit86
weight[51..55] distance[150..228] -> visit17 weight[86..90] distance[163..241]
-> visit84 weight[88..92] distance[170..248] -> visit83 weight[95..99]
distance[179..257] -> visit45 weight[106..110] distance[188..266] -> visit46
weight[122..126] distance[202..280] -> visit36 weight[123..127]
distance[214..292] -> visit47 weight[128..132] distance[224..302] -> visit48
weight[155..159] distance[233..311] -> visit7 weight[191..195]
distance[242..320] -> depot weight[196..200] distance[272..350]
vehicle4 : Unused
vehicle5 : Unused
vehicle6 :
 -> depot weight[0] distance[0..18] -> visit22 weight[0..4] distance[35..53] ->
visit75 weight[18..22] distance[40..58] -> visit23 weight[36..40]
distance[52..70] -> visit56 weight[65..69] distance[61..79] -> visit55
weight[71..75] distance[73..91] -> visit76 weight[73..77] distance[95..113] ->
visit50 weight[86..90] distance[102..120] -> visit70 weight[99..103]
distance[121..139] -> visit30 weight[104..108] distance[128..146] -> visit10
weight[125..129] distance[138..156] -> visit88 weight[141..145]
distance[150..168] -> visit31 weight[150..154] distance[155..173] -> visit90
weight[177..181] distance[163..181] -> visit27 weight[180..184]
distance[171..189] -> depot weight[196..200] distance[176..194] -> visit53
weight[0..4] distance[182..200] -> visit40 weight[14..18] distance[191..209] ->
visit41 weight[23..27] distance[211..229] -> visit15 weight[28..32]
distance[225..243] -> visit67 weight[36..40] distance[234..252] -> visit92
weight[61..65] distance[252..270] -> visit59 weight[63..67] distance[255..273]
-> visit93 weight[91..95] distance[258..276] -> visit61 weight[113..117]
distance[264..282] -> visit8 weight[126..130] distance[285..303] -> visit49
weight[135..139] distance[291..309] -> visit18 weight[165..169]
distance[306..324] -> visit52 weight[177..181] distance[316..334] -> visit63
```

```
weight[186..190] distance[323..341] -> depot weight[196..200]
distance[332..350]
vehicle7 :
 -> depot weight[0] distance[0..116] -> visit12 weight[0..1] distance[15..131]
-> visit29 weight[19..20] distance[26..142] -> visit51 weight[28..29]
distance[47..163] -> visit9 weight[38..39] distance[55..171] -> visit35
weight[54..55] distance[63..179] -> visit20 weight[62..63] distance[70..186] ->
visit11 weight[71..72] distance[95..211] -> visit62 weight[83..84]
distance[106..222] -> visit19 weight[102..103] distance[117..233] -> visit34
weight[119..120] distance[124..240] -> visit82 weight[133..134]
distance[134..250] -> visit64 weight[149..150] distance[138..254] -> visit78
weight[158..159] distance[158..274] -> visit79 weight[161..162]
distance[171..287] -> visit71 weight[184..185] distance[178..294] -> depot
weight[199..200] distance[192..308] -> visit89 weight[0..115]
distance[201..317] -> visit94 weight[15..130] distance[209..325] -> visit95
weight[42..157] distance[213..329] -> visit13 weight[62..177]
distance[219..335] -> depot weight[85..200] distance[234..350]
vehicle8 :
 -> depot weight[0] distance[0..72] -> visit2 weight[0..4] distance[18..90] ->
visit73 weight[7..11] distance[27..99] -> visit74 weight[16..20]
distance[33..105] -> visit39 weight[24..28] distance[47..119] -> visit57
weight[55..59] distance[48..120] -> visit42 weight[62..66] distance[56..128] ->
visit43 weight[67..71] distance[66..138] -> visit38 weight[74..78]
distance[86..158] -> visit44 weight[90..94] distance[101..173] -> visit14
weight[108..112] distance[109..181] -> visit91 weight[128..132]
distance[118..190] -> visit85 weight[129..133] distance[122..194] -> visit25
weight[170..174] distance[125..197] -> visit37 weight[176..180]
distance[128..200] -> visit97 weight[184..188] distance[134..206] -> depot
weight[196..200] distance[158..230] -> visit26 weight[0..3] distance[173..245]
-> visit21 weight[17..20] distance[183..255] -> visit72 weight[28..31]
distance[189..261] -> visit4 weight[53..56] distance[201..273] -> visit77
weight[72..75] distance[226..298] -> visit3 weight[86..89] distance[230..302] -
> visit81 weight[99..102] distance[239..311] -> visit33 weight[125..128]
distance[243..315] -> visit65 weight[136..139] distance[246..318] -> visit66
weight[156..159] distance[248..320] -> visit1 weight[181..184]
distance[258..330] -> visit69 weight[191..194] distance[264..336] -> depot
weight[197..200] distance[278..350]

*/
```

Part II

Transportation Industry Solutions

This part consists of the following lessons:

- Chapter 7, *Pickup and Delivery Problems*
- Chapter 8, *Adding Vehicle Breaks*
- Chapter 9, *Adding Early and Late Costs*
- Chapter 10, *Pickup and Delivery by Multiple Vehicles from Multiple Depots*
- Chapter 11, *Modeling Complex Costs*
- Chapter 12, *Docking Bays: Modeling External Resources*

C H A P T E R

7

Pickup and Delivery Problems

In this lesson, you will learn how to:

- model pickups and deliveries in the same route
- create pairs of visits
- use the member function `IloOrderedVisitPair`

Describe

The problem of en route pickup and delivery is known as the Pickup and Delivery Problem (PDP). It is an extension of the Vehicle Routing Problem (VRP). The important new features of this problem are that pickups can occur en route at customer sites (away from the central depot), and that deliveries are allowed in the same route. This occurs, for example, with transportation systems for handicapped people, where a person can be picked up at one place and dropped at another without going through the central depot. A PDP can be either a single-depot or multiple-depot problem.

PDP problems involve the same kinds of constraints as standard VRPs, with several differences. One is that the vehicle may reach its capacity limit en route (after multiple pickup visits) rather than while loading at the depot. Another difference is that because pickups are occurring en route, the pickups become subject to delays and time windows, just as with the drop off visits. In addition, the pickup and delivery visits are often constrained to be performed by the same vehicle, in order to avoid a return trip to the central depot.

The following figure shows a sample solution for a PDP:

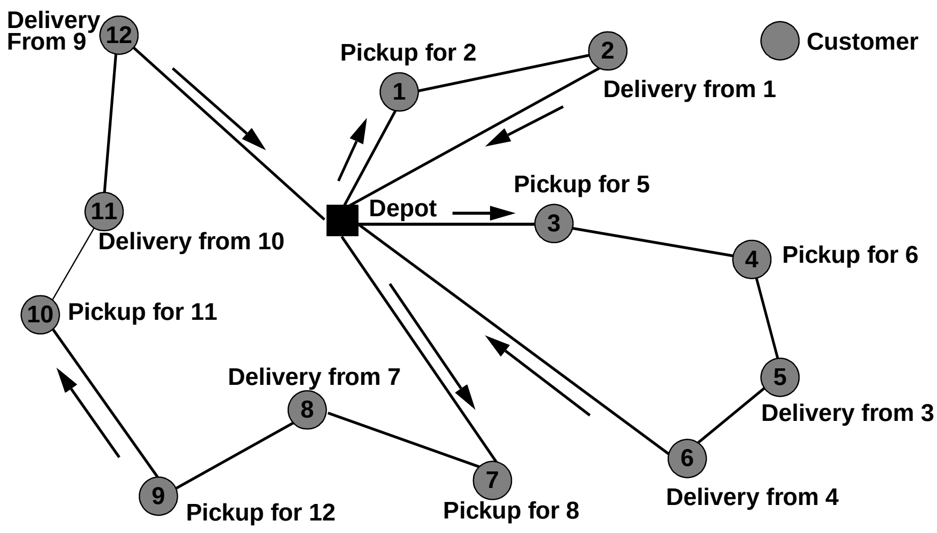

Figure 7.1 *An Example PDP Solution*

Step 1 Describe the problem

The first step is to write a natural language description of the problem. A good way to start this process is to analyze the constraints and objectives.

What are the constraints in this problem?

- ◆ Delivery trucks will pick up parcels en route and deliver them on the same route. This is called an *ordered visit*. No parcels will return to the depot before delivery.
- ◆ Capacity constraints on the vehicles have different ramifications than for a VRP; for example, the truck may reach a maximum capacity constraint along the route, rather than when departing the depot.
- ◆ Pickups as well as deliveries have constraints on the time windows (times that the parcel can be picked up from or delivered to a customer site).

The objective is to minimize the cost of the delivery of all the parcels.

Model

Once you have written a description of your problem, you can use Dispatcher classes to model it.

Step 2 Open the example file

Open the example file `YourDispatcherHome/examples/src/tutorial/pdp_partial.cpp` in your development environment.

As in the previous examples, you will use a `RoutingModel` class to call the functions that create the dimensions, nodes, vehicles, and visits. For this example, this does not vary from the form you have used before.

Step 3 Declare the RoutingModel class

Add the following code after the comment `//Declare the RoutingModel class.`

```
class RoutingModel {
  IloEnv              _env;
  IloModel            _mdl;

  void createDimensions();
  void createIloNodes(char* nodeFileName);
  void createVehicles(char* vehicleFileName);
  void createVisits(char* visitsFileName);
public:
  RoutingModel(IloEnv env,
               int argc,
               char* argv[]);
  ~RoutingModel() {}
  IloEnv getEnv() const { return _env; }
  IloModel getModel() const { return _mdl; }
};
```

The `RoutingModel` class is identified as belonging to the model `_mdl` and the environment `_env`. Next, you will create the functions that `RoutingModel` calls.

Step 4 Create the dimensions

Add the following code after the comment `//Create the dimensions`.

```
void RoutingModel::createDimensions() {
  IloDimension1 weight(_env, "Weight");
  weight.setKey("Weight");
  _mdl.add(weight);
  IloDimension2 time(_env, IloEuclidean, "Time");
  time.setKey("Time");
  _mdl.add(time);
  IloDimension2 distance(_env, IloEuclidean, IloFalse, "Distance");
  distance.setKey("Distance");
  _mdl.add(distance);
}
```

These dimensions are used in determining the cost and feasibility of potential routes. For example, `weight` is used for keeping track of the capacity constraints on the truck, and `time` is used to ensure that the truck is available during necessary time windows. The dimension `distance` will not be constrained for any vehicle, so you use `IloFalse` to prevent its propagation (although it is used to determine the cost of the route). Preventing its propagation improves performance. Just as in the VRP, `time` and `distance` are computed as Euclidian distance, although other options are available.

The member function `setKey` is used on dimensions to make it easy to find them in other functions, such as `createVisits` and `createVehicles`.

The function `createVehicles` constrains vehicles to visit the depot first and last, and adds the capacity constraint `weight` to the vehicles. The constraint of `time` is added to ensure compliance with the time windows, and `distance` is added to compute the cost of using the vehicle.

Step 5 Create the vehicles

Add the following code after the comment `//Create the vehicles`.

```
    IloVisit first(node1, "depot");
    _mdl.add(first.getCumulVar(weight) == 0);
    _mdl.add(first.getCumulVar(time) >= openTime);

    IloVisit last(node2, "depot");
    _mdl.add(last.getCumulVar(time) <= closeTime);
    IloVehicle vehicle(first, last, name);
    vehicle.setCost(distance, 1.0);
    vehicle.setCapacity(weight, capacity);
    _mdl.add(vehicle);
```

This code is the second part of the `createVehicles` function. The first part of the iteration reads in the truck capacity and the opening and closing times of the depot, which are then assigned as the first and last visits of the vehicle. The `Find` function is also used to locate the dimensions (identified with `setKey`, previously) as shown in the code provided for you:

```
IloDimension1 weight = IloDimension1::Find(_env, "Weight");
IloDimension2 time = IloDimension2::Find(_env, "Time");
IloDimension2 distance = IloDimension2::Find(_env, "Distance");
```

The next function called by the class `RoutingModel` is `createIloNodes`, and it is provided for you. `createIloNodes` reads in the (x, y) coordinates of each node, which are used to compute the Euclidian distances between visits.

Next, you must create the pickup and delivery visits. The function `createVisits` is similar in format to `createVehicles`; that is, the first part of the iteration reads in data associated with each visit, like the pickup location (`pickupNode`), the parcel to be picked up (`quantity`), the time window (`pickupMinTime` and `pickupMaxTime`), and the amount of time it takes to pickup the parcel (`pickupTime`). Corresponding data for the drop off is also processed. Then, the second section of the iteration actually creates and adds the pickup and delivery visits.

Step 6 Create the pickup visits

Add the following code after the comment `//Create the pickup visits.`

```
IloVisit pickup(pickupNode, pickupVisitName);
_mdl.add(pickup.getDelayVar(time) == pickupTime);
_mdl.add(pickup.getTransitVar(weight) == quantity);
_mdl.add(pickupMinTime <= pickup.getCumulVar(time) <= pickupMaxTime);
_mdl.add(pickup);
```

This section of the function `CreateVisits` ensures that the pickups and deliveries will occur within the time windows, including any delays associated with the pickup and drop off.

Step 7 Create the delivery visits

Add the following code for the drop off visits after the comment `//Create the delivery visits`.

```
IloVisit delivery(deliveryNode, deliveryVisitName);
_mdl.add(delivery.getDelayVar(time) == dropTime);
_mdl.add(delivery.getTransitVar(weight) == -quantity);
_mdl.add(deliveryMinTime <= delivery.getCumulVar(time) <= deliveryMaxTime);
_mdl.add(delivery);
```

You have now added the pickups and deliveries; but nothing yet guarantees that a delivery will occur after the pickup. The member function `IloOrderedVisitPair` is used to fix the order of two events.

Step 8 Create the pickup and delivery order constraint

Add the following code after the comment
`//Create the pickup and delivery order constraint`.

```
    _mdl.add(IloOrderedVisitPair(_env, pickup, delivery));
```

The function `IloOrderedVisitPair` ensures that `pickup` occurs before `delivery` in the environment `_env`. This line of code is within the iterator of `createVisits`, so each parcel pickup is paired with the drop off.

Solve

The solution is computed, improved, and displayed by methods previously presented for the VRP problem. One thing to note is that the program `pdp.cpp`, as written, can also be used to resolve a problem where some or all of the parcels are initially in the depot.

Step 9 Compile and run the program

The solution improvement phase finds a solution using 9 vehicles with a cost of 1078.88 units:

```
1291.75
Improving solution
Number of fails               : 0
Number of choice points       : 0
Number of variables           : 2329
Number of constraints         : 410
Reversible stack (bytes)      : 197004
Solver heap (bytes)           : 1009300
Solver global heap (bytes)    : 144384
And stack (bytes)             : 20124
Or stack (bytes)              : 44244
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 11160
Total memory used (bytes)     : 1430260
Elapsed time since creation   : 0.02
Number of nodes               : 51
Number of visits              : 80
Number of vehicles            : 15
Number of dimensions          : 3
Number of accepted moves      : 23
===============
Cost         : 1078.88
Number of vehicles used : 9
```

The complete program and output are listed in “Complete Program” on page 186. You can also view it online in the `YourDispatcherHome/examples/src/pdp.cpp` file.

Review Exercises

For answers, see “Suggested Answers” on page 185.

1. What is an *ordered visit*?
2. What function is used to ensure that ordered visits occur?
3. Does using this ordered visit function necessarily mean that the same vehicle will be performing both the pickup and the delivery visits?

Suggested Answers

Exercise 1

What is an *ordered visit*?

Suggested Answer

An ordered visit means that one visit (in this example, a pickup) is constrained to occur before another visit (in this example, a delivery).

Exercise 2

What function is used to ensure that ordered visits occur?

Suggested Answer

`IloOrderedVisitPair`.

Exercise 3

Does using the ordered visit function (`IloOrderedVisitPair`) necessarily mean that the same vehicle will be performing both the pickup and the delivery visits?

Suggested Answer

Yes. See the description of `IloOrderedVisitPair` in the *IBM ILOG Dispatcher Reference Manual* for details.

Complete Program

The complete program follows. You can also view it online in the file `YourDispatcherHome/examples/src/pdp.cpp`.

```
// -------------------------------------------------------------- -*- C++ -*-
// File: examples/src/pdp.cpp
// --------------------------------------------------------------------------


#include <ildispat/ilodispatcher.h>

ILOSTLBEGIN
///////////////////////////////////////////////////////////////////////////////
// Modeling
class RoutingModel {
  IloEnv              _env;
  IloModel            _mdl;

  void createDimensions();
  void createIloNodes(char* nodeFileName);
  void createVehicles(char* vehicleFileName);
  void createVisits(char* visitsFileName);
public:
  RoutingModel(IloEnv env,
               int argc,
               char* argv[]);
  ~RoutingModel() {}
  IloEnv getEnv() const { return _env; }
  IloModel getModel() const { return _mdl; }
};

RoutingModel::RoutingModel(IloEnv env,
                                       int argc,
                                       char * argv[])
  : _env(env), _mdl(env){

  createDimensions();

  char* nodeFileName;
  if(argc < 4)
    nodeFileName = (char *) "../../../examples/data/pdp/nodes.csv";
  else
    nodeFileName = argv[3];
  createIloNodes(nodeFileName);

  char* vehiclesFileName;
  if (argc < 2)
    vehiclesFileName = (char *) "../../../examples/data/pdp/vehicles.csv";
  else
    vehiclesFileName = argv[1];
  createVehicles(vehiclesFileName);

  char* visitsFileName;
  if (argc < 3)
    visitsFileName =
      (char*) “../../../examples/data/pdp/visits.csv”;
  else
    visitsFileName = argv[2];
  createVisits(visitsFileName);
}

// Add dimensions
void RoutingModel::createDimensions() {
  IloDimension1 weight(_env, “Weight”);
  weight.setKey(“Weight”);
  _mdl.add(weight);
  IloDimension2 time(_env, IloEuclidean, “Time”);
  time.setKey(“Time”);
  _mdl.add(time);
  IloDimension2 distance(_env, IloEuclidean, IloFalse, “Distance”);
  distance.setKey(“Distance”);
  _mdl.add(distance);
}

// Create IloNodes
void RoutingModel::createIloNodes(char* nodeFileName) {
  IloCsvReader csvNodeReader(_env, nodeFileName);
  IloCsvReader::LineIterator  it(csvNodeReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char* name = line.getStringByHeader(“name”);
    IloNode node(_env,
                 line.getFloatByHeader(“x”),
                 line.getFloatByHeader(“y”),
                 0,
                 name);
    node.setKey(name);
    ++it;
  }
  csvNodeReader.end();
}

// Create vehicles
void RoutingModel::createVehicles(char* vehicleFileName) {

  IloDimension1 weight = IloDimension1::Find(_env, “Weight”);
  IloDimension2 time = IloDimension2::Find(_env, “Time”);
  IloDimension2 distance = IloDimension2::Find(_env, “Distance”);

  IloCsvReader csvVehicleReader(_env, vehicleFileName);
  IloCsvReader::LineIterator it(csvVehicleReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char * namefirst = line.getStringByHeader(“first”);
    char * namelast = line.getStringByHeader(“last”);
    char * name = line.getStringByHeader(“name”);
    IloNum capacity = line.getFloatByHeader(“capacity”);
    IloNum openTime = line.getFloatByHeader(“open”);
    IloNum closeTime = line.getFloatByHeader(“close”);
    IloNode node1 = IloNode::Find(_env, namefirst);
    IloNode node2 = IloNode::Find(_env, namelast);

    IloVisit first(node1, “depot”);
    _mdl.add(first.getCumulVar(weight) == 0);
    _mdl.add(first.getCumulVar(time) >= openTime);

    IloVisit last(node2, “depot”);
    _mdl.add(last.getCumulVar(time) <= closeTime);
    IloVehicle vehicle(first, last, name);
    vehicle.setCost(distance, 1.0);
    vehicle.setCapacity(weight, capacity);
    _mdl.add(vehicle);

    ++it;
  }
  csvVehicleReader.end();
}

// Create visits
void RoutingModel::createVisits(char* visitsFileName) {
  IloDimension1 weight = IloDimension1::Find(_env, “Weight”);
  IloDimension2 time = IloDimension2::Find(_env, “Time”);
  IloCsvReader csvVisitReader(_env, visitsFileName);
  IloCsvReader::LineIterator it(csvVisitReader);
  while(it.ok()){
    IloCsvLine line = *it;
    //read visit data from files
    char * pickupVisitName = line.getStringByHeader(“pickup”);
    char * pickupNodeName = line.getStringByHeader(“pickupNode”);
    char * deliveryVisitName = line.getStringByHeader(“delivery”);
    char * deliveryNodeName = line.getStringByHeader(“deliveryNode”);
    IloNum quantity = line.getFloatByHeader(“quantity”);
    IloNum pickupMinTime  = line.getFloatByHeader(“pickupMinTime”);
    IloNum pickupMaxTime  = line.getFloatByHeader(“pickupMaxTime”);
    IloNum deliveryMinTime  = line.getFloatByHeader(“deliveryMinTime”);
    IloNum deliveryMaxTime  = line.getFloatByHeader(“deliveryMaxTime”);
    IloNum dropTime = line.getFloatByHeader(“dropTime”);
    IloNum pickupTime = line.getFloatByHeader(“pickupTime”);

    // read data nodes from the file nodes.csv
    // and create pickup and delivery nodes

    IloNode pickupNode = IloNode::Find(_env, pickupNodeName);
    IloNode deliveryNode = IloNode::Find(_env, deliveryNodeName);

    //create and add pickup and delivery visits

    IloVisit pickup(pickupNode, pickupVisitName);
    _mdl.add(pickup.getDelayVar(time) == pickupTime);
    _mdl.add(pickup.getTransitVar(weight) == quantity);
    _mdl.add(pickupMinTime <= pickup.getCumulVar(time) <= pickupMaxTime);
    _mdl.add(pickup);

    IloVisit delivery(deliveryNode, deliveryVisitName);
    _mdl.add(delivery.getDelayVar(time) == dropTime);
    _mdl.add(delivery.getTransitVar(weight) == -quantity);
    _mdl.add(deliveryMinTime <= delivery.getCumulVar(time) <= deliveryMaxTime);
    _mdl.add(delivery);

    //add pickup and delivery order constraint
    _mdl.add(IloOrderedVisitPair(_env, pickup, delivery));
    ++it;
  }
  csvVisitReader.end();
}

////////////////////////////////////////////////////////////////////////////////
// Solving
class RoutingSolver {
  IloModel            _mdl;
  IloSolver           _solver;
  IloRoutingSolution  _solution;

  IloBool findFirstSolution(IloGoal goal);
  void greedyImprove(IloNHood nhood, IloGoal subGoal);
  void improve(IloGoal subgoal);
public:
  RoutingSolver(RoutingModel mdl);
  ~RoutingSolver() {}
  IloRoutingSolution getSolution() const { return _solution; }
  void printInformation() const;
  void solve();
};

RoutingSolver::RoutingSolver(RoutingModel mdl)
  : _mdl(mdl.getModel()), _solver(mdl.getModel()), _solution(mdl.getModel()) {}

// Solving : find first solution
IloBool RoutingSolver::findFirstSolution(IloGoal goal) {
  if (!_solver.solve(goal)) {
    _solver.error() << “Infeasible Routing Plan” << endl;
    return IloFalse;
  }
  IloDispatcher dispatcher(_solver);
  _solver.out() << dispatcher.getTotalCost() << endl;
  _solution.store(_solver);
  return IloTrue;
}

// Improve solution using nhood
void RoutingSolver::greedyImprove(IloNHood nhood, IloGoal subGoal) {
  _solver.out() << “Improving solution” << endl;
  IloEnv env = _solver.getEnv();
  nhood.reset();
  IloGoal improve = IloSingleMove(env,
                                  _solution,
                                  nhood,
                                  IloImprove(env),
                                  subGoal);
  while (_solver.solve(improve)) {}
}

// Improve solution
void RoutingSolver::improve(IloGoal subGoal) {
  IloEnv env = _solver.getEnv();
  IloNHood nhood = IloTwoOpt(env)
                 + IloOrOpt(env)
                 + IloRelocate(env)
                 + IloCross(env)
                 + IloExchange(env);
  greedyImprove(nhood, subGoal);
}

// Display Dispatcher information
void RoutingSolver::printInformation() const {
  IloDispatcher dispatcher(_solver);
  _solver.printInformation();
  dispatcher.printInformation();
  _solver.out() << "==============" << endl
                << "Cost         : " << dispatcher.getTotalCost() << endl
                << "Number of vehicles used : "
                << dispatcher.getNumberOfVehiclesUsed() << endl
                << "Solution     : " << endl
                << dispatcher << endl;
}

// Solving
void RoutingSolver::solve() {
  IloDispatcher dispatcher(_solver);
  IloEnv env = _solver.getEnv();
  // Subgoal
  IloGoal instantiateCost = IloDichotomize(env,
                                           dispatcher.getCostVar(),
                                           IloFalse);
  IloGoal restoreSolution = IloRestoreSolution(env, _solution);
  IloGoal goal = IloSavingsGenerate(env) && instantiateCost;

  // Solving
  if (findFirstSolution(goal)) {
    improve(instantiateCost);
    _solver.solve(restoreSolution);
  }
}

///////////////////////////////////////////////////////////////////////////////

int main(int argc, char * argv[]) {
  IloEnv env;
  try {
    RoutingModel mdl(env, argc, argv);
    RoutingSolver solver(mdl);
    solver.solve();
    solver.printInformation();
  } catch(IloException& ex) {
    cerr << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Complete Output

```
/**
1291.75
```

```
Improving solution
Number of fails              : 0
Number of choice points      : 0
Number of variables          : 2329
Number of constraints        : 410
Reversible stack (bytes)     : 197004
Solver heap (bytes)          : 1009300
Solver global heap (bytes)   : 144384
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11160
Total memory used (bytes)    : 1430260
Elapsed time since creation  : 0.02
Number of nodes              : 51
Number of visits             : 80
Number of vehicles           : 15
Number of dimensions         : 3
Number of accepted moves     : 23
===============
Cost         : 1078.88
Number of vehicles used : 9
Solution     :
Unperformed visits : None
vehicle1 :
 -> depot Weight[0] Time[0..19.1001] Distance[0..Inf) -> visit35 Weight[0..162]
Time[41.0366..60.1367] Distance[0..Inf) -> visit49 Weight[8..170]
Time[108.115..127.216] Distance[0..Inf) -> visit36 Weight[38..200]
Time[127.06..146.16] Distance[0..Inf) -> visit50 Weight[30..192]
Time[183.9..203] Distance[0..Inf) -> depot Weight[0..162] Time[210.87..230]
Distance[0..Inf)
vehicle2 :
 -> depot Weight[0] Time[0..18.7919] Distance[0..Inf) -> visit21 Weight[0..140]
Time[18.0278..36.8197] Distance[0..Inf) -> visit22 Weight[11..151]
Time[38.0278..56.8197] Distance[0..Inf) -> visit23 Weight[0..140] Time[68..78]
Distance[0..Inf) -> visit39 Weight[29..169] Time[86.6023..137.537]
Distance[0..Inf) -> visit24 Weight[60..200] Time[120.14..171.074]
Distance[0..Inf) -> visit40 Weight[31..171] Time[157.065..208] Distance[0..Inf)
-> depot Weight[0..140] Time[178.246..230] Distance[0..Inf)
vehicle3 :
 -> depot Weight[0] Time[0..18.5452] Distance[0..Inf) -> visit43 Weight[0..172]
Time[34.176..52.7212] Distance[0..Inf) -> visit44 Weight[7..179] Time[69..79]
Distance[0..Inf) -> visit17 Weight[0..172] Time[96.088..145.639]
Distance[0..Inf) -> visit5 Weight[2..174] Time[116.088..165.639]
Distance[0..Inf) -> visit18 Weight[28..200] Time[137.268..186.82]
Distance[0..Inf) -> visit6 Weight[26..198] Time[158.449..208] Distance[0..Inf)
-> depot Weight[0..172] Time[179.629..230] Distance[0..Inf)
vehicle4 :
 -> depot Weight[0] Time[0..13.1563] Distance[0..Inf) -> visit33 Weight[0..175]
Time[24.7588..37.9152] Distance[0..Inf) -> visit29 Weight[11..186]
Time[49.6249..62.7813] Distance[0..Inf) -> visit34 Weight[20..195]
Time[72.6633..85.8197] Distance[0..Inf) -> visit9 Weight[9..184] Time[97..107]
Distance[0..Inf) -> visit30 Weight[25..200] Time[122..174] Distance[0..Inf) ->
visit10 Weight[16..191] Time[142..194] Distance[0..Inf) -> depot Weight[0..175]
Time[177.495..230] Distance[0..Inf)
vehicle5 :
 -> depot Weight[0] Time[0..14.5761] Distance[0..Inf) -> visit31 Weight[0..144]
Time[17.4642..32.0404] Distance[0..Inf) -> visit19 Weight[27..171]
```

```
Time[45.3528..59.9289] Distance[0..Inf) -> visit11 Weight[44..188] Time[67..77]
Distance[0..Inf) -> visit32 Weight[56..200] Time[92.5242..143.816]
Distance[0..Inf) -> visit20 Weight[29..173] Time[113.295..164.586]
Distance[0..Inf) -> visit12 Weight[12..156] Time[153.708..205] Distance[0..Inf)
-> depot Weight[0..144] Time[178.708..230] Distance[0..Inf)
vehicle6 :
 -> depot Weight[0] Time[0..19.9727] Distance[0..Inf) -> visit41 Weight[0..164]
Time[28.8617..48.8345] Distance[0..Inf) -> visit15 Weight[5..169] Time[61..71]
Distance[0..Inf) -> visit13 Weight[13..177] Time[109..125.282] Distance[0..Inf)
-> visit42 Weight[36..200] Time[133.318..149.6] Distance[0..Inf) -> visit14
Weight[31..195] Time[152.537..168.82] Distance[0..Inf) -> visit16
Weight[8..172] Time[173.718..190] Distance[0..Inf) -> depot Weight[0..164]
Time[212.872..230] Distance[0..Inf)
vehicle7 :
 -> depot Weight[0] Time[0..40.5736] Distance[0..Inf) -> visit37 Weight[0..192]
Time[21.2132..61.7868] Distance[0..Inf) -> visit38 Weight[8..200] Time[83..93]
Distance[0..Inf) -> visit25 Weight[0..192] Time[172..175.639] Distance[0..Inf)
-> visit26 Weight[6..198] Time[204.361..208] Distance[0..Inf) -> depot
Weight[0..192] Time[225.541..230] Distance[0..Inf)
vehicle8 : Unused
vehicle9 :
 -> depot Weight[0] Time[0..61.5802] Distance[0..Inf) -> visit7 Weight[0..173]
Time[21.2132..82.7934] Distance[0..Inf) -> visit8 Weight[5..178] Time[95..105]
Distance[0..Inf) -> visit45 Weight[0..173] Time[111.403..134.827]
Distance[0..Inf) -> visit46 Weight[16..189] Time[132.173..155.597]
Distance[0..Inf) -> visit47 Weight[0..173] Time[152.173..175.597]
Distance[0..Inf) -> visit48 Weight[27..200] Time[168.577..192] Distance[0..Inf)
-> depot Weight[0..173] Time[206.379..230] Distance[0..Inf)
vehicle10 :
 -> depot Weight[0] Time[0..55.7316] Distance[0..Inf) -> visit27 Weight[0..177]
Time[5..60.7316] Distance[0..Inf) -> visit28 Weight[16..193]
Time[21.7082..77.4398] Distance[0..Inf) -> visit1 Weight[0..177]
Time[43.7082..99.4398] Distance[0..Inf) -> visit3 Weight[10..187]
Time[68.2684..124] Distance[0..Inf) -> visit4 Weight[23..200] Time[149..159]
Distance[0..Inf) -> visit2 Weight[10..187] Time[179.224..202] Distance[0..Inf)
-> depot Weight[0..177] Time[207.224..230] Distance[0..Inf)
vehicle11 : Unused
vehicle12 : Unused
vehicle13 : Unused
vehicle14 : Unused
vehicle15 : Unused
*/
```

C H A P T E R

8

Adding Vehicle Breaks

In this lesson, you will learn how to:

- use the class `IloVehicleBreakCon` to construct breaks such as coffee breaks, lunch breaks, and overnight breaks
- constrain breaks and instantiate breaks in the solution
- use the member function `setSpeed`

Describe

The vehicle routing problems presented so far have presented an ideal situation: continuous availability of vehicles. However, in real-world problems, the people driving these vehicles need to occasionally interrupt the trips. These interruptions might include coffee and lunch breaks, and if the trips are longer than one working day, overnight breaks. In Dispatcher, these interruptions are modeled as *vehicle breaks*.

A vehicle break is performed by a vehicle on a particular dimension (usually time), has bounds for the start time, and possibly a variable duration with bounds as well. Breaks can interrupt visits or not, but by default visits are unbreakable. Note that breaks themselves cannot be interrupted.

Breaks are created as constraints on vehicles. More than one break can be specified per vehicle, and breaks can be involved in metaconstraints. Constraints can be applied across a number of breaks to ensure that drivers take an appropriate amount of time off.

The problem considered in this lesson is a PDP problem. Pickups and deliveries are considered over a five-day period, with vehicle breaks taken each day for morning and afternoon coffee, and lunch. Overnight rest periods are modeled as breaks with a minimal number of rest hours to take between two consecutive night breaks.

Step 1 Describe the problem

The first step is to write a natural language description of the problem. A good way to start this process is to analyze the variables and objectives.

What are the constraints in this problem?

- ◆ Just as for a standard PDP, delivery trucks will pick up parcels en route and deliver them on the same route.
- ◆ The drivers of the vehicles require morning and afternoon coffee breaks, a lunch break, and an overnight break.

The objective is to minimize the cost of the delivery of all the parcels, while meeting all the break requirements.

Model

Once you have written a description of your problem, you can use Dispatcher classes to model it.

Step 2 Open the example file

Open the example file `YourDispatcherHome/examples/src/tutorial/breaks_partial.cpp` in your development environment.

As in the previous examples, you will use a `RoutingModel` class to call the functions that create the dimensions, nodes, vehicles, and visits. In this example, the `RoutingModel` class also calls functions that create and constrain the breaks.

Step 3 Declare the RoutingModel class

Add the following code after the comment `//Declare the RoutingModel class`

```
class RoutingModel {
  IloEnv _env;
  IloModel _mdl;

  void createDimensions();
  void createNodes(char* nodeFileName);
  void createVehicles(char* vehicleFileName);
  void createVisits(char* visitsFileName);
  void createVehicleBreaks(char* vehcileBreaksFileName);
  void createBreaksRelation(char* breaksRelationFileName);
```

Two new functions appear in `RoutingModel` to model breaks: `createVehicleBreaks` and `createBreaksRelation`.

The function `createVehicles` is very similar to previous examples, but introduces a new member function.

Step 4 Create the vehicles

Add the following code after the comment `//Create the vehicles`.

```
    IloVisit first(node1, "Depot");
    _mdl.add(first.getCumulVar(weight) == 0);
    _mdl.add(first.getCumulVar(time) >= line.getFloatByHeader("open"));

    IloVisit last(node2, "Depot");
    _mdl.add(last.getCumulVar(time) <= line.getFloatByHeader("close"));
    IloVehicle vehicle(first, last, name);
    vehicle.setCost(length, 1.0);
    vehicle.setSpeed(time, 30.0);
    vehicle.setCapacity(weight, capacity);
    vehicle.setKey(name);
    _mdl.add(vehicle);
```

The member function `IloVehicle::setSpeed` sets the speed of `vehicle` over the dimension `time` to be 30.0.

Next, you will create the vehicle breaks.

Step 5 Create the vehicle breaks

Add the following code after the comment `//Create the vehicle breaks`.

```
    IloVehicle vehicle = IloVehicle::Find(_env, vehicleName);
    for ( IloInt i = 0; i < nbDays; i++ ) {
      char name[100];
      IloNumVar breakStart(_env,(i * 24) + startLB, (i * 24) + startUB);
      IloNumVar breakDuration(_env, durationLB, durationUB);
      sprintf(name,"%s-day%ld",breakName,i + 1);
      IloVehicleBreakCon breaks(vehicle, time, breakStart, breakDuration,
name);
      breaks.setKey(name);
      _mdl.add(breaks);
```

This section of code iterates over all vehicle breaks, regardless of the break name (coffee, lunch, and so on), and adds them to the model. `startLB`, `startUB`, `durationLB`, and `durationUB` refer to the lower and upper bounds on the start and duration of each break. The various break starts and durations (with their associated bounds) are calculated over the 24 hours of the delivery days, and the resulting vehicle break constraint is then applied to `vehicle` with the constructor `IloVehicleBreakCon`.

The following code is the first section of `RoutingModel::createVehicleBreaks`, and is provided for you. This shows how the data is read and identified from the vehicle breaks input file.

```
void RoutingModel::createVehicleBreaks(char* vehicleBreaksFileName) {
  IloDimension2 time = IloDimension2::Find(_env, "Time");

  IloCsvReader csvVehicleBreaksReader(_env, vehicleBreaksFileName);
  IloCsvReader::LineIterator it(csvVehicleBreaksReader);
  while ( it.ok() ) {
    IloCsvLine line = *it;
    char* vehicleName = line.getStringByHeader("vehicle");
    char* breakName = line.getStringByHeader("breakName");
    IloNum startLB = line.getFloatByHeader("startLB");
    IloNum startUB = line.getFloatByHeader("startUB");
    IloNum durationLB = line.getFloatByHeader("durationLB");
    IloNum durationUB = line.getFloatByHeader("durationUB");
    IloInt nbDays = line.getIntByHeader("nbDays");
```

Step 6 Create the break constraints

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

```
    IloNumVar breakDuration(_env);
    IloVehicleBreakCon vehicleBreakCon1 = IloVehicleBreakCon::Find(_env,
break1);
    if ( strlen(break2) != 0 ) {
      IloVehicleBreakCon vehicleBreakCon2 = IloVehicleBreakCon::Find(_env,
break2);
      _mdl.add(breakDuration == vehicleBreakCon1.getDurationVar()
                              + vehicleBreakCon2.getDurationVar());
    }
    else {
      _mdl.add(breakDuration == vehicleBreakCon1.getStartVar()
                              + vehicleBreakCon1.getDurationVar());
    }
    if ( strcmp(constraint,"equal") == 0 )
      _mdl.add(breakDuration == totalDuration);
    if ( strcmp(constraint,"at least") == 0 )
      _mdl.add(breakDuration >= totalDuration);
    if ( strcmp(constraint,"at most") == 0 )
      _mdl.add(breakDuration <= totalDuration);
```

This code is a section of the function `RoutingModel::createBreaksRelation`. `break1`, `break2`, `totalDuration`, and `constraint` are read in from the breaks relation input file. `break1` and `break2` are added together, and the `totalDuration` is constrained to be `equal`, `at least`, or `at most` the amount of break time specified.

Solve

The solve portion of the example needs to add the breaks to the solution and instantiate them. You will add the breaks to the solution with a function called `addBreaks`.

Step 7 Create the RoutingSolver class

Add the following code after the comment `//Create the RoutingSolver class`.

```
class RoutingSolver {
  IloModel _mdl;
  IloEnv _env;
  IloSolver _solver;
  IloRoutingSolution _solution;
  IloRoutingSolution _breaks;

  IloBool findFirstSolution(IloGoal goal);
  void greedyImprove(IloNHood nhood, IloGoal subGoal);
  void improve(IloGoal subgoal);
  void addBreaks();
```

Notice the differences in `RoutingSolver` compared to the PDP version: there is a second `IloRoutingSolution` called `_breaks`. Additionally, a function `addBreaks` is present, which adds the breaks to `_breaks`.

Step 8 Add breaks to the _breaks routing solution

Add the following code after the comment
`//Add breaks to the _breaks routing solution`.

```
//add breaks to _breaks routing solution
void RoutingSolver::addBreaks() {
  IloVehicleBreakConIterator it(_mdl);
  while ( it.ok() ) {
    _breaks.add(*it);
    ++it;
  }
}
```

Next, the breaks must be instantiated to ensure that the start times and durations are set in the solution.

Step 9 Instantiate the vehicle breaks

Add the following code after the comment `//Instantiate the vehicle breaks`.

```
void RoutingSolver::solve() {
  IloDispatcher dispatcher(_solver);

  //add breaks to _breaks routing solution
  addBreaks();

  // Subgoals

  //The IloInstantiateVehicleBreaks goal uses a precision of 1 minutes
  //and treats all vehicles independently in the instantiation of
  //vehicle breaks
  IloGoal instantiateBreaks = IloLimitSearch(_env,
                                             IloInstantiateVehicleBreaks(_env,
1.0/60.0, IloTrue),
                                             IloFailLimit(_env, 100));

  IloGoal instantiateCost = IloDichotomize(_env,
                                           dispatcher.getCostVar(),
                                           IloFalse);

  IloGoal subGoal = instantiateBreaks && instantiateCost;
  IloGoal goal = IloSavingsGenerate(_env, instantiateBreaks) &&
instantiateCost;
  IloGoal restoreSolution = IloRestoreSolution(_env, _solution) &&
IloRestoreSolution(_env, _breaks);
```

The goal `IloInstantiateVehicleBreaks` instantiates the breaks of all vehicles (decides exactly where, when, and for how long they are taken) in the environment `_env`. Many other break instantiation goals are provided to increase flexibility and allow you to build your own goals, if desired (see the *IBM ILOG Dispatcher Reference Manual* for more information).

The parameter `1.0/60.0` ensures that a precision of one minute is used in calculating the break times, and `IloTrue` means that the goal will instantiate all the breaks in the routing plan, independent vehicle by independent vehicle.

The goal `instantiateBreaks` is created with an instance of `IloLimitSearch`. This ensures that the solution search will fail if no initial solution is found after 100 iterations. A goal is created to instantiate the cost variable for the dispatcher. A first solution is then sought. The goal `IloSavingsGenerate` builds that first solution. If a first solution is found, the solution is improved using neighborhoods. Note that in the proposed solution, only a simple greedy search is used. You might want to try improvement methods other than the basic `IloImprove`; for example `IloDispatcherGLS` or `IloDispatcherTabuSearch`.

If no first solution is found, the program ends.

Step 10 Compile and run the program

The solution improvement phase finds a solution using 6 vehicles with a cost of 1097.5 units:

```
 1127.84
Improving solution
Number of fails                : 0
Number of choice points        : 0
Number of variables            : 3364
Number of constraints          : 1118
Reversible stack (bytes)       : 554784
Solver heap (bytes)            : 2460520
Solver global heap (bytes)     : 156444
And stack (bytes)              : 20124
Or stack (bytes)               : 44244
Search Stack (bytes)           : 4044
Constraint queue (bytes)       : 17172
Total memory used (bytes)      : 3257332
Elapsed time since creation    : 0.04
Number of nodes                : 51
Number of visits               : 80
Number of vehicles             : 15
Number of dimensions           : 3
Number of accepted moves       : 4
===============
Cost         : 1097.5
Number of vehicles used : 6
```

The complete program and output are listed in "Complete Program" on page 202. You can also view it online in the `YourDispatcherHome/examples/src/breaks.cpp` file.

Review Exercises

For answers, see "Suggested Answers" on page 201.

1. What class is used to construct a vehicle break constraint?
2. Visits can be interrupted by breaks, but the default is that visits are not interruptible by breaks. Do you think that this option is controlled with the class that is used to construct a vehicle break constraint, or with the class that is used to model visits?

3. In the following code, the parameter `IloFalse` ensures that the goal instantiates all the breaks in the routing plan by considering all the vehicles during the instantiation process. If this parameter were changed to `IloTrue`, and thereby each vehicle were considered independently from the others, what do you think would be the likely impact on the instantiation processing time and to the goal failure rate?

```
IloGoal instantiateBreaks = IloLimitSearch(_env,
                                           IloInstantiateVehicleBreaks(_env,
                                                1.0/60.0, IloFalse),
                                           IloFailLimit(_env, 100));
```

Suggested Answers

Exercise 1

What class is used to construct a vehicle break constraint?

Suggested Answer

`IloVehicleBreakCon`.

Exercise 2

Visits can be interrupted by breaks, but the default is that visits are not interruptible by breaks. Do you think that this option is controlled with the class that is used to construct a vehicle break constraint, or with the class that is used to model visits?

Suggested Answer

This option is controlled with the class `IloVisit`, the class used to model visits. Using this class allows each visit to be individually controlled as to whether it can be interrupted or not.

Exercise 3

In the following code, the parameter `IloFalse` ensures that the goal instantiates all the breaks in the routing plan by considering all the vehicles during the instantiation process. If this parameter were changed to `IloTrue`, and thereby each vehicle were considered independently from the others, what do you think would be the likely impact on the instantiation processing time and to the goal failure rate?

```
IloGoal instantiateBreaks = IloLimitSearch(_env,
                                           IloInstantiateVehicleBreaks(_env,
                                                1.0/60.0, IloFalse),
                                           IloFailLimit(_env, 100));
```

Suggested Answer

The instantiation processing time would likely decrease and the goal failure rate would likely increase.

When set to `IloTrue`, this parameter means that if the break instantiation for a single vehicle is searched entirely and the break cannot be placed, then the goal fails. In other words, there is no attempt to backtrack to a previous vehicle to explore other break positions that may work in a successive attempt going forward.

Complete Program

The complete program follows. You can also view it online in the file `YourDispatcherHome/examples/src/breaks.cpp`.

```
// -------------------------------------------------------------- -*- C++ -*-
// File: examples/src/breaks.cpp
// --------------------------------------------------------------------------


#include <ildispat/ilodispatcher.h>

ILOSTLBEGIN

///////////////////////////////////////////////////////////////////////////////
// Modeling
class RoutingModel {
  IloEnv _env;
  IloModel _mdl;

  void createDimensions();
  void createNodes(char* nodeFileName);
  void createVehicles(char* vehicleFileName);
  void createVisits(char* visitsFileName);
  void createVehicleBreaks(char* vehcileBreaksFileName);
  void createBreaksRelation(char* breaksRelationFileName);

public:
  RoutingModel(IloEnv env,int argc,char* argv[]);
  ~RoutingModel() {
  }
  IloEnv getEnv() const {
    return _env;
  }
  IloModel getModel() const {
    return _mdl;
  }
};

RoutingModel::RoutingModel(IloEnv env,int argc,char* argv[]):_env(env),
_mdl(env) {
  createDimensions();
  char* nodeFileName;
  if ( argc < 4 )
    nodeFileName = (char *) “../../../examples/data/break/nodes.csv”;
  else
    nodeFileName = argv[3];
  createNodes(nodeFileName);

  char* vehiclesFileName;
  if ( argc < 2 )
    vehiclesFileName = (char *) “../../../examples/data/break/vehicles.csv”;
  else
    vehiclesFileName = argv[1];
  createVehicles(vehiclesFileName);

  char* visitsFileName;
  if ( argc < 3 )
    visitsFileName = (char *) “../../../examples/data/break/visits.csv”;
  else
    visitsFileName = argv[2];
  createVisits(visitsFileName);

  char* vehicleBreaksFileName;
  if ( argc < 5 )
    vehicleBreaksFileName = (char *) “../../../examples/data/break/
vehicleBreaks.csv”;
  else
    vehicleBreaksFileName = argv[4];
  createVehicleBreaks(vehicleBreaksFileName);

  char* breaksRelationFileName;
  if ( argc < 6 )
    breaksRelationFileName = (char *) “../../../examples/data/break/
breaksRelation.csv”;
  else
    breaksRelationFileName = argv[5];
  createBreaksRelation(breaksRelationFileName);
}

// Create dimensions
void RoutingModel::createDimensions() {
  IloDimension2 time(_env, IloEuclidean, “Time”);
  time.setKey(“Time”);
  _mdl.add(time);
  IloDimension2 length(_env, IloEuclidean, IloFalse, “Length”);
  length.setKey(“Length”);
  _mdl.add(length);
  IloDimension1 weight(_env, “Weight”);
  weight.setKey(“Weight”);
  _mdl.add(weight);
}

// Create nodes
void RoutingModel::createNodes(char* nodeFileName) {
  IloCsvReader csvNodeReader(_env, nodeFileName);
  IloCsvReader::LineIterator it(csvNodeReader);
  while ( it.ok() ) {
    IloCsvLine line = *it;
    char* name = line.getStringByHeader(“name”);
    IloNode node(_env, line.getFloatByHeader("x"), line.getFloatByHeader("y"),
0, name);
    node.setKey(name);
    ++it;
  }
  csvNodeReader.end();
}

// Create vehicles
void RoutingModel::createVehicles(char* vehicleFileName) {
  IloDimension1 weight = IloDimension1::Find(_env, "Weight");
  IloDimension2 time = IloDimension2::Find(_env, "Time");
  IloDimension2 length = IloDimension2::Find(_env, "Length");

  IloCsvReader csvVehicleReader(_env,vehicleFileName);
  IloCsvReader::LineIterator it(csvVehicleReader);
  while ( it.ok() ) {
    IloCsvLine line = *it;
    char* namefirst = line.getStringByHeader("first");
    char* namelast = line.getStringByHeader("last");
    char* name = line.getStringByHeader("name");
    IloNum capacity = line.getFloatByHeader("capacity");
    IloNode node1 = IloNode::Find(_env, namefirst);
    IloNode node2 = IloNode::Find(_env, namelast);

    IloVisit first(node1, "Depot");
    _mdl.add(first.getCumulVar(weight) == 0);
    _mdl.add(first.getCumulVar(time) >= line.getFloatByHeader("open"));

    IloVisit last(node2, "Depot");
    _mdl.add(last.getCumulVar(time) <= line.getFloatByHeader("close"));
    IloVehicle vehicle(first, last, name);
    vehicle.setCost(length, 1.0);
    vehicle.setSpeed(time, 30.0);
    vehicle.setCapacity(weight, capacity);
    vehicle.setKey(name);
    _mdl.add(vehicle);

    ++it;
  }
  csvVehicleReader.end();
}

// Create visits
void RoutingModel::createVisits(char* visitsFileName) {
  IloDimension1 weight = IloDimension1::Find(_env, "Weight");
  IloDimension2 time = IloDimension2::Find(_env, "Time");
  IloCsvReader csvVisitReader(_env,visitsFileName);
  IloCsvReader::LineIterator it(csvVisitReader);
  while ( it.ok() ) {
    IloCsvLine line = *it;
    //read visit data from files
    char* pickupVisitName = line.getStringByHeader("pickup");
    char* pickupNodeName = line.getStringByHeader("pickupNode");
    char* deliveryVisitName = line.getStringByHeader("delivery");
    char* deliveryNodeName = line.getStringByHeader("deliveryNode");
    IloNum quantity = line.getFloatByHeader("quantity");
    IloNum pickupMinTime = line.getFloatByHeader("pickupMinTime");
    IloNum pickupMaxTime = line.getFloatByHeader(“pickupMaxTime”);
    IloNum deliveryMinTime = line.getFloatByHeader(“deliveryMinTime”);
    IloNum deliveryMaxTime = line.getFloatByHeader(“deliveryMaxTime”);
    IloNum dropTime = line.getFloatByHeader(“dropTime”);
    IloNum pickupTime = line.getFloatByHeader(“pickupTime”);

    IloNode pickupNode = IloNode::Find(_env, pickupNodeName);
    IloNode deliveryNode = IloNode::Find(_env, deliveryNodeName);

    //create and add pickup and delivery visits
    IloVisit pickup(pickupNode, pickupVisitName);
    _mdl.add(pickup.getDelayVar(time) == pickupTime);
    _mdl.add(pickup.getTransitVar(weight) == quantity);
    _mdl.add(pickupMinTime <= pickup.getCumulVar(time) <= pickupMaxTime);
    _mdl.add(pickup);
    IloVisit delivery(deliveryNode, deliveryVisitName);
    _mdl.add(delivery.getDelayVar(time) == dropTime);
    _mdl.add(delivery.getTransitVar(weight) == -quantity);
    _mdl.add(deliveryMinTime <= delivery.getCumulVar(time) <= deliveryMaxTime);
    _mdl.add(delivery);
    //add pickup and delivery order constraint
    _mdl.add(IloOrderedVisitPair(_env, pickup, delivery));
    ++it;
  }
  csvVisitReader.end();
}

//create vehicle breaks
void RoutingModel::createVehicleBreaks(char* vehicleBreaksFileName) {
  IloDimension2 time = IloDimension2::Find(_env, “Time”);

  IloCsvReader csvVehicleBreaksReader(_env, vehicleBreaksFileName);
  IloCsvReader::LineIterator it(csvVehicleBreaksReader);
  while ( it.ok() ) {
    IloCsvLine line = *it;
    char* vehicleName = line.getStringByHeader(“vehicle”);
    char* breakName = line.getStringByHeader(“breakName”);
    IloNum startLB = line.getFloatByHeader(“startLB”);
    IloNum startUB = line.getFloatByHeader(“startUB”);
    IloNum durationLB = line.getFloatByHeader(“durationLB”);
    IloNum durationUB = line.getFloatByHeader(“durationUB”);
    IloInt nbDays = line.getIntByHeader(“nbDays”);

    IloVehicle vehicle = IloVehicle::Find(_env, vehicleName);
    for ( IloInt i = 0; i < nbDays; i++ ) {
      char name[100];
      IloNumVar breakStart(_env,(i * 24) + startLB, (i * 24) + startUB);
      IloNumVar breakDuration(_env, durationLB, durationUB);
      sprintf(name,”%s-day%ld”,breakName,i + 1);
      IloVehicleBreakCon breaks(vehicle, time, breakStart, breakDuration,
name);
      breaks.setKey(name);
      _mdl.add(breaks);

    }
    ++it;
  }
  csvVehicleBreaksReader.end();
```

```
}

//add breaks constraints
void RoutingModel::createBreaksRelation(char* breaksRelationFileName) {
  IloCsvReader csvBreaksRelationReader(_env, breaksRelationFileName);
  IloCsvReader::LineIterator it(csvBreaksRelationReader);
  while ( it.ok() ) {
    IloCsvLine line = *it;
    char* break1 = line.getStringByHeader(“break1”);
    char* break2 = line.getStringByHeader(“break2”);
    IloNum totalDuration = line.getFloatByHeader(“totalDuration”);
    char* constraint = line.getStringByHeader(“constraint”);

    IloNumVar breakDuration(_env);
    IloVehicleBreakCon vehicleBreakCon1 = IloVehicleBreakCon::Find(_env,
break1);
    if ( strlen(break2) != 0 ) {
      IloVehicleBreakCon vehicleBreakCon2 = IloVehicleBreakCon::Find(_env,
break2);
      _mdl.add(breakDuration == vehicleBreakCon1.getDurationVar()
                                + vehicleBreakCon2.getDurationVar());
    }
    else {
      _mdl.add(breakDuration == vehicleBreakCon1.getStartVar()
                                + vehicleBreakCon1.getDurationVar());
    }
    if ( strcmp(constraint,”equal”) == 0 )
      _mdl.add(breakDuration == totalDuration);
    if ( strcmp(constraint,”at least”) == 0 )
      _mdl.add(breakDuration >= totalDuration);
    if ( strcmp(constraint,”at most”) == 0 )
      _mdl.add(breakDuration <= totalDuration);


    ++it;
  }
  csvBreaksRelationReader.end();
}

////////////////////////////////////////////////////////////////////////////////
// Solving
class RoutingSolver {
  IloModel _mdl;
  IloEnv _env;
  IloSolver _solver;
  IloRoutingSolution _solution;
  IloRoutingSolution _breaks;

  IloBool findFirstSolution(IloGoal goal);
  void greedyImprove(IloNHood nhood, IloGoal subGoal);
  void improve(IloGoal subgoal);
  void addBreaks();

public:
  RoutingSolver(RoutingModel mdl);
  ~RoutingSolver() {
  }
  IloRoutingSolution getSolution() const {
    return _solution;
  }
  void printInformation() const;
  void solve();
};

RoutingSolver::RoutingSolver(RoutingModel mdl):_mdl(mdl.getModel()),
                                               _env(mdl.getEnv()),
                                               _solver(mdl.getModel()),
                                               _solution(mdl.getModel()),
                                               _breaks(mdl.getEnv()) {
}

// Solving : find first solution
IloBool RoutingSolver::findFirstSolution(IloGoal goal) {
  if ( !_solver.solve(goal) ) {
    _solver.error() << “Infeasible Routing Plan” << endl;
    return IloFalse;
  }
  IloDispatcher dispatcher(_solver);
  _solver.out() << dispatcher.getTotalCost() << endl;
  _solution.store(_solver);
  _breaks.store(_solver);
  return IloTrue;
}

// Improve solution using nhood
void RoutingSolver::greedyImprove(IloNHood nhood, IloGoal subGoal) {
  _solver.out() << “Improving solution” << endl;
  IloEnv env = _solver.getEnv();
  nhood.reset();
  IloGoal improve = IloSingleMove(env,
                                  _solution,
                                  nhood,
                                  IloImprove(env),
                                  subGoal);
  while ( _solver.solve(improve) )
    _breaks.store(_solver);
}

// Improve solution
void RoutingSolver::improve(IloGoal subGoal) {
  IloNHood nhood = IloTwoOpt(_env)
                 + IloOrOpt(_env)
                 + IloRelocate(_env)
                 + IloCross(_env)
                 + IloExchange(_env);
  greedyImprove(nhood,subGoal);
}

//add breaks to _breaks routing solution
void RoutingSolver::addBreaks() {
  IloVehicleBreakConIterator it(_mdl);
  while ( it.ok() ) {
    _breaks.add(*it);
    ++it;
  }
}
```

```
// Display Dispatcher information
void RoutingSolver::printInformation() const {
  IloDispatcher dispatcher(_solver);
  _solver.printInformation();
  dispatcher.printInformation();
  _solver.out() << "==============" << endl
                << "Cost         : " << dispatcher.getTotalCost() << endl
                << "Number of vehicles used : " <<
dispatcher.getNumberOfVehiclesUsed() << endl
                << "Solution     : " << endl
                << IloVerbose(dispatcher) << endl;
}

// Solving
void RoutingSolver::solve() {
  IloDispatcher dispatcher(_solver);

  //add breaks to _breaks routing solution
  addBreaks();

  // Subgoals

  //The IloInstantiateVehicleBreaks goal uses a precision of 1 minutes
  //and treats all vehicles independently in the instantiation of vehicle
breaks
  IloGoal instantiateBreaks = IloLimitSearch(_env,
                                             IloInstantiateVehicleBreaks(_env,
1.0/60.0, IloTrue),
                                             IloFailLimit(_env, 100));

  IloGoal instantiateCost = IloDichotomize(_env,
                                           dispatcher.getCostVar(),
                                           IloFalse);

  IloGoal subGoal = instantiateBreaks && instantiateCost;
  IloGoal goal = IloSavingsGenerate(_env, instantiateBreaks) &&
instantiateCost;
  IloGoal restoreSolution = IloRestoreSolution(_env, _solution) &&
IloRestoreSolution(_env, _breaks);

  // Solving
  if ( findFirstSolution(goal) ) {
    improve(subGoal);
    _solver.solve(restoreSolution);
  }
}

///////////////////////////////////////////////////////////////////////////////
int main(int argc,char* argv[]) {
  IloEnv env;
  try {
    RoutingModel mdl(env,argc,argv);
    RoutingSolver solver(mdl);
    solver.solve();
    solver.printInformation();
  }
  catch ( IloException& ex ) {
    cerr << “Error: “ << ex << endl;
  }
  env.end();
  return 0;
}
```

Complete Output

```
//////////////////////////////////////////////////////////////////////
// Output

/**
1127.84
Improving solution
Number of fails              : 0
Number of choice points      : 0
Number of variables          : 3364
Number of constraints        : 1118
Reversible stack (bytes)     : 554784
Solver heap (bytes)          : 2460520
Solver global heap (bytes)   : 156444
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 17172
Total memory used (bytes)    : 3257332
Elapsed time since creation  : 0.04
Number of nodes              : 51
Number of visits             : 80
Number of vehicles           : 15
Number of dimensions         : 3
Number of accepted moves     : 4
===============
Cost          : 1097.5
Number of vehicles used : 6
Solution      :
Unperformed visits : None

vehicle1
Cost coefficients : Length[1]
   Route : Depot -> visit33 -> visit9 -> visit34 -> visit35 -> visit10 ->
visit19 -> visit20 -> visit36 -> Depot
    Time : Depot [9..9.21235], delay [0..0.212353] -> travel [0.825295], wait
[0..0.212353] -> visit33 [9.82529..10.0376], delay [0.25] -> travel [0.274874],
wait [52.4375..52.7135] -> visit9 [63..63.0637], delay [0.25] -> travel
[0.372678], wait [0..0.063661] -> visit34 [63.6227..63.6863], delay [0.25] ->
travel [0.339935], wait [19.7237..19.9248] -> visit35 [84..84.1374], delay
[0.25] -> travel [1.11255], wait [0.5..0.637445] -> visit10 [85.8626..86],
delay [0.25] -> travel [0.5], wait [0..0.637445] -> visit19 [86.6126..87.25],
delay [0.25] -> travel [1.01379], wait [0.25..2.87365] -> visit20
[88.1263..90.75], delay [0.25] -> travel [1.44299], wait [11.557..14.8006] ->
visit36 [104..104.62], delay [0.25] -> travel [1.38002], wait [0..0.619984] ->
Depot [105.63..106.25], delay [0..0.619984] -> travel [0], wait [1..Inf)
TransitSum [102.522..Inf)
  Length : Depot [0..Inf), delay [0..Inf) -> travel [24.7588], wait [0..Inf) ->
visit33 [0..Inf), delay [0..Inf) -> travel [8.24621], wait [0..Inf) -> visit9
```

```
[0..Inf), delay [0..Inf) -> travel [11.1803], wait [0..Inf) -> visit34
[0..Inf), delay [0..Inf) -> travel [10.198], wait [0..Inf) -> visit35 [0..Inf),
delay [0..Inf) -> travel [33.3766], wait [0..Inf) -> visit10 [0..Inf), delay
[0..Inf) -> travel [15], wait [0..Inf) -> visit19 [0..Inf), delay [0..Inf) ->
travel [30.4138], wait [0..Inf) -> visit20 [0..Inf), delay [0..Inf) -> travel
[43.2897], wait [0..Inf) -> visit36 [0..Inf), delay [0..Inf) -> travel
[41.4005], wait [0..Inf) -> Depot [0..Inf), delay [0..Inf) -> travel [0], wait
[0..Inf) TransitSum [217.864..Inf)
  Weight : Depot [0], quantity [0..173] -> visit33 [0..173], quantity [11] ->
visit9 [11..184], quantity [16] -> visit34 [27..200], quantity [-11] -> visit35
[16..189], quantity [8] -> visit10 [24..197], quantity [-16] -> visit19
[8..181], quantity [17] -> visit20 [25..198], quantity [-17] -> visit36
[8..181], quantity [-8] -> Depot [0..173], quantity (-Inf..Inf) TransitSum (-
Inf..Inf)
  Breaks : lunch1-day1 after visit33, Time : start [12.2383..12.25], duration
[0.5]
           lunch1-day2 after visit33, Time : start [36.2383..36.25], duration
[0.5]
           lunch1-day3 after visit33, Time : start [60.2383..60.25], duration
[0.5]
           lunch1-day4 after visit35, Time : start [84.6133..84.625], duration
[0.5]
           lunch1-day5 after Depot, Time : start [108.238..108.25], duration
[0.5]
           morningCoffee1-day1 after visit33, Time : start [10.2744..10.2876],
duration [0.25]
           morningCoffee1-day2 after visit33, Time : start [34.2344..34.25],
duration [0.25]
           morningCoffee1-day3 after visit33, Time : start [58.2344..58.25],
duration [0.25]
           morningCoffee1-day4 after visit34, Time : start [82.2344..82.25],
duration [0.25]
           morningCoffee1-day5 after Depot, Time : start [106.234..106.25],
duration [0.25]
           aftenoonCoffee1-day1 after visit33, Time : start [15.4844..15.5],
duration [0.25]
           aftenoonCoffee1-day2 after visit33, Time : start [39.4844..39.5],
duration [0.25]
           aftenoonCoffee1-day3 after visit34, Time : start [63.9204..63.9363],
duration [0.25]
           aftenoonCoffee1-day4 after visit19, Time : start [87.4844..87.5],
duration [0.25]
           aftenoonCoffee1-day5 after Depot, Time : start [111.484..111.5],
duration [0.25]
           night1-day1 after visit33, Time : start [18.9844..19], duration [7]
           night1-day2 after visit33, Time : start [42.9844..43], duration [10]
           night1-day3 after visit34, Time : start [66.9844..67], duration [7]
           night1-day4 after visit20, Time : start [90.9844..91], duration [10]

vehicle2
Cost coefficients : Length[1]
   Route : Depot -> visit1 -> visit3 -> visit2 -> visit41 -> visit4 -> visit13
-> visit37 -> visit14 -> visit43 -> visit38 -> visit44 -> visit42 -> Depot
    Time : Depot [9..9.49228], delay [0..0.492282] -> travel [0.507718], wait
[0..0.492282] -> visit1 [9.50772..10], delay [0.25] -> travel [0.485341], wait
[20.2647..22.36] -> visit3 [31..32.603], delay [0.25] -> travel [1.14698], wait
[0..1.60302] -> visit2 [32.397..34], delay [0.25] -> travel [0.406885], wait
[23.3431..25.0711] -> visit41 [58..58.125], delay [0.25] -> travel [0.612826],
wait [0.25..0.393669] -> visit4 [59.1128..59.2565], delay [0.25] -> travel
[0.849837], wait [0..0.143669] -> visit13 [60.2127..60.3563], delay [0.25] ->
travel [0.372678], wait [21.521..22.542] -> visit37 [82.5..83.3773], delay
[0.25] -> travel [0.372678], wait [0..0.877322] -> visit14 [83.1227..84], delay
[0.25] -> travel [0.354338], wait [2.89566..3.89798] -> visit43 [87.5..87.625],
delay [0.25] -> travel [0.603692], wait [16.5213..17.3916] -> visit38
[105..105.745], delay [0.25] -> travel [0.360555], wait [0.25..0.995311] ->
visit44 [106.417..106.606], delay [0.25] -> travel [0.438432], wait
[0..0.189054] -> visit42 [107.105..107.294], delay [0.25] -> travel [0.849837],
wait [0..0.189054] -> Depot [108.205..108.394], delay [0..0.189054] -> travel
[0], wait [0.75..Inf) TransitSum [102.242..Inf)
  Length : Depot [0..Inf), delay [0..Inf) -> travel [15.2315], wait [0..Inf) ->
visit1 [0..Inf), delay [0..Inf) -> travel [14.5602], wait [0..Inf) -> visit3
[0..Inf), delay [0..Inf) -> travel [34.4093], wait [0..Inf) -> visit2 [0..Inf),
delay [0..Inf) -> travel [12.2066], wait [0..Inf) -> visit41 [0..Inf), delay
[0..Inf) -> travel [18.3848], wait [0..Inf) -> visit4 [0..Inf), delay [0..Inf)
-> travel [25.4951], wait [0..Inf) -> visit13 [0..Inf), delay [0..Inf) ->
travel [11.1803], wait [0..Inf) -> visit37 [0..Inf), delay [0..Inf) -> travel
[11.1803], wait [0..Inf) -> visit14 [0..Inf), delay [0..Inf) -> travel
[10.6301], wait [0..Inf) -> visit43 [0..Inf), delay [0..Inf) -> travel
[18.1108], wait [0..Inf) -> visit38 [0..Inf), delay [0..Inf) -> travel
[10.8167], wait [0..Inf) -> visit44 [0..Inf), delay [0..Inf) -> travel
[13.1529], wait [0..Inf) -> visit42 [0..Inf), delay [0..Inf) -> travel
[25.4951], wait [0..Inf) -> Depot [0..Inf), delay [0..Inf) -> travel [0], wait
[0..Inf) TransitSum [220.854..Inf)
  Weight : Depot [0], quantity [0..164] -> visit1 [0..164], quantity [10] ->
visit3 [10..174], quantity [13] -> visit2 [23..187], quantity [-10] -> visit41
[13..177], quantity [5] -> visit4 [18..182], quantity [-13] -> visit13
[5..169], quantity [23] -> visit37 [28..192], quantity [8] -> visit14
[36..200], quantity [-23] -> visit43 [13..177], quantity [7] -> visit38
[20..184], quantity [-8] -> visit44 [12..176], quantity [-7] -> visit42
[5..169], quantity [-5] -> Depot [0..164], quantity (-Inf..Inf) TransitSum (-
Inf..Inf)
  Breaks : lunch2-day1 after visit1, Time : start [12.2383..12.25], duration
[0.5]
           lunch2-day2 after visit2, Time : start [36.2383..36.25], duration
[0.5]
           lunch2-day3 after visit13, Time : start [60.594..60.6063], duration
[0.5]
           lunch2-day4 after visit14, Time : start [84.2383..84.25], duration
[0.5]
           lunch2-day5 after Depot, Time : start [108.385..108.394], duration
[0.5]
           morningCoffee2-day1 after visit1, Time : start [10.2344..10.25],
duration [0.25]
           morningCoffee2-day2 after visit2, Time : start [34.2344..34.25],
duration [0.25]
           morningCoffee2-day3 after visit41, Time : start [58.3594..58.375],
duration [0.25]
           morningCoffee2-day4 after visit13, Time : start [82.2344..82.25],
duration [0.25]
           morningCoffee2-day5 after visit38, Time : start [106.167..106.178],
duration [0.25]
           aftenoonCoffee2-day1 after visit1, Time : start [15.4844..15.5],
duration [0.25]
           aftenoonCoffee2-day2 after visit2, Time : start [39.4844..39.5],
duration [0.25]
           aftenoonCoffee2-day3 after visit13, Time : start [63.4844..63.5],
duration [0.25]
           aftenoonCoffee2-day4 after visit43, Time : start [87.8594..87.875],
duration [0.25]
           aftenoonCoffee2-day5 after Depot, Time : start [111.484..111.5],
duration [0.25]
           night2-day1 after visit1, Time : start [18.9844..19], duration [7]
           night2-day2 after visit2, Time : start [42.9844..43], duration [10]
           night2-day3 after visit13, Time : start [66.9844..67], duration [7]
           night2-day4 after visit43, Time : start [90.9844..91], duration [10]

vehicle3
Cost coefficients : Length[1]
   Route : Depot -> visit47 -> visit48 -> visit11 -> visit49 -> visit50 ->
visit12 -> Depot
    Time : Depot [9..10.25], delay [0..1.25] -> travel [1.14018], wait
[1.85982..5.10982] -> visit47 [14.5..15.25], delay [0.25] -> travel [0.213437],
wait [23.2866..24.0444] -> visit48 [39..39.0078], delay [0.25] -> travel
[0.492161], wait [0..0.00783923] -> visit11 [39.7422..39.75], delay [0.25] ->
travel [0.477261], wait [41.5227..41.6556] -> visit49 [82..82.125], delay
[0.25] -> travel [1.53551], wait [22.5895..23.4272] -> visit50
[106.5..107.213], delay [0.25] -> travel [0.412311], wait [0..0.712689] ->
visit12 [107.5..107.875], delay [0.25] -> travel [0.5], wait [0..0.375] ->
Depot [108.25..108.625], delay [0..0.375] -> travel [0], wait [0.75..Inf)
TransitSum [101.484..Inf)
  Length : Depot [0..Inf), delay [0..Inf) -> travel [34.2053], wait [0..Inf) ->
visit47 [0..Inf), delay [0..Inf) -> travel [6.40312], wait [0..Inf) -> visit48
[0..Inf), delay [0..Inf) -> travel [14.7648], wait [0..Inf) -> visit11
[0..Inf), delay [0..Inf) -> travel [14.3178], wait [0..Inf) -> visit49
[0..Inf), delay [0..Inf) -> travel [46.0652], wait [0..Inf) -> visit50
[0..Inf), delay [0..Inf) -> travel [12.3693], wait [0..Inf) -> visit12
[0..Inf), delay [0..Inf) -> travel [15], wait [0..Inf) -> Depot [0..Inf), delay
[0..Inf) -> travel [0], wait [0..Inf) TransitSum [143.126..Inf)
  Weight : Depot [0], quantity [0..158] -> visit47 [0..158], quantity [27] ->
visit48 [27..185], quantity [-27] -> visit11 [0..158], quantity [12] -> visit49
[12..170], quantity [30] -> visit50 [42..200], quantity [-30] -> visit12
[12..170], quantity [-12] -> Depot [0..158], quantity (-Inf..Inf) TransitSum (-
Inf..Inf)
  Breaks : lunch3-day1 after Depot, Time : start [12.2383..12.25], duration
[0.5]
           lunch3-day2 after visit47, Time : start [36.2383..36.25], duration
[0.5]
           lunch3-day3 after visit11, Time : start [60.2383..60.25], duration
[0.5]
           lunch3-day4 after visit49, Time : start [84.2383..84.25], duration
[0.5]
           lunch3-day5 after Depot, Time : start [108.613..108.625], duration
[0.5]
           morningCoffee3-day1 after Depot, Time : start [10.2344..10.25],
duration [0.25]
           morningCoffee3-day2 after visit47, Time : start [34.2344..34.25],
duration [0.25]
           morningCoffee3-day3 after visit11, Time : start [58.2344..58.25],
duration [0.25]
           morningCoffee3-day4 after visit49, Time : start [82.3594..82.375],
duration [0.25]
           morningCoffee3-day5 after visit49, Time : start [106.234..106.25],
duration [0.25]
           aftenoonCoffee3-day1 after visit47, Time : start [15.4844..15.5],
duration [0.25]
           aftenoonCoffee3-day2 after visit11, Time : start [39.9922..40],
duration [0.25]
           aftenoonCoffee3-day3 after visit11, Time : start [63.4844..63.5],
duration [0.25]
           aftenoonCoffee3-day4 after visit49, Time : start [87.4844..87.5],
duration [0.25]
           aftenoonCoffee3-day5 after Depot, Time : start [111.484..111.5],
duration [0.25]
           night3-day1 after visit47, Time : start [18.9844..19], duration [7]
           night3-day2 after visit11, Time : start [42.9844..43], duration [10]
           night3-day3 after visit11, Time : start [66.9844..67], duration [7]
           night3-day4 after visit49, Time : start [90.9844..91], duration [10]

vehicle4
Cost coefficients : Length[1]
   Route : Depot -> visit15 -> visit16 -> visit5 -> visit6 -> visit17 ->
visit23 -> visit39 -> visit25 -> visit24 -> visit29 -> visit40 -> visit18 ->
visit30 -> visit26 -> visit21 -> visit22 -> Depot
    Time : Depot [9..10.25], delay [0..1.25] -> travel [1.01379], wait
[0.75..5.23621] -> visit15 [13..15.25], delay [0.25] -> travel [0.833333], wait
[0.25..3.29399] -> visit16 [16..17.3773], delay [0.25] -> travel [0.372678],
wait [0..1.37732] -> visit5 [16.6227..18], delay [0.25] -> travel [0.333333],
wait [16.4167..18.794] -> visit6 [35..36], delay [0.25] -> travel [0.666667],
wait [1.08333..3.08333] -> visit17 [38..39], delay [0.25] -> travel [1.86339],
wait [14.8866..17.8866] -> visit23 [56..58], delay [0.25] -> travel [0.286744],
wait [2.96326..6.71326] -> visit39 [61.5..63.25], delay [0.25] -> travel
[0.314466], wait [15.1855..19.1855] -> visit25 [79..81.25], delay [0.25] ->
travel [0.5], wait [0..2.25] -> visit24 [81..82], delay [0.25] -> travel
[0.235702], wait [2.25258..3.60159] -> visit29 [84.7383..85.0873], delay [0.25]
-> travel [0.980363], wait [0..0.349011] -> visit40 [85.9686..86.3177], delay
[0.25] -> travel [0.833333], wait [0..0.349011] -> visit18 [87.052..87.401],
delay [0.25] -> travel [0.942809], wait [0.25..2.50521] -> visit30
[88.4948..90.75], delay [0.25] -> travel [1.01379], wait [13.9862..16.3664] ->
visit26 [106..106.125], delay [0.25] -> travel [0.333333], wait
[0.25..0.416667] -> visit21 [106.833..107], delay [0.25] -> travel [0.333333],
wait [2.41667..3.00957] -> visit22 [110..110.426], delay [0.25] -> travel
[0.897527], wait [0..0.426236] -> Depot [111.148..111.574], delay [0..0.426236]
-> travel [0], wait [0.25..Inf) TransitSum [101.56..Inf)
  Length : Depot [0..Inf), delay [0..Inf) -> travel [30.4138], wait [0..Inf) ->
visit15 [0..Inf), delay [0..Inf) -> travel [25], wait [0..Inf) -> visit16
[0..Inf), delay [0..Inf) -> travel [11.1803], wait [0..Inf) -> visit5 [0..Inf),
delay [0..Inf) -> travel [10], wait [0..Inf) -> visit6 [0..Inf), delay [0..Inf)
-> travel [20], wait [0..Inf) -> visit17 [0..Inf), delay [0..Inf) -> travel
[55.9017], wait [0..Inf) -> visit23 [0..Inf), delay [0..Inf) -> travel
[8.60233], wait [0..Inf) -> visit39 [0..Inf), delay [0..Inf) -> travel
[9.43398], wait [0..Inf) -> visit25 [0..Inf), delay [0..Inf) -> travel [15],
wait [0..Inf) -> visit24 [0..Inf), delay [0..Inf) -> travel [7.07107], wait
[0..Inf) -> visit29 [0..Inf), delay [0..Inf) -> travel [29.4109], wait [0..Inf)
-> visit40 [0..Inf), delay [0..Inf) -> travel [25], wait [0..Inf) -> visit18
[0..Inf), delay [0..Inf) -> travel [28.2843], wait [0..Inf) -> visit30
[0..Inf), delay [0..Inf) -> travel [30.4138], wait [0..Inf) -> visit26
[0..Inf), delay [0..Inf) -> travel [10], wait [0..Inf) -> visit21 [0..Inf),
delay [0..Inf) -> travel [10], wait [0..Inf) -> visit22 [0..Inf), delay
[0..Inf) -> travel [26.9258], wait [0..Inf) -> Depot [0..Inf), delay [0..Inf) -
> travel [0], wait [0..Inf) TransitSum [352.638..Inf)
  Weight : Depot [0], quantity [0..132] -> visit15 [0..132], quantity [8] ->
visit16 [8..140], quantity [-8] -> visit5 [0..132], quantity [26] -> visit6
```

```
[26..158], quantity [-26] -> visit17 [0..132], quantity [2] -> visit23
[2..134], quantity [29] -> visit39 [31..163], quantity [31] -> visit25
[62..194], quantity [6] -> visit24 [68..200], quantity [-29] -> visit29
[39..171], quantity [9] -> visit40 [48..180], quantity [-31] -> visit18
[17..149], quantity [-2] -> visit30 [15..147], quantity [-9] -> visit26
[6..138], quantity [-6] -> visit21 [0..132], quantity [11] -> visit22
[11..143], quantity [-11] -> Depot [0..132], quantity (-Inf..Inf) TransitSum (-
Inf..Inf)
  Breaks : lunch4-day1 after Depot, Time : start [12.2383..12.25], duration
[0.5]
           lunch4-day2 after visit6, Time : start [36.2383..36.25], duration
[0.5]
           lunch4-day3 after visit23, Time : start [60.2383..60.25], duration
[0.5]
           lunch4-day4 after visit24, Time : start [84.2383..84.25], duration
[0.5]
           lunch4-day5 after visit21, Time : start [108.238..108.25], duration
[0.5]
           morningCoffee4-day1 after Depot, Time : start [10.2344..10.25],
duration [0.25]
           morningCoffee4-day2 after visit5, Time : start [34.2344..34.25],
duration [0.25]
           morningCoffee4-day3 after visit23, Time : start [58.2344..58.25],
duration [0.25]
           morningCoffee4-day4 after visit24, Time : start [82.2344..82.25],
duration [0.25]
           morningCoffee4-day5 after visit26, Time : start [106.359..106.375],
duration [0.25]
           aftenoonCoffee4-day1 after visit15, Time : start [15.4844..15.5],
duration [0.25]
           aftenoonCoffee4-day2 after visit17, Time : start [39.4844..39.5],
duration [0.25]
           aftenoonCoffee4-day3 after visit39, Time : start [63.4844..63.5],
duration [0.25]
           aftenoonCoffee4-day4 after visit18, Time : start [87.6401..87.651],
duration [0.25]
           aftenoonCoffee4-day5 after Depot, Time : start [111.56..111.574],
duration [0.25]
           night4-day1 after visit5, Time : start [18.9844..19], duration [7]
           night4-day2 after visit17, Time : start [42.9844..43], duration [10]
           night4-day3 after visit39, Time : start [66.9844..67], duration [7]
           night4-day4 after visit30, Time : start [90.9844..91], duration [10]

vehicle5
Cost coefficients : Length[1]
   Route : Depot -> visit7 -> visit8 -> visit45 -> visit46 -> Depot
    Time : Depot [9..10.25], delay [0..1.25] -> travel [0.707107], wait
[3.29289..5.91789] -> visit7 [15.5..15.625], delay [0.25] -> travel [0.406885],
wait [41.7181..41.9681] -> visit8 [58..58.125], delay [0.25] -> travel
[0.213437], wait [1.91156..2.16156] -> visit45 [60.5..60.625], delay [0.25] ->
travel [0.359011], wait [24.766..25.1749] -> visit46 [86..86.2839], delay
[0.25] -> travel [1.18228], wait [0..0.283862] -> Depot [87.4323..87.7161],
delay [0..0.283862] -> travel [0], wait [11.25..Inf) TransitSum [101.484..Inf)
  Length : Depot [0..Inf), delay [0..Inf) -> travel [21.2132], wait [0..Inf) ->
visit7 [0..Inf), delay [0..Inf) -> travel [12.2066], wait [0..Inf) -> visit8
[0..Inf), delay [0..Inf) -> travel [6.40312], wait [0..Inf) -> visit45
[0..Inf), delay [0..Inf) -> travel [10.7703], wait [0..Inf) -> visit46
[0..Inf), delay [0..Inf) -> travel [35.4683], wait [0..Inf) -> Depot [0..Inf),
delay [0..Inf) -> travel [0], wait [0..Inf) TransitSum [86.0615..Inf)
  Weight : Depot [0], quantity [0..184] -> visit7 [0..184], quantity [5] ->
visit8 [5..189], quantity [-5] -> visit45 [0..184], quantity [16] -> visit46
[16..200], quantity [-16] -> Depot [0..184], quantity (-Inf..Inf) TransitSum (-
Inf..Inf)
  Breaks : lunch5-day1 after Depot, Time : start [12.2383..12.25], duration
[0.5]
           lunch5-day2 after visit7, Time : start [36.2383..36.25], duration
[0.5]
           lunch5-day3 after visit45, Time : start [60.8594..60.875], duration
[0.5]
           lunch5-day4 after visit45, Time : start [84.2383..84.25], duration
[0.5]
           lunch5-day5 after Depot, Time : start [108.238..108.25], duration
[0.5]
           morningCoffee5-day1 after Depot, Time : start [10.2344..10.25],
duration [0.25]
           morningCoffee5-day2 after visit7, Time : start [34.2344..34.25],
duration [0.25]
           morningCoffee5-day3 after visit8, Time : start [58.3594..58.375],
duration [0.25]
           morningCoffee5-day4 after visit45, Time : start [82.2344..82.25],
duration [0.25]
           morningCoffee5-day5 after Depot, Time : start [106.234..106.25],
duration [0.25]
           aftenoonCoffee5-day1 after visit7, Time : start [15.8594..15.875],
duration [0.25]
           aftenoonCoffee5-day2 after visit7, Time : start [39.4844..39.5],
duration [0.25]
           aftenoonCoffee5-day3 after visit45, Time : start [63.4844..63.5],
duration [0.25]
           aftenoonCoffee5-day4 after Depot, Time : start [87.7073..87.7161],
duration [0.25]
           aftenoonCoffee5-day5 after Depot, Time : start [111.484..111.5],
duration [0.25]
           night5-day1 after visit7, Time : start [18.9844..19], duration [7]
           night5-day2 after visit7, Time : start [42.9844..43], duration [10]
           night5-day3 after visit45, Time : start [66.9844..67], duration [7]
           night5-day4 after Depot, Time : start [90.9844..91], duration [10]

vehicle6
Cost coefficients : Length[1]
   Route : Depot -> visit31 -> visit32 -> visit27 -> visit28 -> Depot
    Time : Depot [9..10.25], delay [0..1.25] -> travel [0.582142], wait
[25.9179..29.6679] -> visit31 [38..39.25], delay [0.25] -> travel [0.582142],
wait [18.9179..21.1679] -> visit32 [59..60], delay [0.25] -> travel [0.966667],
wait [44.7833..45.9083] -> visit27 [106..106.125], delay [0.25] -> travel
[0.223607], wait [3.40139..4.56557] -> visit28 [110..111.039], delay [0.25] ->
travel [0.210819], wait [0..1.03918] -> Depot [110.461..111.5], delay
[0..1.03918] -> travel [0], wait [0.25..Inf) TransitSum [101.484..Inf)
  Length : Depot [0..Inf), delay [0..Inf) -> travel [17.4642], wait [0..Inf) ->
visit31 [0..Inf), delay [0..Inf) -> travel [17.4642], wait [0..Inf) -> visit32
[0..Inf), delay [0..Inf) -> travel [29], wait [0..Inf) -> visit27 [0..Inf),
delay [0..Inf) -> travel [6.7082], wait [0..Inf) -> visit28 [0..Inf), delay
[0..Inf) -> travel [6.32456], wait [0..Inf) -> Depot [0..Inf), delay [0..Inf) -
> travel [0], wait [0..Inf) TransitSum [76.9613..Inf)
  Weight : Depot [0], quantity [0..173] -> visit31 [0..173], quantity [27] ->
visit32 [27..200], quantity [-27] -> visit27 [0..173], quantity [16] -> visit28
```

```
[16..189], quantity [-16] -> Depot [0..173], quantity (-Inf..Inf) TransitSum (-
Inf..Inf)
  Breaks : lunch6-day1 after Depot, Time : start [12.2383..12.25], duration
[0.5]
           lunch6-day2 after Depot, Time : start [36.2383..36.25], duration
[0.5]
           lunch6-day3 after visit32, Time : start [60.2383..60.25], duration
[0.5]
           lunch6-day4 after visit32, Time : start [84.2383..84.25], duration
[0.5]
           lunch6-day5 after visit27, Time : start [108.238..108.25], duration
[0.5]
           morningCoffee6-day1 after Depot, Time : start [10.2344..10.25],
duration [0.25]
           morningCoffee6-day2 after Depot, Time : start [34.2344..34.25],
duration [0.25]
           morningCoffee6-day3 after visit31, Time : start [58.2344..58.25],
duration [0.25]
           morningCoffee6-day4 after visit32, Time : start [82.2344..82.25],
duration [0.25]
           morningCoffee6-day5 after visit27, Time : start [106.359..106.375],
duration [0.25]
           aftenoonCoffee6-day1 after Depot, Time : start [15.4844..15.5],
duration [0.25]
           aftenoonCoffee6-day2 after visit31, Time : start [39.4844..39.5],
duration [0.25]
           aftenoonCoffee6-day3 after visit32, Time : start [63.4844..63.5],
duration [0.25]
           aftenoonCoffee6-day4 after visit32, Time : start [87.4844..87.5],
duration [0.25]
           aftenoonCoffee6-day5 after Depot, Time : start [111.484..111.5],
duration [0.25]
           night6-day1 after Depot, Time : start [18.9844..19], duration [7]
           night6-day2 after visit31, Time : start [42.9844..43], duration [10]
           night6-day3 after visit32, Time : start [66.9844..67], duration [7]
           night6-day4 after visit32, Time : start [90.9844..91], duration [10]

vehicle7 : Unused
  Breaks : lunch7-day1 after Depot, Time : start [12.2383..12.25], duration
[0.5]
           lunch7-day2 after Depot, Time : start [36.2383..36.25], duration
[0.5]
           lunch7-day3 after Depot, Time : start [60.2383..60.25], duration
[0.5]
           lunch7-day4 after Depot, Time : start [84.2383..84.25], duration
[0.5]
           lunch7-day5 after Depot, Time : start [108.238..108.25], duration
[0.5]
           morningCoffee7-day1 after Depot, Time : start [10.2344..10.25],
duration [0.25]
           morningCoffee7-day2 after Depot, Time : start [34.2344..34.25],
duration [0.25]
           morningCoffee7-day3 after Depot, Time : start [58.2344..58.25],
duration [0.25]
           morningCoffee7-day4 after Depot, Time : start [82.2344..82.25],
duration [0.25]
           morningCoffee7-day5 after Depot, Time : start [106.234..106.25],
duration [0.25]
```

```
           aftenoonCoffee7-day1 after Depot, Time : start [15.4844..15.5],
duration [0.25]
           aftenoonCoffee7-day2 after Depot, Time : start [39.4844..39.5],
duration [0.25]
           aftenoonCoffee7-day3 after Depot, Time : start [63.4844..63.5],
duration [0.25]
           aftenoonCoffee7-day4 after Depot, Time : start [87.4844..87.5],
duration [0.25]
           aftenoonCoffee7-day5 after Depot, Time : start [111.484..111.5],
duration [0.25]
           night7-day1 after Depot, Time : start [18.9844..19], duration [7]
           night7-day2 after Depot, Time : start [42.9844..43], duration [10]
           night7-day3 after Depot, Time : start [66.9844..67], duration [7]
           night7-day4 after Depot, Time : start [90.9844..91], duration [10]

vehicle8 : Unused
  Breaks : lunch8-day1 after Depot, Time : start [12.2383..12.25], duration
[0.5]
           lunch8-day2 after Depot, Time : start [36.2383..36.25], duration
[0.5]
           lunch8-day3 after Depot, Time : start [60.2383..60.25], duration
[0.5]
           lunch8-day4 after Depot, Time : start [84.2383..84.25], duration
[0.5]
           lunch8-day5 after Depot, Time : start [108.238..108.25], duration
[0.5]
           morningCoffee8-day1 after Depot, Time : start [10.2344..10.25],
duration [0.25]
           morningCoffee8-day2 after Depot, Time : start [34.2344..34.25],
duration [0.25]
           morningCoffee8-day3 after Depot, Time : start [58.2344..58.25],
duration [0.25]
           morningCoffee8-day4 after Depot, Time : start [82.2344..82.25],
duration [0.25]
           morningCoffee8-day5 after Depot, Time : start [106.234..106.25],
duration [0.25]
           aftenoonCoffee8-day1 after Depot, Time : start [15.4844..15.5],
duration [0.25]
           aftenoonCoffee8-day2 after Depot, Time : start [39.4844..39.5],
duration [0.25]
           aftenoonCoffee8-day3 after Depot, Time : start [63.4844..63.5],
duration [0.25]
           aftenoonCoffee8-day4 after Depot, Time : start [87.4844..87.5],
duration [0.25]
           aftenoonCoffee8-day5 after Depot, Time : start [111.484..111.5],
duration [0.25]
           night8-day1 after Depot, Time : start [18.9844..19], duration [7]
           night8-day2 after Depot, Time : start [42.9844..43], duration [10]
           night8-day3 after Depot, Time : start [66.9844..67], duration [7]
           night8-day4 after Depot, Time : start [90.9844..91], duration [10]

vehicle9 : Unused
  Breaks : lunch9-day1 after Depot, Time : start [12.2383..12.25], duration
[0.5]
           lunch9-day2 after Depot, Time : start [36.2383..36.25], duration
[0.5]
           lunch9-day3 after Depot, Time : start [60.2383..60.25], duration
[0.5]
```

```
           lunch9-day4 after Depot, Time : start [84.2383..84.25], duration
[0.5]
           lunch9-day5 after Depot, Time : start [108.238..108.25], duration
[0.5]
           morningCoffee9-day1 after Depot, Time : start [10.2344..10.25],
duration [0.25]
           morningCoffee9-day2 after Depot, Time : start [34.2344..34.25],
duration [0.25]
           morningCoffee9-day3 after Depot, Time : start [58.2344..58.25],
duration [0.25]
           morningCoffee9-day4 after Depot, Time : start [82.2344..82.25],
duration [0.25]
           morningCoffee9-day5 after Depot, Time : start [106.234..106.25],
duration [0.25]
           aftenoonCoffee9-day1 after Depot, Time : start [15.4844..15.5],
duration [0.25]
           aftenoonCoffee9-day2 after Depot, Time : start [39.4844..39.5],
duration [0.25]
           aftenoonCoffee9-day3 after Depot, Time : start [63.4844..63.5],
duration [0.25]
           aftenoonCoffee9-day4 after Depot, Time : start [87.4844..87.5],
duration [0.25]
           aftenoonCoffee9-day5 after Depot, Time : start [111.484..111.5],
duration [0.25]
           night9-day1 after Depot, Time : start [18.9844..19], duration [7]
           night9-day2 after Depot, Time : start [42.9844..43], duration [10]
           night9-day3 after Depot, Time : start [66.9844..67], duration [7]
           night9-day4 after Depot, Time : start [90.9844..91], duration [10]

vehicle10 : Unused
  Breaks : lunch10-day1 after Depot, Time : start [12.2383..12.25], duration
[0.5]
           lunch10-day2 after Depot, Time : start [36.2383..36.25], duration
[0.5]
           lunch10-day3 after Depot, Time : start [60.2383..60.25], duration
[0.5]
           lunch10-day4 after Depot, Time : start [84.2383..84.25], duration
[0.5]
           lunch10-day5 after Depot, Time : start [108.238..108.25], duration
[0.5]
           morningCoffee10-day1 after Depot, Time : start [10.2344..10.25],
duration [0.25]
           morningCoffee10-day2 after Depot, Time : start [34.2344..34.25],
duration [0.25]
           morningCoffee10-day3 after Depot, Time : start [58.2344..58.25],
duration [0.25]
           morningCoffee10-day4 after Depot, Time : start [82.2344..82.25],
duration [0.25]
           morningCoffee10-day5 after Depot, Time : start [106.234..106.25],
duration [0.25]
           aftenoonCoffee10-day1 after Depot, Time : start [15.4844..15.5],
duration [0.25]
           aftenoonCoffee10-day2 after Depot, Time : start [39.4844..39.5],
duration [0.25]
           aftenoonCoffee10-day3 after Depot, Time : start [63.4844..63.5],
duration [0.25]
           aftenoonCoffee10-day4 after Depot, Time : start [87.4844..87.5],
duration [0.25]
```

```
           aftenoonCoffee10-day5 after Depot, Time : start [111.484..111.5],
duration [0.25]
           night10-day1 after Depot, Time : start [18.9844..19], duration [7]
           night10-day2 after Depot, Time : start [42.9844..43], duration [10]
           night10-day3 after Depot, Time : start [66.9844..67], duration [7]
           night10-day4 after Depot, Time : start [90.9844..91], duration [10]

vehicle11 : Unused
  Breaks : lunch11-day1 after Depot, Time : start [12.2383..12.25], duration
[0.5]
           lunch11-day2 after Depot, Time : start [36.2383..36.25], duration
[0.5]
           lunch11-day3 after Depot, Time : start [60.2383..60.25], duration
[0.5]
           lunch11-day4 after Depot, Time : start [84.2383..84.25], duration
[0.5]
           lunch11-day5 after Depot, Time : start [108.238..108.25], duration
[0.5]
           morningCoffee11-day1 after Depot, Time : start [10.2344..10.25],
duration [0.25]
           morningCoffee11-day2 after Depot, Time : start [34.2344..34.25],
duration [0.25]
           morningCoffee11-day3 after Depot, Time : start [58.2344..58.25],
duration [0.25]
           morningCoffee11-day4 after Depot, Time : start [82.2344..82.25],
duration [0.25]
           morningCoffee11-day5 after Depot, Time : start [106.234..106.25],
duration [0.25]
           aftenoonCoffee11-day1 after Depot, Time : start [15.4844..15.5],
duration [0.25]
           aftenoonCoffee11-day2 after Depot, Time : start [39.4844..39.5],
duration [0.25]
           aftenoonCoffee11-day3 after Depot, Time : start [63.4844..63.5],
duration [0.25]
           aftenoonCoffee11-day4 after Depot, Time : start [87.4844..87.5],
duration [0.25]
           aftenoonCoffee11-day5 after Depot, Time : start [111.484..111.5],
duration [0.25]
           night11-day1 after Depot, Time : start [18.9844..19], duration [7]
           night11-day2 after Depot, Time : start [42.9844..43], duration [10]
           night11-day3 after Depot, Time : start [66.9844..67], duration [7]
           night11-day4 after Depot, Time : start [90.9844..91], duration [10]

vehicle12 : Unused
  Breaks : lunch12-day1 after Depot, Time : start [12.2383..12.25], duration
[0.5]
           lunch12-day2 after Depot, Time : start [36.2383..36.25], duration
[0.5]
           lunch12-day3 after Depot, Time : start [60.2383..60.25], duration
[0.5]
           lunch12-day4 after Depot, Time : start [84.2383..84.25], duration
[0.5]
           lunch12-day5 after Depot, Time : start [108.238..108.25], duration
[0.5]
           morningCoffee12-day1 after Depot, Time : start [10.2344..10.25],
duration [0.25]
           morningCoffee12-day2 after Depot, Time : start [34.2344..34.25],
duration [0.25]
```

```
           morningCoffee12-day3 after Depot, Time : start [58.2344..58.25],
duration [0.25]
           morningCoffee12-day4 after Depot, Time : start [82.2344..82.25],
duration [0.25]
           morningCoffee12-day5 after Depot, Time : start [106.234..106.25],
duration [0.25]
           aftenoonCoffee12-day1 after Depot, Time : start [15.4844..15.5],
duration [0.25]
           aftenoonCoffee12-day2 after Depot, Time : start [39.4844..39.5],
duration [0.25]
           aftenoonCoffee12-day3 after Depot, Time : start [63.4844..63.5],
duration [0.25]
           aftenoonCoffee12-day4 after Depot, Time : start [87.4844..87.5],
duration [0.25]
           aftenoonCoffee12-day5 after Depot, Time : start [111.484..111.5],
duration [0.25]
           night12-day1 after Depot, Time : start [18.9844..19], duration [7]
           night12-day2 after Depot, Time : start [42.9844..43], duration [10]
           night12-day3 after Depot, Time : start [66.9844..67], duration [7]
           night12-day4 after Depot, Time : start [90.9844..91], duration [10]

vehicle13 : Unused
  Breaks : lunch13-day1 after Depot, Time : start [12.2383..12.25], duration
[0.5]
           lunch13-day2 after Depot, Time : start [36.2383..36.25], duration
[0.5]
           lunch13-day3 after Depot, Time : start [60.2383..60.25], duration
[0.5]
           lunch13-day4 after Depot, Time : start [84.2383..84.25], duration
[0.5]
           lunch13-day5 after Depot, Time : start [108.238..108.25], duration
[0.5]
           morningCoffee13-day1 after Depot, Time : start [10.2344..10.25],
duration [0.25]
           morningCoffee13-day2 after Depot, Time : start [34.2344..34.25],
duration [0.25]
           morningCoffee13-day3 after Depot, Time : start [58.2344..58.25],
duration [0.25]
           morningCoffee13-day4 after Depot, Time : start [82.2344..82.25],
duration [0.25]
           morningCoffee13-day5 after Depot, Time : start [106.234..106.25],
duration [0.25]
           aftenoonCoffee13-day1 after Depot, Time : start [15.4844..15.5],
duration [0.25]
           aftenoonCoffee13-day2 after Depot, Time : start [39.4844..39.5],
duration [0.25]
           aftenoonCoffee13-day3 after Depot, Time : start [63.4844..63.5],
duration [0.25]
           aftenoonCoffee13-day4 after Depot, Time : start [87.4844..87.5],
duration [0.25]
           aftenoonCoffee13-day5 after Depot, Time : start [111.484..111.5],
duration [0.25]
           night13-day1 after Depot, Time : start [18.9844..19], duration [7]
           night13-day2 after Depot, Time : start [42.9844..43], duration [10]
           night13-day3 after Depot, Time : start [66.9844..67], duration [7]
           night13-day4 after Depot, Time : start [90.9844..91], duration [10]

vehicle14 : Unused
```

```
  Breaks : lunch14-day1 after Depot, Time : start [12.2383..12.25], duration 
[0.5]
           lunch14-day2 after Depot, Time : start [36.2383..36.25], duration 
[0.5]
           lunch14-day3 after Depot, Time : start [60.2383..60.25], duration 
[0.5]
           lunch14-day4 after Depot, Time : start [84.2383..84.25], duration 
[0.5]
           lunch14-day5 after Depot, Time : start [108.238..108.25], duration 
[0.5]
           morningCoffee14-day1 after Depot, Time : start [10.2344..10.25], 
duration [0.25]
           morningCoffee14-day2 after Depot, Time : start [34.2344..34.25], 
duration [0.25]
           morningCoffee14-day3 after Depot, Time : start [58.2344..58.25], 
duration [0.25]
           morningCoffee14-day4 after Depot, Time : start [82.2344..82.25], 
duration [0.25]
           morningCoffee14-day5 after Depot, Time : start [106.234..106.25], 
duration [0.25]
           aftenoonCoffee14-day1 after Depot, Time : start [15.4844..15.5], 
duration [0.25]
           aftenoonCoffee14-day2 after Depot, Time : start [39.4844..39.5], 
duration [0.25]
           aftenoonCoffee14-day3 after Depot, Time : start [63.4844..63.5], 
duration [0.25]
           aftenoonCoffee14-day4 after Depot, Time : start [87.4844..87.5], 
duration [0.25]
           aftenoonCoffee14-day5 after Depot, Time : start [111.484..111.5], 
duration [0.25]
           night14-day1 after Depot, Time : start [18.9844..19], duration [7]
           night14-day2 after Depot, Time : start [42.9844..43], duration [10]
           night14-day3 after Depot, Time : start [66.9844..67], duration [7]
           night14-day4 after Depot, Time : start [90.9844..91], duration [10]

vehicle15 : Unused
  Breaks : lunch15-day1 after Depot, Time : start [12.2383..12.25], duration 
[0.5]
           lunch15-day2 after Depot, Time : start [36.2383..36.25], duration 
[0.5]
           lunch15-day3 after Depot, Time : start [60.2383..60.25], duration 
[0.5]
           lunch15-day4 after Depot, Time : start [84.2383..84.25], duration 
[0.5]
           lunch15-day5 after Depot, Time : start [108.238..108.25], duration 
[0.5]
           morningCoffee15-day1 after Depot, Time : start [10.2344..10.25], 
duration [0.25]
           morningCoffee15-day2 after Depot, Time : start [34.2344..34.25], 
duration [0.25]
           morningCoffee15-day3 after Depot, Time : start [58.2344..58.25], 
duration [0.25]
           morningCoffee15-day4 after Depot, Time : start [82.2344..82.25], 
duration [0.25]
           morningCoffee15-day5 after Depot, Time : start [106.234..106.25], 
duration [0.25]
           aftenoonCoffee15-day1 after Depot, Time : start [15.4844..15.5], 
duration [0.25]
```

```
          aftenoonCoffee15-day2 after Depot, Time : start [39.4844..39.5], 
duration [0.25]
          aftenoonCoffee15-day3 after Depot, Time : start [63.4844..63.5], 
duration [0.25]
          aftenoonCoffee15-day4 after Depot, Time : start [87.4844..87.5], 
duration [0.25]
          aftenoonCoffee15-day5 after Depot, Time : start [111.484..111.5], 
duration [0.25]
          night15-day1 after Depot, Time : start [18.9844..19], duration [7]
          night15-day2 after Depot, Time : start [42.9844..43], duration [10]
          night15-day3 after Depot, Time : start [66.9844..67], duration [7]
          night15-day4 after Depot, Time : start [90.9844..91], duration [10]

*/
```

CHAPTER

9

Adding Early and Late Costs

In this lesson, you will learn how to:

- create a *soft deadline*
- model costs for both early and late deliveries
- use the member function `setStartCost`, and the functions `IloEarlinessFunction` and `IloTardinessFunction`.

Describe

In standard vehicle routing problems, an optimal solution is found by building a set of routes to perform each visit and minimizing the cost of the routes. The cost of any vehicle is usually a linear combination of the length and duration of its route.

However, in the real world, the number of available vehicles may be limited or the distances so great that it may not be possible to perform all visits within the specified time windows. This may mean additional costs to the customer as a lack of stock leads to lost sales or a slowdown in a manufacturing line. This introduces a new type of cost into the solution, a cost based on the lateness of deliveries.

In Dispatcher, this type of problem can be modeled by “softening” the delivery deadlines by introducing an associated cost for “tardy” deliveries.

In soft deadline problems, the best delivery time is usually just the earliest possible delivery time. However, in some cases, costs can arise for *early* deliveries as well. For example, a customer may incur extra storage and handling expenses if a delivery is too early to be used immediately, or if it exceeds the customer's normal storage capacity.

The example in this chapter considers both late and early delivery costs. This substantially increases the difficulty, since making an initial delivery on time may cause subsequent deliveries to be late and costly; making the subsequent deliveries on time may cause the initial delivery to be costly as it is too early. Each visit in a route has more interdependencies; in some cases, it may be beneficial to have a vehicle wait to make a delivery.

Early and Late Cost Interdependence

Figure 9.1 illustrates a cost-time relationship for a single delivery that has both early and late costs. Cost for an early delivery declines as it approaches the earliest desired arrival time (`tmin`), at which point cost is zero. After the latest desired arrival time (`tmax`), cost increases as time increases.

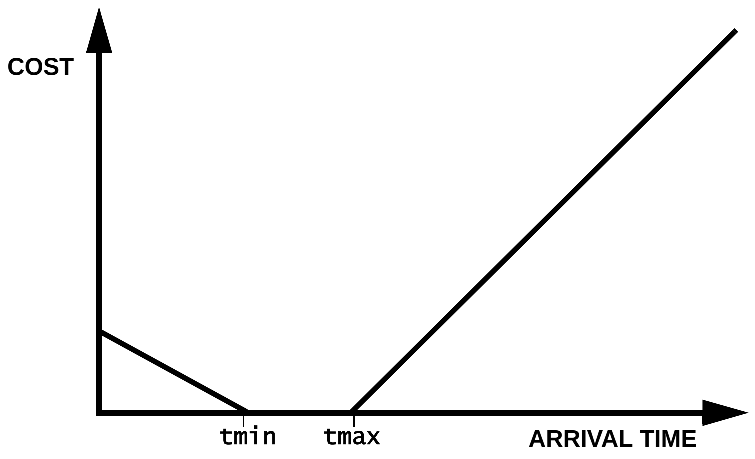

Figure 9.1 *Cost-time for one delivery with earliness and lateness costs*

The relationship becomes more complex when you consider two visits, V_1 and V_2, such that V_2 is directly after V_1 on the route. Figure 9.2 depicts the cost function for these visits based upon arrival time at V_1, where the time window for V_1 is [2, 5]; the time window for V_2 is [4, 8]; and the transit time from V_1 to V_2 is 1.

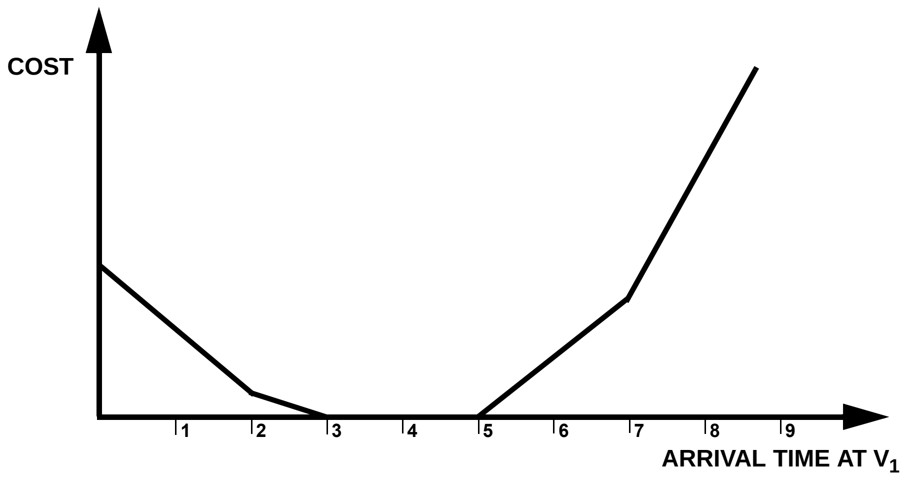

Figure 9.2 *Cost-time for two deliveries with earliness and lateness costs*

Under certain circumstances, a vehicle might need to wait between V_1 and V_2, causing an effective increase in the transit time. Figure 9.3 depicts the same two visits, V_1 and V_2, but with a transit time of 3.

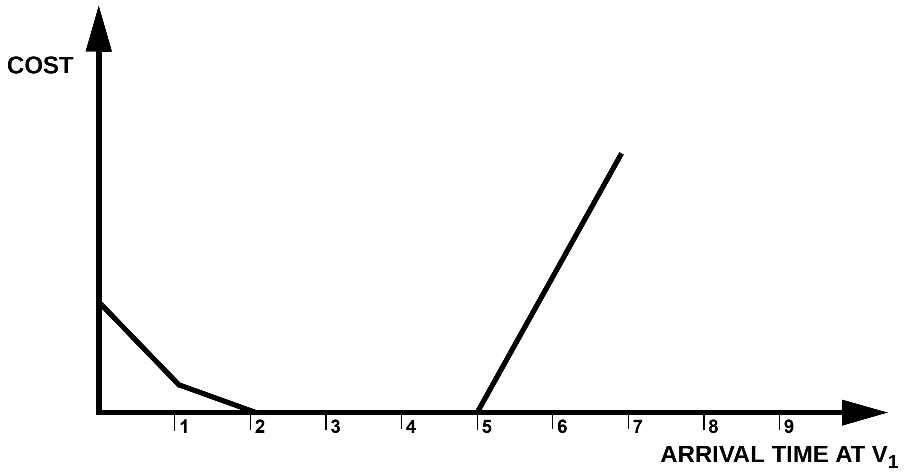

Figure 9.3 *Revised cost-time for two deliveries with earliness and lateness costs*

This illustrates the importance of determining waiting times in order to identify optimal arrival times.

Step 1 Describe the problem

The first step is to write a natural language description of the problem. A good way to start this process is to analyze the variables and objectives.

What are the constraints in this problem?

- Just as for a standard PDP, delivery trucks will pick up parcels en route and deliver them on the same route. As a consequence, capacity constraints on the vehicles may reach a limit along the route, rather than when departing the depot.
- Pickups as well as deliveries have constraints on the time windows when the parcel can be picked up or delivered. Pickup or delivery activities outside these time windows incur additional costs for the delivery.

The objective is to minimize the cost of the delivery of all the parcels.

Model

Once you have written a description of your problem, you can use Dispatcher classes to model it.

Step 2 Open the example file

Open the example file `YourDispatcherHome/examples/src/tutorial/earlytardy_partial.cpp` in your development environment.

The problem in this lesson is the same as the PDP problem in Chapter 8 that you have already worked with, except that costs are added for early and late deliveries.

The early and tardy costs are added in the function `createVisits`.

Step 3 Add the early and tardy costs

Add the following code after the comment `//Add the early and tardy costs.`

```
IloInt earlyCost = 5;
IloInt tardyCost = 10;
```

The integer variables `earlyCost` and `tardyCost` are used later by `IloEarlinessFunction` and `IloTardinessFunction` to compute the cost of the delivery schedules.

Step 4 Add the earliness and tardiness functions

Add the following code after the comment `//Add the earliness and tardiness functions`.

```
pickup.setStartCost(time,
                    IloEarlinessFunction(_env,
                                         pickupMinTime, earlyCost)
                    + IloTardinessFunction(_env,
                                           pickupMaxTime, tardyCost));
```

`IloEarlinessFunction` is used to compute the total cost of an early pickup time from the `earlyCost` variable and the amount of `time` that the delivery is early. `IloTardinessFunction` similarily computes the tardy cost. The output of these two functions is added together to compute the cost of the pickup visit. The total cost is then assigned to the pickup visit with the member function `setStartCost`. `setStartCost` is used (as opposed to `setEndCost`) to express the fact that the cost will be a function of the `cumul` variable (as opposed to the `endCumul` variable).

This process is also performed for every delivery visit.

```
delivery.setStartCost(time,
                      IloEarlinessFunction(_env,
                                           deliveryMinTime, earlyCost)
                      + IloTardinessFunction(_env,
                                             deliveryMaxTime, tardyCost));
```

The remainder of the modeling code is the same or similar to `pdp.cpp`, presented in Chapter 7, *Pickup and Delivery Problems*.

Solve

The solution is computed, improved, and displayed by methods previously presented for the VRP problem, with one addition.

The solve portion of the problem needs to account for the fact that time needs to be instantiated. In our previous examples, the earliest possible delivery time was always the optimal delivery time. In this example, that is not necessarily true due to the possibility of early delivery charges; shifting the schedule to the earliest possible time may actually increase the cost of the schedule.

Step 5 Instantiate time

Add the following code after the comment `//Instantiate time`.

```
  IloGoal subGoal = IloSetVisitCumuls(env,
                                    IloDimension2::Find(env, "Time"),
                                    1e-6)
               && instantiateCost;
```

Step 6 Compile and run the program

The solution improvement phase finds a solution using 9 vehicles with a cost of 1128.95 units:

```
3706.08
Improving solution
Number of fails               : 0
Number of choice points       : 1049
Number of variables           : 2329
Number of constraints         : 410
Reversible stack (bytes)      : 213084
Solver heap (bytes)           : 1049500
Solver global heap (bytes)    : 1363004
And stack (bytes)             : 20124
Or stack (bytes)              : 44244
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 11160
Total memory used (bytes)     : 2705160
Elapsed time since creation   : 0.016
Number of nodes               : 51
Number of visits              : 80
Number of vehicles            : 15
Number of dimensions          : 3
Number of accepted moves      : 44
===============
Cost         : 1128.95
Number of vehicles used : 9
```

The complete program and output are listed in “Complete Program” on page 230. You can also view it online in the `YourDispatcherHome/examples/src/earlytardy.cpp` file.

Review Exercises

For answers, see “Suggested Answers” on page 229.

1. What is a *soft deadline*?

2. What function is used in this example to model a soft deadline?

3. Are additional functions used to model early costs?
4. The solving and goals portion of the code is shown below. How would this section of code change if there were no early costs?

```
void RoutingSolver::solve() {
  IloDispatcher dispatcher(_solver);
  IloEnv env = _solver.getEnv();
  // Subgoals
  IloGoal instantiateCost = IloDichotomize(env,
                                           dispatcher.getCostVar(),
                                           IloFalse);

  IloGoal subGoal = IloSetVisitCumuls(env,
                                      IloDimension2::Find(env, "Time"),
                                      1e-6)
                 && instantiateCost;

  IloGoal restoreSolution = IloRestoreSolution(env, _solution) && subGoal;
  IloGoal goal = IloSavingsGenerate(env) && subGoal;

  // Solving
  if (findFirstSolution(goal)) {
    improve(subGoal);
    _solver.solve(restoreSolution);
  }
}
```

Suggested Answers

Exercise 1

What is a *soft deadline*?

Suggested Answer

A soft deadline is a deadline that can be violated, but at a cost.

Exercise 2

What function is used in this example to model a soft deadline?

Suggested Answer

`IloTardinessFunction` is used to model a soft deadline.

Exercise 3

Are additional functions used to model early costs?

Suggested Answer

Yes. `IloEarlinessFunction` is used to compute the cost of an early delivery.

Exercise 4

How would the solving and goals code change if there were no early costs?

Suggested Answer

If there were no early costs, there would be no need to instantiate time, as the earliest possible delivery time would be the best delivery time. The modified code follows.

```
void RoutingSolver::solve() {
  IloDispatcher dispatcher(_solver);
  IloEnv env = _solver.getEnv();
  // Subgoals
  IloGoal subGoal = IloDichotomize(env,
                                   dispatcher.getCostVar(),
                                   IloFalse);
  IloGoal restoreSolution = IloRestoreSolution(env, _solution) && subGoal;
  IloGoal goal = IloSavingsGenerate(env) && subGoal;

  // Solving
  if (findFirstSolution(goal)) {
    improve(subGoal);
    _solver.solve(restoreSolution);
  }
}
```

Complete Program

The complete program follows. You can also view it online in the file `YourDispatcherHome/examples/src/earlytardy.cpp`.

```
// -------------------------------------------------------------- -*- C++ -*-
// File: examples/src/earlytardy.cpp
// --------------------------------------------------------------------------


#include <ildispat/ilodispatcher.h>

ILOSTLBEGIN
///////////////////////////////////////////////////////////////////////////////
// Modeling
class RoutingModel {
  IloEnv            _env;
  IloModel          _mdl;

  void createDimensions();
  void createNodes(char* nodeFileName);
  void createVehicles(char* vehicleFileName);
  void createVisits(char* visitsFileName);
public:
  RoutingModel(IloEnv env, int argc, char* argv[]);
  ~RoutingModel() {}
  IloEnv getEnv() const { return _env; }
  IloModel getModel() const { return _mdl; }
};

RoutingModel::RoutingModel(IloEnv env, int argc, char * argv[])
  : _env(env), _mdl(env) {

  createDimensions();

  char* nodeFileName;
  if(argc < 4)
    nodeFileName = (char *) "../../../examples/data/pdp/nodes.csv";
  else
    nodeFileName = argv[3];
  createNodes(nodeFileName);

  char* vehiclesFileName;
  if (argc < 2)
    vehiclesFileName = (char *) "../../../examples/data/pdp/vehicles.csv";
  else
    vehiclesFileName = argv[1];
  createVehicles(vehiclesFileName);

  char* visitsFileName;
  if (argc < 3)
    visitsFileName = (char*) "../../../examples/data/pdp/visits.csv";
  else
    visitsFileName = argv[2];
  createVisits(visitsFileName);
}

// Create dimensions
void RoutingModel::createDimensions() {
  IloDimension2 time(_env, IloEuclidean, "Time");
  time.setKey("Time");
  _mdl.add(time);
  IloDimension2 length(_env, IloEuclidean, IloFalse, "Length");
  length.setKey("Length");
  _mdl.add(length);
  IloDimension1 weight(_env, "Weight");
  weight.setKey("Weight");
  _mdl.add(weight);
}

// Create nodes
void RoutingModel::createNodes(char* nodeFileName) {
  IloCsvReader csvNodeReader(_env, nodeFileName);
  IloCsvReader::LineIterator it(csvNodeReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char* name = line.getStringByHeader("name");
    IloNode node(_env,
                 line.getFloatByHeader("x"),
                 line.getFloatByHeader("y"),
                 0,
                 name);
    node.setKey(name);
    ++it;
  }
  csvNodeReader.end();
}

// Create vehicles
void RoutingModel::createVehicles(char* vehicleFileName) {
  IloDimension1 weight = IloDimension1::Find(_env, “Weight”);
  IloDimension2 time = IloDimension2::Find(_env, “Time”);
  IloDimension2 length = IloDimension2::Find(_env, “Length”);

  IloCsvReader csvVehicleReader(_env, vehicleFileName);
  IloCsvReader::LineIterator  it(csvVehicleReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char * namefirst = line.getStringByHeader(“first”);
    char * namelast = line.getStringByHeader(“last”);
    char * name = line.getStringByHeader(“name”);
    IloNum capacity = line.getFloatByHeader(“capacity”);
    IloNode node1 = IloNode::Find(_env, namefirst);
    IloNode node2 = IloNode::Find(_env,namelast);

    IloVisit first(node1, “Depot”);
    _mdl.add(first.getCumulVar(weight) == 0);
    _mdl.add(first.getCumulVar(time) >= line.getFloatByHeader(“open”));

    IloVisit last(node2, “Depot”);
    _mdl.add(last.getCumulVar(time) <= line.getFloatByHeader(“close”));
    IloVehicle vehicle(first, last, name);
    vehicle.setCost(length, 1.0);
    vehicle.setCapacity(weight, capacity);
    _mdl.add(vehicle);
    ++it;
  }
  csvVehicleReader.end();
}

// Create visits
void RoutingModel::createVisits(char* visitsFileName) {

  IloInt earlyCost = 5;
  IloInt tardyCost = 10;

  IloDimension1 weight = IloDimension1::Find(_env, “Weight”);
  IloDimension2 time = IloDimension2::Find(_env, “Time”);

  IloCsvReader csvVisitReader(_env, visitsFileName);
  IloCsvReader::LineIterator  it(csvVisitReader);
  while(it.ok()){
    IloCsvLine line = *it;
    //read visit data from files
    char * pickupVisitName = line.getStringByHeader(“pickup”);
    char * pickupNodeName = line.getStringByHeader(“pickupNode”);
    char * deliveryVisitName = line.getStringByHeader(“delivery”);
    char * deliveryNodeName = line.getStringByHeader(“deliveryNode”);
    IloNum quantity = line.getFloatByHeader(“quantity”);
    IloNum pickupMinTime  = line.getFloatByHeader(“pickupMinTime”);
    IloNum pickupMaxTime  = line.getFloatByHeader(“pickupMaxTime”);
    IloNum deliveryMinTime  = line.getFloatByHeader(“deliveryMinTime”);
    IloNum deliveryMaxTime  = line.getFloatByHeader(“deliveryMaxTime”);
    IloNum dropTime = line.getFloatByHeader(“dropTime”);
    IloNum pickupTime = line.getFloatByHeader(“pickupTime”);

    // read data nodes from the file nodes.csv
    // and creat pickup and delivery nodes

    IloNode pickupNode = IloNode::Find(_env, pickupNodeName);
    IloNode deliveryNode = IloNode::Find(_env, deliveryNodeName);
    //create and add pickup and delivery visits
    IloVisit pickup(pickupNode, pickupVisitName);
    _mdl.add(pickup.getDelayVar(time) == pickupTime);
    _mdl.add(pickup.getTransitVar(weight) == quantity);

    pickup.setStartCost(time,
                        IloEarlinessFunction(_env,
                                             pickupMinTime, earlyCost)
                        + IloTardinessFunction(_env,
                                               pickupMaxTime, tardyCost));

    _mdl.add(pickup);
    IloVisit delivery(deliveryNode, deliveryVisitName);
    _mdl.add(delivery.getDelayVar(time) == dropTime);
    _mdl.add(delivery.getTransitVar(weight) == -quantity);

    delivery.setStartCost(time,
                          IloEarlinessFunction(_env,
                                               deliveryMinTime, earlyCost)
                          + IloTardinessFunction(_env,
                                                 deliveryMaxTime, tardyCost));

    _mdl.add(delivery);
    //add pickup and delivery order constraint
    _mdl.add(IloOrderedVisitPair(_env, pickup, delivery));
    ++it;
  }
  csvVisitReader.end();
}

///////////////////////////////////////////////////////////////////////////////
// Solving
class RoutingSolver {
  IloModel            _mdl;
  IloSolver           _solver;
  IloRoutingSolution  _solution;

  IloBool findFirstSolution(IloGoal goal);
  void greedyImprove(IloNHood nhood, IloGoal subGoal);
  void improve(IloGoal subgoal);
public:
  RoutingSolver(RoutingModel mdl);
  ~RoutingSolver() {}
  IloRoutingSolution getSolution() const { return _solution; }
  void printInformation() const;
  void solve();
};

RoutingSolver::RoutingSolver(RoutingModel mdl)
  : _mdl(mdl.getModel()), _solver(mdl.getModel()), _solution(mdl.getModel()) {}

// Solving : find first solution
IloBool RoutingSolver::findFirstSolution(IloGoal goal) {
  if (!_solver.solve(goal)) {
    _solver.error() << “Infeasible Routing Plan” << endl;
    return IloFalse;
  }
  IloDispatcher dispatcher(_solver);
  _solver.out() << dispatcher.getTotalCost() << endl;
  _solution.store(_solver);
  return IloTrue;
}

// Improve solution using nhood
void RoutingSolver::greedyImprove(IloNHood nhood, IloGoal subGoal) {
  _solver.out() << “Improving solution” << endl;
  IloEnv env = _solver.getEnv();
  nhood.reset();
  IloGoal improve = IloSingleMove(env,
                                  _solution,
                                  nhood,
                                  IloImprove(env),
                                  subGoal);
  while (_solver.solve(improve)) {}
}

// Improve solution
void RoutingSolver::improve(IloGoal subGoal) {
  IloEnv env = _solver.getEnv();
  IloNHood nhood = IloTwoOpt(env)
                 + IloOrOpt(env)
                 + IloRelocate(env)
                 + IloCross(env)
                 + IloExchange(env);
  greedyImprove(nhood, subGoal);
}

// Display Dispatcher information
void RoutingSolver::printInformation() const {
  IloDispatcher dispatcher(_solver);
  _solver.printInformation();
  dispatcher.printInformation();
  _solver.out() << “===============” << endl
                << “Cost         : “ << dispatcher.getTotalCost() << endl
                << “Number of vehicles used : “
                << dispatcher.getNumberOfVehiclesUsed() << endl
                << “Solution     : “ << endl
                << dispatcher << endl;
}

// Solving
void RoutingSolver::solve() {
  IloDispatcher dispatcher(_solver);
  IloEnv env = _solver.getEnv();
  // Subgoals
  IloGoal instantiateCost = IloDichotomize(env,
                                          dispatcher.getCostVar(),
                                          IloFalse);

  IloGoal subGoal = IloSetVisitCumuls(env,
                                     IloDimension2::Find(env, “Time”),
                                     1e-6)
                 && instantiateCost;

  IloGoal restoreSolution = IloRestoreSolution(env, _solution) && subGoal;
  IloGoal goal = IloSavingsGenerate(env) && subGoal;

  // Solving
  if (findFirstSolution(goal)) {
    improve(subGoal);
    _solver.solve(restoreSolution);
  }
}

///////////////////////////////////////////////////////////////////////////////
int main(int argc, char * argv[]) {
  IloEnv env;
  try {
    RoutingModel mdl(env, argc, argv);
    RoutingSolver solver(mdl);
    solver.solve();
    solver.printInformation();
  } catch(IloException& ex) {
    cerr << “Error: “ << ex << endl;
  }
  env.end();
  return 0;
}
```

Complete Output

```
/**
3706.08
Improving solution
Number of fails              : 0
Number of choice points      : 1049
Number of variables          : 2329
Number of constraints        : 410
Reversible stack (bytes)     : 213084
Solver heap (bytes)          : 1049500
Solver global heap (bytes)   : 1363004
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11160
Total memory used (bytes)    : 2705160
Elapsed time since creation  : 0.016
Number of nodes              : 51
Number of visits             : 80
Number of vehicles           : 15
Number of dimensions         : 3
Number of accepted moves     : 44
===============
Cost         : 1128.95
Number of vehicles used : 9
Solution     :
Unperformed visits : None
vehicle1 :
 -> Depot Time[0..1e-006] Length[0..Inf) Weight[0] -> visit35
Time[41.0366..41.0366] Length[0..Inf) Weight[0..162] -> visit49
Time[108.115..108.115] Length[0..Inf) Weight[8..170] -> visit36
Time[127.06..127.06] Length[0..Inf) Weight[38..200] -> visit50
Time[183.9..183.9] Length[0..Inf) Weight[30..192] -> Depot Time[210.87..210.87]
Length[0..Inf) Weight[0..162]
vehicle2 : Unused
vehicle3 :
 -> Depot Time[0..1e-006] Length[0..Inf) Weight[0] -> visit21
Time[18.0278..18.0278] Length[0..Inf) Weight[0..129] -> visit39
Time[45.0278..45.0278] Length[0..Inf) Weight[11..140] -> visit23
Time[66.7218..66.7218] Length[0..Inf) Weight[42..171] -> visit22
Time[87.9021..87.9021] Length[0..Inf) Weight[71..200] -> visit40
Time[113.713..113.713] Length[0..Inf) Weight[60..189] -> visit24
Time[150.639..150.639] Length[0..Inf) Weight[29..158] -> visit25
Time[175.639..175.639] Length[0..Inf) Weight[0..129] -> visit26 Time[208..208]
Length[0..Inf) Weight[6..135] -> Depot Time[229.18..229.18] Length[0..Inf)
Weight[0..129]
vehicle4 :
 -> Depot Time[0..1e-006] Length[0..Inf) Weight[0] -> visit45
Time[29.1548..29.1548] Length[0..Inf) Weight[0..156] -> visit46
Time[49.9251..49.9251] Length[0..Inf) Weight[16..172] -> visit47
Time[69.9251..69.9251] Length[0..Inf) Weight[0..156] -> visit48
Time[86.3282..86.3282] Length[0..Inf) Weight[27..183] -> visit19
Time[104.574..104.574] Length[0..Inf) Weight[0..156] -> visit31
Time[132.463..132.463] Length[0..Inf) Weight[17..173] -> visit32
Time[159.927..159.927] Length[0..Inf) Weight[44..200] -> visit20
Time[180.698..180.698] Length[0..Inf) Weight[17..173] -> Depot
Time[222.32..222.32] Length[0..Inf) Weight[0..156]
vehicle5 :
 -> Depot Time[0..1e-006] Length[0..Inf) Weight[0] -> visit33
Time[24.7588..24.7588] Length[0..Inf) Weight[0..175] -> visit29
Time[49.6249..49.6249] Length[0..Inf) Weight[11..186] -> visit34
Time[72.6633..72.6633] Length[0..Inf) Weight[20..195] -> visit9 Time[97..97]
Length[0..Inf) Weight[9..184] -> visit30 Time[122..122] Length[0..Inf)
Weight[25..200] -> visit10 Time[142..142] Length[0..Inf) Weight[16..191] ->
Depot Time[177.495..177.495] Length[0..Inf) Weight[0..175]
vehicle6 :
 -> Depot Time[0..1e-006] Length[0..Inf) Weight[0] -> visit1
Time[15.2315..15.2315] Length[0..Inf) Weight[0..167] -> visit7
Time[46.2553..46.2553] Length[0..Inf) Weight[10..177] -> visit8 Time[95..95]
Length[0..Inf) Weight[15..182] -> visit13 Time[131.907..131.907] Length[0..Inf)
Weight[10..177] -> visit14 Time[163.12..163.12] Length[0..Inf) Weight[33..200]
-> visit2 Time[194.31..194.31] Length[0..Inf) Weight[10..177] -> Depot
Time[222.31..222.31] Length[0..Inf) Weight[0..167]
vehicle7 :
 -> Depot Time[0..1e-006] Length[0..Inf) Weight[0] -> visit41
Time[28.8617..28.8617] Length[0..Inf) Weight[0..174] -> visit15 Time[61..61]
Length[0..Inf) Weight[5..179] -> visit42 Time[80.2195..80.2195] Length[0..Inf)
```

```
Weight[13..187] -> visit16 Time[106.344..106.344] Length[0..Inf) Weight[8..182]
-> visit5 Time[127.524..127.524] Length[0..Inf) Weight[0..174] -> visit6
Time[147.524..147.524] Length[0..Inf) Weight[26..200] -> Depot
Time[168.705..168.705] Length[0..Inf) Weight[0..174]
vehicle8 :
 -> Depot Time[0..1e-006] Length[0..Inf) Weight[0] -> visit37
Time[21.2132..21.2132] Length[0..Inf) Weight[0..192] -> visit38 Time[83..83]
Length[0..Inf) Weight[8..200] -> visit17 Time[118..118] Length[0..Inf)
Weight[0..192] -> visit18 Time[146.028..146.028] Length[0..Inf) Weight[2..194]
-> Depot Time[171.839..171.839] Length[0..Inf) Weight[0..192]
vehicle9 :
 -> Depot Time[0..1e-006] Length[0..Inf) Weight[0] -> visit43
Time[34.176..34.176] Length[0..Inf) Weight[0..193] -> visit44 Time[69..69]
Length[0..Inf) Weight[7..200] -> Depot Time[110.89..110.89] Length[0..Inf)
Weight[0..193]
vehicle10 :
 -> Depot Time[0..1e-006] Length[0..Inf) Weight[0] -> visit27 Time[5..5]
Length[0..Inf) Weight[0..159] -> visit11 Time[67..67] Length[0..Inf)
Weight[16..175] -> visit3 Time[117.311..117.311] Length[0..Inf) Weight[28..187]
-> visit4 Time[152.311..152.311] Length[0..Inf) Weight[41..200] -> visit12
Time[178.123..178.123] Length[0..Inf) Weight[28..187] -> visit28
Time[197.342..197.342] Length[0..Inf) Weight[16..175] -> Depot
Time[213.667..213.667] Length[0..Inf) Weight[0..159]
vehicle11 : Unused
vehicle12 : Unused
vehicle13 : Unused
vehicle14 : Unused
vehicle15 : Unused

*/
```

C H A P T E R

10

Pickup and Delivery by Multiple Vehicles from Multiple Depots

In this lesson, you will learn how to:

- decompose a problem
- use a mixed iterative solving technique
- maintain the coherence of submodels

The problem of pickup and delivery by multiple vehicles from multiple depots is known as the Multiple-Depot Pickup and Delivery Problem (MDPDP). For many distribution companies in the real world, customers are widely geographically separated, making pickup or delivery from a single depot too costly. A possible solution is to open anything from a few to many depots as “home bases” for vehicles. This problem shows you how to model and solve a problem where there is more than one vehicle depot. In this problem each visit must be assigned to a particular truck. Each truck is itself assigned to a particular depot. The aim is still to minimize the global cost of the routes serving all the visits. Moreover, since MDPDP problems are usually large, this problem aims to increase performance by decomposing the problem into smaller subproblems, bringing them all together for a complete solution.

Describe

In terms of the specification of the problem, a multiple depot pickup and delivery problem is much the same as a standard single depot problem. The main difference is that not all vehicles are located at the same node. Multiple depot problems may also have some properties which are local to each depot, for instance opening hours, although such variations are not addressed here. The constraints here are the usual ones of vehicle capacity and time windows. The object is to minimize the total distance traveled.

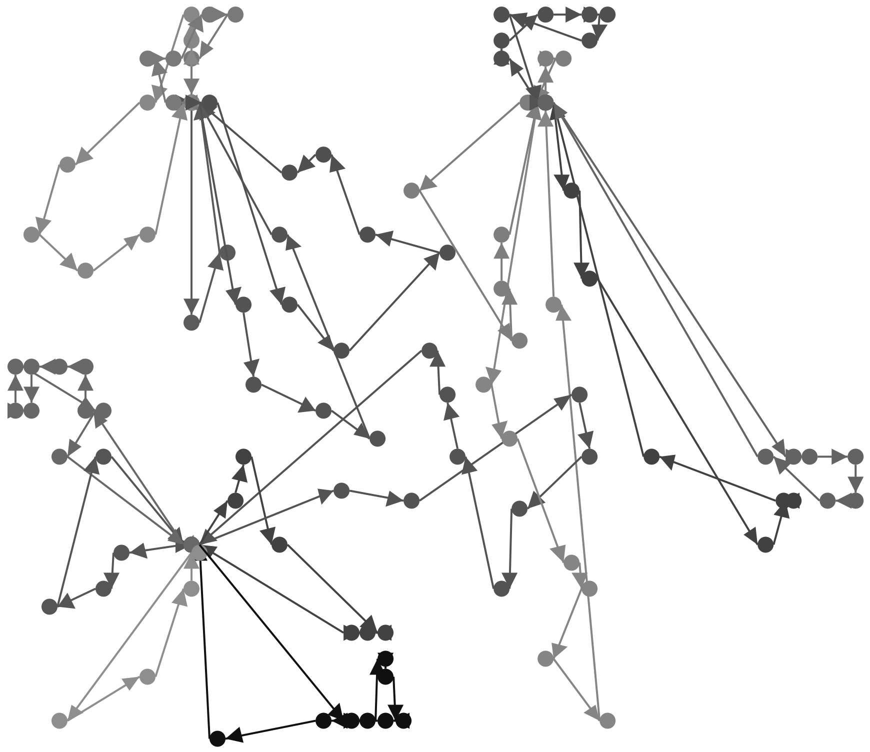

Figure 10.1 *Example of a Solution for a Multiple Depot Pickup and Delivery Problem*

The MDPDP problem is broken down to a set of subproblems using Concert Technology: one subproblem per depot. Each subproblem is then improved independently. This has the advantage that the whole problem can be solved more quickly. However, there is a disadvantage. By decomposing the problem, the quality of solution obtained can be poorer than if the problem were considered as a whole. This problem, therefore, uses a mixed technique where each subproblem is improved independently and then the whole problem is improved. The process is then iterated with the subproblems being improved once more in light of the changes made while optimizing the whole problem. This process continues until some stopping condition is met. In this example, the stopping condition is that no more improvement takes place. This process is shown in the following figure:

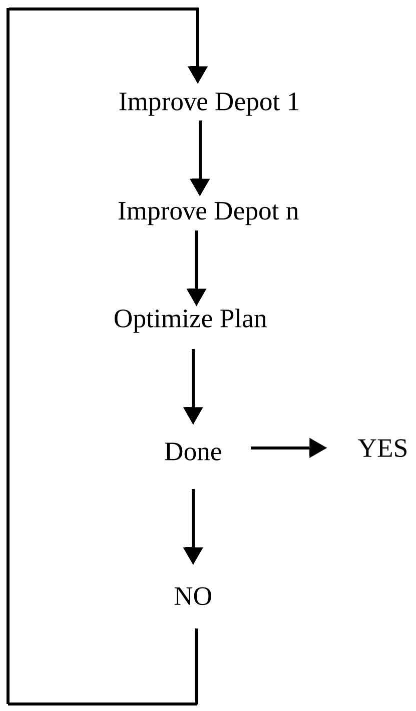

Figure 10.2 *Solution Process Using Subproblems*

To perform the decomposition of the problem into subproblems, one model, an instance of `IloModel`, per subproblem is maintained. Each of these models contains the dimensions of the problem and the vehicles particular to the depot in question. In addition, each such model contains constraints particular to the depot, such as the time constraints on vehicles based at that depot. A model of the whole problem, containing the depot models, is also maintained. The assignment of depots to visits is not known in advance, and, in fact, may change during the improvement process. It will thus be necessary to add visits to and remove visits from models. To ease this process, a small model is created for each visit which holds the visit along with constraints pertaining to the visit itself (for instance, time windows). In this way, visits, together with their pertinent constraints, can be transferred from model to model. A reference to this model is stored in the visit itself using `IloVisit::setObject(IloAny)`. A model is also created to manage the dimensions used by all models.

The solution process is iterative. Each depot model is improved and then the whole problem is improved. This latter phase is one that can cause visits to migrate from one depot to another. In other words, moves can be performed that cause a vehicle from a different depot to perform a visit. When the next iteration to improve depots begins, a synchronization needs to be performed. During this synchronization, the submodels are updated with the changes made during the improvement stage of the whole problem.

This problem demonstrates three main techniques of Dispatcher problem solving: the decomposition of a problem, a mixed iterative solving technique, and the maintenance of the coherence of submodels.

Step 1 Describe the problem

The first step is to write a natural language description of the problem.

The components of the routing model for this problem are the same as for a standard PDP, except that there are multiple depots. Vehicles are associated with a specific depot.

The constraints in this problem are the same as those in a standard PDP: vehicle capacity, time windows, and visit quantities.

The objective is to minimize the total cost of the solution, which is directly proportional to the total distance traveled by all vehicles.

Model

Once you have written a description of your problem, you can use Dispatcher classes to model it. This solution process is highly depot based. It is worth introducing a class for depots which encapsulates the composition of the depot, along with methods for improving the solution for a depot. You will also create a `RoutingModel` class, as in a standard PDP.

Step 2 Open the example file

Open the example file `YourDispatcherHome/examples/src/tutorial/mdpdp_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 of this lesson, you will have completed the code for the problem and you will be able to compile and run it.

Declare the Depot class

The code for the declaration of the Depot class is provided for you:

```
class Depot {
private:
  IloEnv            _env;
  IloNode           _node;
  IloInt            _nbOfTrucks;
  IloInt            _capacity;
  IloNum            _openTime;
  IloNum            _closeTime;
  IloVehicleArray   _vehicles;
  IloVisitArray     _visits;
  IloModel          _model;
  IloNHood          _nhood;
  IloMetaHeuristic  _mh;
public:
  Depot(IloEnv env, const char *name, IloNum x, IloNum y,
        IloInt nbOfTrucks, IloInt capacity,
        IloNum openTime, IloNum closeTime
        );
  ~Depot();

  const char* getName() const { return _node.getName();}
  void add(IloExtractable ex) { _model.add(ex); }
  void add(IloVehicle vehicle) { _vehicles.add(vehicle); _model.add(vehicle); }
  IloModel getModel() const { return _model; }
  IloBool improve(IloSolver solver, IloRoutingSolution rs, IloGoal g);
  void fillModel(IloRoutingSolution rs);

  void createVehicles(IloDimension2 time, IloDimension2 length, IloDimension1
weight);
  IloVehicle createOneVehicle(IloInt vehicleIndex,
                              IloDimension2 time,
                              IloDimension2 length,
                              IloDimension1 weight);
};
```

Define the Depot constructor

A depot is built from an instance of `IloEnv` and is given a name `name`. The following code is provided for you:

```
Depot
::Depot(IloEnv env, const char *name, IloNum x, IloNum y,
        IloInt nbOfTrucks, IloInt capacity,
        IloNum openTime, IloNum closeTime
        )
  : _env(env),
    _node(env, x, y, 0, name),
    _nbOfTrucks(nbOfTrucks),
    _capacity(capacity),
    _openTime(openTime),
    _closeTime(closeTime),
    _vehicles(env),
    _visits(env),
    _model(env)

{
```

In order to improve solutions for this depot, a neighborhood `_nhood` and greedy metaheuristic `_mh` are created. As the subproblem of improving the solution to a single depot is likely to be relatively small, five move operators are used in the neighborhood.

Step 3 Create the depot neighborhood

Add the following code after the comment `//Create the depot neighborhood`

```
    _nhood = IloTwoOpt(env) + IloOrOpt(env) + IloRelocate(env) +
           IloCross(env) + IloExchange(env)
    //+ IloRelocateTours(env)
    ;
```

Step 4 Create the depot search heuristic

Add the following code after the comment `//Create the depot search heuristic`

```
   _mh = IloImprove(env);
}
```

Define the Depot destructor

A destructor cleans up the temporary objects. This code is provided for you:

```
Depot::~Depot() {
  _nhood.end();
  _mh.end();
  _vehicles.end();
  _visits.end();
}
```

Define the Depot::improve function

You define a method `improve` that is used for improving the solution `rs` passed as an argument. A solver `solver` and a goal `g` to be executed at each move are also passed as parameters. The number of moves made is returned. First, the neighborhood and greedy metaheuristic are reset. Then, the improvement goal is created using `IloSingleMove`. An improvement loop is then entered, which stops when no move can improve the cost of the current solution. Finally, the number of moves made is returned. The following code is provided for you:

```
IloBool Depot::improve(IloSolver solver, IloRoutingSolution rs, IloGoal g) {
  _nhood.reset();
  _mh.reset();
  IloGoal improve = IloSingleMove(_env, rs, _nhood, _mh, g);
  solver.out() << "  Optimizing depot " << getName() << " "
               << rs.getSolution().getObjectiveValue() << flush;
  IloBool moves = 0;
  while (solver.solve(improve)) ++moves;
  solver.out() << " ---> " << rs.getSolution().getObjectiveValue() << endl;
  return (moves>0);
}
```

Define the Depot::fillModel function

The class `Depot` also requires the ability to synchronize its internal model with a routing solution. To perform this you first empty the model of the visit models which previously comprised the model. You then scan the solution for the vehicles which are based at this depot, and for each one, find out which visits are made by these vehicles. These visits (or more accurately, the models pertaining to these visits) must then be placed in the depot model. You keep track of which visits are in the model in the array `_visits`.

Step 5 Fill the depot model

Add the following code after the comment `//Fill the depot model`

```
  void Depot::fillModel(IloRoutingSolution rs) {
  IloInt i;
  for (i = 0; i < _visits.getSize(); i++)
    _model.remove((IloModelI *)_visits[i].getObject());
  _visits.end();
  _visits = IloVisitArray(_env);
  for (i = 0; i < _vehicles.getSize(); i++) {
    for (IloRoutingSolution::RouteIterator r(rs, _vehicles[i]); r.ok(); ++r) {
      IloVisit visit = *r;
      if ( !visit.isFirstVisit() && !visit.isLastVisit() ) {
        _visits.add(visit);
        _model.add( (IloModelI*)visit.getObject() );
      }
    }
  }
}
```

Define the Depot::createOneVehicle function

You add a function to create a vehicle associated to the depot. The vehicles have first and last visits at their associated depot. You add side constraints that the vehicles must leave the depot after it opens and return to the depot before it closes. You set the capacity of the vehicle. The cost of each vehicle is set to be directly proportional to the dimension `length`.

Step 6 Create one vehicle associated to the depot

Ad the following code after the comment
`//Create one vehicle associated to the depot`

```
IloVehicle Depot::createOneVehicle(IloInt vehicleIndex,
                                   IloDimension2 time,
                                   IloDimension2 length,
                                   IloDimension1 weight) {
  char namebuf[128];
  const char* depotName = getName();

  sprintf(namebuf, "Truck %ld leaving %s", vehicleIndex, depotName);
  IloVisit first(_node, namebuf);
  sprintf(namebuf, "Truck %ld returning to %s", vehicleIndex, depotName);
  IloVisit last (_node, namebuf);
  sprintf(namebuf, "Vehicle %ld of Depot %s", vehicleIndex, getName());
  IloVehicle vehicle(first, last, namebuf);
  vehicle.setCost(length, 1.0);
  vehicle.setCapacity(weight, _capacity);
  add(vehicle);
  add(first.getCumulVar(time) >= _openTime);
  add(last.getCumulVar(time)  <= _closeTime);
  last.getCumulVar(weight).setBounds(0,0);
  first.getCumulVar(length).setBounds(0,0);
  first.getDelayVar(length).setBounds(0,0);
  first.getWaitVar(length).setBounds(0,0);
  last.getDelayVar(length).setBounds(0,0);
  last.getWaitVar(length).setBounds(0,0);

  vehicle.setObject(this);
  return vehicle;
}
```

Define the Depot::createVehicles function

You create a function that will be called by `RoutingModel::createDepots`. This loop will be used to create the vehicles attached to each depot. This code is provided for you:

```
void Depot::createVehicles(IloDimension2 time,
                           IloDimension2 length,
                           IloDimension1 weight) {
  for (IloInt v=0; v < _nbOfTrucks; ++v) {
    IloVehicle vehicle = createOneVehicle(v, time, length, weight);
    _vehicles[v] = vehicle;
  }
}
```

Declare the RoutingModel class

As in Chapter 7, *Pickup and Delivery Problems*, you create a `RoutingModel` class, which is used to model the problem. There are several important differences between the `RoutingModel` class used in a standard PDP and the `RoutingModel` class used in this problem. There is a model for the whole problem `_mdl` and a model for the dimensions `_dimModel`. Instead of a `createVehicles` function, you have a `createDepots` function. The `createDepots` function will call the `Depot` class constructor and create the submodel for each depot. The `createVisits` function creates small submodels for each pickup and delivery visit. The `RoutingModel` class uses Concert Technology's ability to maintain nested submodels to maintain the coherence of the model for the whole problem and the submodels for each depot, the submodels for each pickup and delivery visit, and the submodel for dimensions. The code for the declaration of the class `RoutingModel` is provided for you:

```
class RoutingModel {
  IloEnv              _env;
  IloModel            _mdl;
  IloModel            _dimModel;
  IloDistance         _distance;
  IloDimension2       _time;
  IloDimension2       _length;
  IloDimension1       _weight;
  const char* _depotPath; //  "../../../examples/data/mdvrp/depots.csv";
  const char* _visitPath; //  "../../../examples/data/mdvrp/vrp100.csv";
  const char* _nodePath;  //  "../../../examples/data/mdvrp/node100.csv";

  IloInt _nbOfDepots;
  Depot** _depots;
protected:
  void createDimensions();
  void createNodes (const char* nodePath);
  void createDepots(const char* depotPath);
  void createVisits(const char* orderPath);
public:
  RoutingModel(IloEnv env);
  ~RoutingModel() {}

  IloEnv    getEnv() const { return _env; }
  IloModel  getModel() const { return _mdl; }
  IloInt getNumberOfDepots() const { return _nbOfDepots;}
  Depot* getDepot(IloInt d) const {
    assert( d >= 0 );
    assert( d < _nbOfDepots);
    return _depots[d];
  }
  void parse(int argc, char** argv);
  void init();
  void createModel();

};
```

Define the RoutingModel constructor

The constructor is defined on the environment `env`. This constructor will be called from the `main` function. This code is provided for you:

```
RoutingModel::RoutingModel(IloEnv env)
  : _env(env),
    _depotPath("../../../examples/data/mdpdp/depots.csv"),
    _visitPath("../../../examples/data/mdpdp/vrp100.csv"),
    _nodePath ("../../../examples/data/mdpdp/node100.csv"),
    _nbOfDepots(0),
    _depots(0)
{
}
```

Define the RoutingModel::parse function

This member function allows you to specify the names of the input files using command line syntax. If you do not specify input files, the defaults in the `RoutingModel` constructor will be used. This member function is called from the `main` function.

Step 7 Parse the input files

Add the following code after the comment `//Parse the input files`

```
  void RoutingModel::parse(int argc, char** argv) {
  if ( argc >= 4 ) {
    _depotPath = argv[1];
    _visitPath = argv[2];
    _nodePath  = argv[3];
  }
}
```

Define the RoutingModel::createModel function

You define a member function to create the model `_mdl` for the whole problem. It calls the functions that create the dimensions, nodes, depots, and visits. This function will be called from the `main` function.

Step 8 Create the model

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

```
void RoutingModel::createModel() {
  _mdl = IloModel(_env);
  createDimensions();
  createNodes(_nodePath);
  createDepots(_depotPath);
  createVisits(_visitPath);
}
```

Define the RoutingModel::createDimensions function

Now, you declare three dimensions: `_time` and `_length`, computed as Euclidean distance, and `_weight`, used for capacity constraints. Note that Euclidean distance is used here for both time and length, but that other kinds of distance may be used as needed. As the model for each depot will require dimensions to be added, you keep the dimensions together in a model called `_dimModel`.

Step 9 Add the dimensions

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

```
void RoutingModel::createDimensions() {
  _dimModel = IloModel(_env);

  _distance = IloDistance(_env, IloEuclidean);
  _weight = IloDimension1(_env, "weight");
  _dimModel.add(_weight);

  _time = IloDimension2(_env, _distance , "time");
  _dimModel.add(_time);

  _length = IloDimension2(_env, _distance , "distance");
  _dimModel.add(_length);

  _mdl.add(_dimModel);
}
```

Define the RoutingModel::createNodes function

The `createNodes` function is defined as in *Lesson 7 Pickup and Delivery Problem*s. You use Concert Technology's csv reader functionality to input node data and node coordinate data from csv files. Here is the complete code for defining the `createNodes` function:

```
void RoutingModel::createNodes(const char* nodePath) {
  IloCsvReader nodeReader(_env, nodePath);
  for (IloCsvReader::LineIterator it(nodeReader); it.ok(); ++it) {
    IloCsvLine line = *it;
    const char* name = line.getStringByHeader("name");
    IloNode node(_env,
                 line.getFloatByHeader("x"),
                 line.getFloatByHeader("y"),
                 0, // no Z coordinate here.
                 name);
    node.setKey(name);
  }
  nodeReader.end();
}
```

Define the RoutingModel::createDepots function

You use csv reader functionality to input depot data from csv files. For each of the depots, you get the number of trucks per depot (`nbOfTrucks`), its coordinates (`x` and `y`), and its opening and closing times (`openTime` and `closeTime`). An array of pointers to class type `Depot` are created. This array is then populated by a loop that creates the depots. For each depot, the function `Depot::createVehicles` is called to create the vehicles associated with each depot and the model containing all dimensions `_dimModel` is added to the depot model. Finally, each depot model is added to the model of the whole problem `_mdl`.

Step 10 Create the depot

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

```
void RoutingModel::createDepots(const char* depotPath) {
  IloCsvReader depotReader(_env, depotPath);
  _nbOfDepots = depotReader.getNumberOfItems();
  _depots = new (_env) Depot* [ _nbOfDepots ];

  IloInt depotIndex = 0;
  for (IloCsvReader::LineIterator it(depotReader); it.ok(); ++it, ++depotIndex)
{
    IloCsvLine line = *it;
    const char* name = line.getStringByHeader("name");
    IloNum x = line.getFloatByHeader("x");
    IloNum y = line.getFloatByHeader("y");
    IloNum openTime   = line.getFloatByHeader("openTime");
    IloNum closeTime  = line.getFloatByHeader("closeTime");
    IloInt nbOfTrucks = line.getIntByHeader("nbOfTrucks");
    IloInt capacity   = line.getIntByHeader("capacity");

    Depot* depot =
      new (_env) Depot(_env, name, x, y, nbOfTrucks, capacity, openTime,
closeTime);
    _depots[ depotIndex ] = depot;
  }
  depotReader.end();
  IloInt d;
  for (d=0; d < _nbOfDepots; ++d) {
    Depot* depot = _depots[d];
    depot->createVehicles(_time, _length, _weight);
    depot->add(_dimModel);
    _mdl.add(depot->getModel());
  }

}
```

Define the RoutingModel::createVisits function

You use csv reader functionality to input visits from csv files. You then create a node for each pickup and delivery visit. This code is provided for you:

```
void RoutingModel::createVisits(const char* visitPath) {
  IloCsvReader orderReader(_env, visitPath);
  IloCsvReader::LineIterator it(orderReader);
  while ( it.ok() ) {
    IloCsvLine line = *it;
    //read visit data from files
    char * pickupVisitName = line.getStringByHeader("pickup");
    char * pickupNodeName = line.getStringByHeader("pickupNode");
    char * deliveryVisitName = line.getStringByHeader("delivery");
    char * deliveryNodeName = line.getStringByHeader("deliveryNode");
    IloNum quantity = line.getFloatByHeader("quantity");
    IloNum pickupMinTime  = line.getFloatByHeader("pickupMinTime");
    IloNum pickupMaxTime  = line.getFloatByHeader("pickupMaxTime");
    IloNum deliveryMinTime  = line.getFloatByHeader("deliveryMinTime");
    IloNum deliveryMaxTime  = line.getFloatByHeader("deliveryMaxTime");
    IloNum dropTime = line.getFloatByHeader("dropTime");
    IloNum pickupTime = line.getFloatByHeader("pickupTime");
    //read data nodes from the file nodes.csv and creat pickup and delivery
nodes
    IloNode pickupNode = IloNode::Find(_env, pickupNodeName);
    IloNode deliveryNode = IloNode::Find(_env, deliveryNodeName);
```

You next create the submodel for each pickup and delivery visit.

Step 11 Create visit models

Add the following code after the comment `//Create visit models`

```
    IloModel pickupModel(_env);
    IloModel deliveryModel(_env);
```

You create a pickup visit and add the constraints. Each visit takes a certain amount of time (`pickupTime`) and involves the delivery of a certain `quantity` of good. It must be performed within a given time window. The visit is added to the submodel `pickupModel`.

Step 12 Create pickup

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

```
    IloVisit pickup(pickupNode, pickupVisitName);
    pickupModel.add(pickup.getDelayVar(_time) == pickupTime);
    pickupModel.add(pickup.getTransitVar(_weight) == quantity);
    pickupModel.add(pickupMinTime <= pickup.getCumulVar(_time) <=
pickupMaxTime);
    pickupModel.add(pickup);
    pickup.setObject( (IloAny)pickupModel.getImpl() );
    pickup.getDelayVar(_length).setBounds(0,0);
    pickup.getWaitVar(_length).setBounds(0,0);
```

You follow the same procedure to create the `delivery` visit, add constraints, and add the delivery visit to the submodel `deliveryModel`.

Step 13 Create delivery

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

```
    IloVisit delivery(deliveryNode, deliveryVisitName);
    deliveryModel.add(delivery.getDelayVar(_time) == dropTime);
    deliveryModel.add(delivery.getTransitVar(_weight) == -quantity);
    deliveryModel.add(deliveryMinTime <= delivery.getCumulVar(_time) <=
deliveryMaxTime);
    deliveryModel.add(delivery);
    delivery.getDelayVar(_length).setBounds(0,0);
    delivery.getWaitVar(_length).setBounds(0,0);
```

Finally, you add the pickup and delivery order constraint to the `pickupModel`. A reference to the `deliveryModel` is placed via `delivery.setObject`. This means that the submodel for a delivery can be retrieved from the delivery at any time. Finally, each `pickupModel` and `deliveryModel` are added to the model of the whole problem `_mdl`.

Step 14 Order the pickup and delivery

Add the following code after the comment `//Order the pickup and delivery`

```
    pickupModel.add( IloOrderedVisitPair(_env, pickup, delivery) );
    delivery.setObject( (IloAny)deliveryModel.getImpl() );
    _mdl.add( pickupModel );
    _mdl.add( deliveryModel );
    ++it;
  }
  orderReader.end();
}
```

Solve

As in Chapter 7, *Pickup and Delivery Problems*, you search for a solution using the two-phase approach of generating a first solution and then improving it. However, in this example the solution process is iterative. Each depot model is improved and then the whole problem is improved. When the next iteration to improve depots begins, a synchronization needs to be performed. During this synchronization, the submodels are updated with the changes made during the improvement stage of the whole problem.

Here is a short description of the proposed heuristic:

1. Find a first solution.
2. Improve the solution at each depot.
3. Synchronize the solution with the full model.
4. Improve the whole routing plan. This phase is one that can cause visits to migrate from one depot to another. In other words, moves can be performed that cause a vehicle from a different depot to perform a visit.
5. Return to Step 2 until no further improvement is possible.
6. Do a final solution synchronization.

Declare the RoutingSolver class

The code for the declaration of the class `RoutingSolver` is provided for you:

```
class RoutingSolver {
  RoutingModel&        _routing;
  IloModel            _mdl;
  IloSolver           _solver;
  IloDispatcher       _dispatcher;
  IloRoutingSolution  _solution;
  IloGoal             _instantiateCost;
  IloGoal             _restoreSolution;
  IloGoal             _fsGoal;
  IloNHood            _nhood;
  IloMetaHeuristic    _greedy;
  IloGoal             _move;

public:
  RoutingSolver(RoutingModel& routingModel);
  ~RoutingSolver() {}

  IloEnv getEnv() const { return _mdl.getEnv();}
  void syncSolution(IloModel mdl, IloSolution s, IloGoal g);
  void syncSolution();
  IloBool findFirstSolution();
  IloBool improveDepots();
  IloBool improvePlan();
  void printInformation() const;
};
```

There are several important differences between the `RoutingSolver` class used in a standard PDP and the `RoutingSolver` class used in this problem. These include several member functions used in the iterative solution process: `syncSolution`, `improveDepots`, and `improvePlan`.

Define the RoutingSolver constructor

The `RoutingSolver` constructor takes an instance of `RoutingModel` as a parameter. The goal to instantiate cost is created using the function `IloDichotomize`. The goal to restore the solution is created using `IloRestoreSolution`. The goal used to find the first solution is created by the predefined first solution heuristic `IloNearestAdditionGenerate`. The nearest addition heuristic builds a first solution route by adding visits to the route. The visit added to the route is the visit closest—that is, the least costly to get to—to the end of the current partial route of the vehicle. After this visit is added, the heuristic finds the visit that is closest to the visit just added to the end of the current partial route, and so on until all visits are added. For more information about `IloNearestAdditionGenerate`, see Appendix A, *Predefined First Solution Heuristics*. The neighborhood `_nhood` used for the improvement of the entire problem is smaller than that for the improvement of a depot. Here only `IloRelocate` and `IloExchange` are used to increase the solving performance on this large problem. A greedy search heuristic is used. This constructor will be called from the `main` function. This code is provided for you:

```
 RoutingSolver::RoutingSolver(RoutingModel& routingModel)
  : _routing( routingModel),
    _mdl(routingModel.getModel()),
    _solver(routingModel.getModel()),
    _dispatcher(_solver),
    _solution(routingModel.getModel())
{
  IloEnv env = getEnv();
  _instantiateCost =
    IloDichotomize(env, _dispatcher.getCostVar(), IloFalse);
  _restoreSolution = IloRestoreSolution(env, _solution);
  _fsGoal = IloNearestAdditionGenerate(env);

  _nhood = IloRelocate(env) + IloExchange(env);
  _greedy = IloImprove(env);

}
```

Define the RoutingSolver::findFirstSolution function

The function `findFirstSolution` searches for a solution using the goals `_fsGoal` and `_instantiateCost`. If successful, the solution is stored. If no first solution is found, the program reports that it was unsuccessful in finding a first solution. All other parts of the program remain unchanged. The following code is provided for you:

```
IloBool RoutingSolver::findFirstSolution() {
  if (!_solver.solve(_fsGoal && _instantiateCost)) {
    _solver.error() << "Infeasible Routing Plan" << endl;
    return IloFalse;
  }
  _solution.store(_solver);
  _solver.out() << "First solution with cost: "
                << _solution.getObjectiveValue()
                << endl;
  return IloTrue;
}
```

You do not need one solution per depot. One solution to the whole problem is sufficient. You will simply extract the part of it that you wish to work on. This is possible for two reasons. First, operations such as `store` and `restore` on solutions ignore parts of the solution that are not extracted. Therefore, a solution can cover more variables than have been extracted with no problems. Second, Dispatcher's neighborhoods (such as `IloRelocate`) ignore nonextracted parts of the solution and only generate neighbors for extracted parts. The result is that you can perform local search over an extracted part of the routing solution, while the remainder remains unchanged.

> ***Note:*** *There is one part of the solution that will always be extracted, regardless of the part of the solution being optimized: the objective. Thus, if you extract a submodel and restore the corresponding part of a routing solution, the stored value of the objective reflects only the cost of the submodel and not the whole solution. This is a situation where the cost value of a solution must be treated with care. The function* `syncSolution` *ensures that the cost value is consistent with that computed from the extracted model.*

Define the syncSolution functions

You define a function that takes the model for the whole problem `_mdl` and a solution, and synchronizes the solution with the model. As discussed, the objective stored in the solution may be inconsistent with the model you wish to work on. You correct this with the following function which extracts the model passed, then restores and stores the solution. A subgoal is also executed. After storing, the solution's objective is consistent with the model. This function is used in the improvement iteration phase.

Step 15 Synchronize the solution

Add the following code after the comment `//Synchronize the solution`

```
void RoutingSolver::syncSolution(IloModel mdl, IloSolution s, IloGoal g) {
  _solver.extract(mdl);
  if ( !_solver.solve( IloRestoreSolution( _solver.getEnv(), s) && g))
    _solver.out() << "Synchronization failed" << endl;
  else
    s.store(_solver);
}
```

The following member function is called after the iterative improvements have ended to do a final solution synchronization. This code is provided for you:

```
void RoutingSolver::syncSolution() {
  syncSolution(_mdl, _solution, _instantiateCost);
}
```

Define the RoutingSolver::improveDepots function

If a first solution is found, the solution is improved using local search. You iterate over all depots, using `Depot::fillModel` to populate each model. You then update the cost of the global solution according to the model of this depot. You use `Depot::improve` to attempt to reduce the cost of the routing for this depot. Finally, the solution is synchronized with the global model.

Step 16 Improve the depots

Add the following code after the comment `//Improve the depots`

```
IloBool RoutingSolver::improveDepots() {
  IloEnv env = getEnv();
  _move =
    IloSingleMove(env, _solution, _nhood, _greedy, _instantiateCost);

  _solver.out() << endl << "Improvement loop" << endl;
  _solver.out() << "================" << endl;
  IloBool improved = IloTrue;
  IloInt nbOfDepots = _routing.getNumberOfDepots();
  improved = IloFalse;
  for (IloInt d = 0; d < nbOfDepots; ++d) {
    Depot* depot = _routing.getDepot(d);
    depot->fillModel(_solution);
    syncSolution(depot->getModel(), _solution, _instantiateCost);
    if ( depot->improve(_solver, _solution, _instantiateCost) ) {
      improved = IloTrue;
    }
  }

  // sync back to full model.
  syncSolution(_mdl, _solution, _instantiateCost);
  return improved;
}
```

Define the RoutingSolver::improvePlan function

After the routing for all depots has been improved independently, the routing for the problem as a whole is improved. After resetting the neighborhood and greedy metaheuristic, an improvement loop is entered to improve the cost of all depots using the goal `_move`. If there was some improvement, then you go back to an independent improvement of each depot. Otherwise, you leave the main improvement loop.

Step 17 Improve the routing plan

Add the following code after the comment `//Improve the routing plan`

```
IloBool RoutingSolver::improvePlan() {
  _solver.out() << "Optimizing plan "
                << _solution.getSolution().getObjectiveValue()
                << flush;
  _nhood.reset();
  _greedy.reset();
  IloInt nbImproved = 0;
  while ( _solver.solve(_move) ) { ++nbImproved;}
  _solver.out() << " ---> "
               << _solution.getSolution().getObjectiveValue() << endl;
  return ( nbImproved > 0 );
}
```

Define the RoutingSolver::printInformation function

The `printInformation` function is the same as in Chapter 7, *Pickup and Delivery Problems*.

Define the main function

After you finish creating the `Depot`, `RoutingModel`, and `RoutingSolver` classes, you use them in the `main` function. You can use command line syntax to pass the names of input files to the model. If you do not specify input files, the defaults will be used. In the `main` function, you first create an environment. Then you create an instance of the `RoutingModel` class and parse the input files. You create the model for the whole problem with the member function `RoutingModel::createModel`. You create an instance of the `RoutingSolver` class. You find a first solution and then use iterative improvement to improve the depots and the routing plan. You synchronize the final solution and print this solution. The following code is provided for you:

```
int main(int argc, char* argv[]) {
  IloEnv env;
  try {
    RoutingModel routingModel(env);
    routingModel.parse(argc, argv);
    routingModel.createModel();

    RoutingSolver rsolver(routingModel);
    if ( rsolver.findFirstSolution() ) {
      IloBool improved = IloTrue;
      do {
        improved =
          rsolver.improveDepots() && rsolver.improvePlan();
      } while ( improved );
    }
    rsolver.syncSolution();
    rsolver.printInformation();
  } catch (IloException& ex) {
    env.out() << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Step 18 Compile and run the program

Compile and run the program. The first solution has a cost of 1397.52 units. After three improvement loops, the solution has a cost of 983.385 units. The solution uses 15 vehicles:

```
Number of fails              : 0
Number of choice points      : 0
Number of variables          : 7354
Number of constraints        : 1280
Reversible stack (bytes)     : 944724
Solver heap (bytes)          : 2509244
Solver global heap (bytes)   : 2446204
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 20176
Total memory used (bytes)    : 5988760
Elapsed time since creation  : 0.461
Number of nodes              : 103
Number of visits             : 260
Number of vehicles           : 30
Number of dimensions         : 3
Number of accepted moves     : 141
===============
Cost         : 983.385
Number of vehicles used : 15
First solution with cost: 1397.52

Improvement loop
================
  Optimizing depot Depot 0 509.991 ---> 411.418
  Optimizing depot Depot 1 385.144 ---> 351.056
  Optimizing depot Depot 2 502.382 ---> 345.504
Optimizing plan 1107.98 ---> 1073.71

Improvement loop
================
  Optimizing depot Depot 0 412.393 ---> 391.898
  Optimizing depot Depot 1 325.838 ---> 323.948
  Optimizing depot Depot 2 335.482 ---> 310.54
Optimizing plan 1026.39 ---> 992.007

Improvement loop
================
  Optimizing depot Depot 0 385.09 ---> 378.142
  Optimizing depot Depot 1 323.948 ---> 323.948
  Optimizing depot Depot 2 282.968 ---> 281.295
  Optimizing plan 983.385 ---> 983.385
```

The complete program and output are listed in “Complete Program” on page 264. You can also view it online in the `YourDispatcherHome/examples/src/mdpdp.cpp` file.

Review Exercises

1. Why do you only need one solution for the whole problem?
2. Which part of the solution will always be extracted, regardless of the part of the solution being optimized?
3. How is a reference to a visit model stored in the visit itself?

Suggested Answers

Exercise 1

Why do you only need one solution for the whole problem?

Suggested Answer

One solution to the whole problem is sufficient. You will simply extract the part of it that you wish to work on. This is possible for two reasons. First, operations such as `store` and `restore` on solutions ignore parts of the solution that are not extracted. Therefore, a solution can cover more variables than have been extracted with no problems. Second, Dispatcher's neighborhoods (such as `IloRelocate`) ignore nonextracted parts of the solution and only generate neighbors for extracted parts. The result is that you can perform local search over an extracted part of the routing solution, while the remainder remains unchanged.

Exercise 2

Which part of the solution will always be extracted, regardless of the part of the solution being optimized?

Suggested Answer

The objective will always be extracted. Thus, if you extract a submodel and restore the corresponding part of a routing solution, the stored value of the objective reflects only the cost of the submodel and not the whole solution. This is a situation where the cost value of a solution must be treated with care. The function `syncSolution` ensures that the cost value is consistent with that computed from the extracted model.

Exercise 3

How is a reference to a visit model stored in the visit itself?

Suggested Answer

A pointer to the model implementation is stored in the object field using `IloVisit::setObject(IloAny)`.

Complete Program

The complete program for `mpdpd.cpp` follows. You can also view it online in the file `YourDispatcherHome/examples/src/mdpdp.cpp`.

```
// -------------------------------------------------------------- -*- C++ -*-
// File: examples/src/mdpdp.cpp
// --------------------------------------------------------------------------


#include <ildispat/ilodispatcher.h>

ILOSTLBEGIN

class Depot {
private:
  IloEnv             _env;
  IloNode            _node;
  IloInt             _nbOfTrucks;
  IloInt             _capacity;
  IloNum             _openTime;
  IloNum             _closeTime;
  IloVehicleArray    _vehicles;
  IloVisitArray      _visits;
  IloModel           _model;
  IloNHood           _nhood;
  IloMetaHeuristic   _mh;

public:
  Depot(IloEnv env, const char *name, IloNum x, IloNum y,
        IloInt nbOfTrucks, IloInt capacity,
        IloNum openTime, IloNum closeTime
        );
  ~Depot();

  const char* getName() const { return _node.getName();}
  void add(IloExtractable ex) { _model.add(ex); }
  void add(IloVehicle vehicle) { _vehicles.add(vehicle); _model.add(vehicle); }
  IloModel getModel() const { return _model; }
  IloBool improve(IloSolver solver, IloRoutingSolution rs, IloGoal g);
  void fillModel(IloRoutingSolution rs);

  void createVehicles(IloDimension2 time, IloDimension2 length, IloDimension1
weight);
  IloVehicle createOneVehicle(IloInt vehicleIndex,
                              IloDimension2 time,
                              IloDimension2 length,
                              IloDimension1 weight);
};
```

```
Depot
::Depot(IloEnv env, const char *name, IloNum x, IloNum y,
        IloInt nbOfTrucks, IloInt capacity,
        IloNum openTime, IloNum closeTime
        )
  : _env(env),
    _node(env, x, y, 0, name),
    _nbOfTrucks(nbOfTrucks),
    _capacity(capacity),
    _openTime(openTime),
    _closeTime(closeTime),
    _vehicles(env),
    _visits(env),
    _model(env)

{

  _nhood = IloTwoOpt(env) + IloOrOpt(env) + IloRelocate(env) +
           IloCross(env) + IloExchange(env)
    //+ IloRelocateTours(env)
    ;

  _mh = IloImprove(env);
}

Depot::~Depot() {
  _nhood.end();
  _mh.end();
  _vehicles.end();
  _visits.end();
}

IloBool Depot::improve(IloSolver solver, IloRoutingSolution rs, IloGoal g) {
  _nhood.reset();
  _mh.reset();
  IloGoal improve = IloSingleMove(_env, rs, _nhood, _mh, g);
  solver.out() << “  Optimizing depot “ << getName() << “ “
               << rs.getSolution().getObjectiveValue() << flush;
  IloBool moves = 0;
  while (solver.solve(improve)) ++moves;
  solver.out() << “ ---> “ << rs.getSolution().getObjectiveValue() << endl;
  return (moves>0);
}

void Depot::fillModel(IloRoutingSolution rs) {
  IloInt i;
  for (i = 0; i < _visits.getSize(); i++)
    _model.remove((IloModelI *)_visits[i].getObject());
  _visits.end();
  _visits = IloVisitArray(_env);
  for (i = 0; i < _vehicles.getSize(); i++) {
    for (IloRoutingSolution::RouteIterator r(rs, _vehicles[i]); r.ok(); ++r) {
      IloVisit visit = *r;
      if ( !visit.isFirstVisit() && !visit.isLastVisit() ) {
        _visits.add(visit);
        _model.add( (IloModelI*)visit.getObject() );
      }
    }
  }
}

void Depot::createVehicles(IloDimension2 time,
                           IloDimension2 length,
                           IloDimension1 weight) {
  for (IloInt v=0; v < _nbOfTrucks; ++v) {
    IloVehicle vehicle = createOneVehicle(v, time, length, weight);
    _vehicles[v] = vehicle;
  }
}

IloVehicle Depot::createOneVehicle(IloInt vehicleIndex,
                                   IloDimension2 time,
                                   IloDimension2 length,
                                   IloDimension1 weight) {
  char namebuf[128];
  const char* depotName = getName();

  sprintf(namebuf, “Truck %ld leaving %s”, vehicleIndex, depotName);
  IloVisit first(_node, namebuf);
  sprintf(namebuf, “Truck %ld returning to %s”, vehicleIndex, depotName);
  IloVisit last (_node, namebuf);
  sprintf(namebuf, “Vehicle %ld of Depot %s”, vehicleIndex, getName());
  IloVehicle vehicle(first, last, namebuf);
  vehicle.setCost(length, 1.0);
  vehicle.setCapacity(weight, _capacity);
  add(vehicle);
  add(first.getCumulVar(time) >= _openTime);
  add(last.getCumulVar(time)  <= _closeTime);
  last.getCumulVar(weight).setBounds(0,0);
  first.getCumulVar(length).setBounds(0,0);
  first.getDelayVar(length).setBounds(0,0);
  first.getWaitVar(length).setBounds(0,0);
  last.getDelayVar(length).setBounds(0,0);
  last.getWaitVar(length).setBounds(0,0);

  vehicle.setObject(this);
  return vehicle;
}

class RoutingModel {
  IloEnv              _env;
  IloModel            _mdl;
  IloModel            _dimModel;
  IloDistance         _distance;
  IloDimension2       _time;
  IloDimension2       _length;
  IloDimension1       _weight;
  const char* _depotPath; //  “../../../examples/data/mdvrp/depots.csv”;
  const char* _visitPath; //  “../../../examples/data/mdvrp/vrp100.csv”;
  const char* _nodePath;  //  “../../../examples/data/mdvrp/node100.csv”;

  IloInt _nbOfDepots;
  Depot** _depots;

protected:
  void createDimensions();
  void createNodes (const char* nodePath);
  void createDepots(const char* depotPath);
  void createVisits(const char* orderPath);

public:
  RoutingModel(IloEnv env);
  ~RoutingModel() {}

  IloEnv    getEnv() const { return _env; }
  IloModel  getModel() const { return _mdl; }
  IloInt getNumberOfDepots() const { return _nbOfDepots;}
  Depot* getDepot(IloInt d) const {
    assert( d >= 0 );
    assert( d < _nbOfDepots);
    return _depots[d];
  }

  void parse(int argc, char** argv);
  void init();
  void createModel();

};

RoutingModel::RoutingModel(IloEnv env)
  : _env(env),
    _depotPath("../../../examples/data/mdpdp/depots.csv"),
    _visitPath("../../../examples/data/mdpdp/vrp100.csv"),
    _nodePath ("../../../examples/data/mdpdp/node100.csv"),
    _nbOfDepots(0),
    _depots(0)
{
}

void RoutingModel::parse(int argc, char** argv) {
  if ( argc >= 4 ) {
    _depotPath = argv[1];
    _visitPath = argv[2];
    _nodePath  = argv[3];
  }
}

void RoutingModel::createModel() {
  _mdl = IloModel(_env);
  createDimensions();
  createNodes(_nodePath);
  createDepots(_depotPath);
  createVisits(_visitPath);
}

void RoutingModel::createDimensions() {
  _dimModel = IloModel(_env);

  _distance = IloDistance(_env, IloEuclidean);
  _weight = IloDimension1(_env, "weight");
  _dimModel.add(_weight);

  _time = IloDimension2(_env, _distance , "time");
  _dimModel.add(_time);
  _length = IloDimension2(_env, _distance , “distance”);
  _dimModel.add(_length);

  _mdl.add(_dimModel);
}

void RoutingModel::createNodes(const char* nodePath) {
  IloCsvReader nodeReader(_env, nodePath);
  for (IloCsvReader::LineIterator it(nodeReader); it.ok(); ++it) {
    IloCsvLine line = *it;
    const char* name = line.getStringByHeader(“name”);
    IloNode node(_env,
                 line.getFloatByHeader(“x”),
                 line.getFloatByHeader(“y”),
                 0, // no Z coordinate here.
                 name);
    node.setKey(name);
  }
  nodeReader.end();
}

void RoutingModel::createDepots(const char* depotPath) {
  IloCsvReader depotReader(_env, depotPath);
  _nbOfDepots = depotReader.getNumberOfItems();
  _depots = new (_env) Depot* [ _nbOfDepots ];

  IloInt depotIndex = 0;
  for (IloCsvReader::LineIterator it(depotReader); it.ok(); ++it, ++depotIndex)
{
    IloCsvLine line = *it;
    const char* name = line.getStringByHeader(“name”);
    IloNum x = line.getFloatByHeader(“x”);
    IloNum y = line.getFloatByHeader(“y”);
    IloNum openTime   = line.getFloatByHeader(“openTime”);
    IloNum closeTime  = line.getFloatByHeader(“closeTime”);
    IloInt nbOfTrucks = line.getIntByHeader(“nbOfTrucks”);
    IloInt capacity   = line.getIntByHeader(“capacity”);

    Depot* depot =
      new (_env) Depot(_env, name, x, y, nbOfTrucks, capacity, openTime,
closeTime);
    _depots[ depotIndex ] = depot;
  }
  depotReader.end();

  IloInt d;
  for (d=0; d < _nbOfDepots; ++d) {
    Depot* depot = _depots[d];
    depot->createVehicles(_time, _length, _weight);
    depot->add(_dimModel);
    _mdl.add(depot->getModel());
  }

}

void RoutingModel::createVisits(const char* visitPath) {
  IloCsvReader orderReader(_env, visitPath);
  IloCsvReader::LineIterator it(orderReader);
  while ( it.ok() ) {
    IloCsvLine line = *it;
    //read visit data from files
    char * pickupVisitName = line.getStringByHeader(“pickup”);
    char * pickupNodeName = line.getStringByHeader(“pickupNode”);
    char * deliveryVisitName = line.getStringByHeader(“delivery”);
    char * deliveryNodeName = line.getStringByHeader(“deliveryNode”);
    IloNum quantity = line.getFloatByHeader(“quantity”);
    IloNum pickupMinTime  = line.getFloatByHeader(“pickupMinTime”);
    IloNum pickupMaxTime  = line.getFloatByHeader(“pickupMaxTime”);
    IloNum deliveryMinTime  = line.getFloatByHeader(“deliveryMinTime”);
    IloNum deliveryMaxTime  = line.getFloatByHeader(“deliveryMaxTime”);
    IloNum dropTime = line.getFloatByHeader(“dropTime”);
    IloNum pickupTime = line.getFloatByHeader(“pickupTime”);
    //read data nodes from the file nodes.csv and creat pickup and delivery
nodes
    IloNode pickupNode = IloNode::Find(_env, pickupNodeName);
    IloNode deliveryNode = IloNode::Find(_env, deliveryNodeName);

    //create and add pickup and delivery visits
    IloModel pickupModel(_env);
    IloModel deliveryModel(_env);

    IloVisit pickup(pickupNode, pickupVisitName);
    pickupModel.add(pickup.getDelayVar(_time) == pickupTime);
    pickupModel.add(pickup.getTransitVar(_weight) == quantity);
    pickupModel.add(pickupMinTime <= pickup.getCumulVar(_time) <=
pickupMaxTime);
    pickupModel.add(pickup);
    pickup.setObject( (IloAny)pickupModel.getImpl() );
    pickup.getDelayVar(_length).setBounds(0,0);
    pickup.getWaitVar(_length).setBounds(0,0);

    IloVisit delivery(deliveryNode, deliveryVisitName);
    deliveryModel.add(delivery.getDelayVar(_time) == dropTime);
    deliveryModel.add(delivery.getTransitVar(_weight) == -quantity);
    deliveryModel.add(deliveryMinTime <= delivery.getCumulVar(_time) <=
deliveryMaxTime);
    deliveryModel.add(delivery);
    delivery.getDelayVar(_length).setBounds(0,0);
    delivery.getWaitVar(_length).setBounds(0,0);

    //add pickup and delivery order constraint
    pickupModel.add( IloOrderedVisitPair(_env, pickup, delivery) );
    delivery.setObject( (IloAny)deliveryModel.getImpl() );
    _mdl.add( pickupModel );
    _mdl.add( deliveryModel );
    ++it;
  }
  orderReader.end();
}

class RoutingSolver {
  RoutingModel&       _routing;
  IloModel           _mdl;
  IloSolver          _solver;
  IloDispatcher      _dispatcher;
  IloRoutingSolution  _solution;
  IloGoal             _instantiateCost;
  IloGoal             _restoreSolution;
  IloGoal             _fsGoal;
  IloNHood            _nhood;
  IloMetaHeuristic    _greedy;
  IloGoal             _move;

public:
  RoutingSolver(RoutingModel& routingModel);
  ~RoutingSolver() {}

  IloEnv getEnv() const { return _mdl.getEnv();}
  void syncSolution(IloModel mdl, IloSolution s, IloGoal g);
  void syncSolution();
  IloBool findFirstSolution();
  IloBool improveDepots();
  IloBool improvePlan();
  void printInformation() const;
};

RoutingSolver::RoutingSolver(RoutingModel& routingModel)
  : _routing( routingModel),
    _mdl(routingModel.getModel()),
    _solver(routingModel.getModel()),
    _dispatcher(_solver),
    _solution(routingModel.getModel())
{
  IloEnv env = getEnv();
  _instantiateCost =
    IloDichotomize(env, _dispatcher.getCostVar(), IloFalse);
  _restoreSolution = IloRestoreSolution(env, _solution);
  _fsGoal = IloNearestAdditionGenerate(env);

  _nhood = IloRelocate(env) + IloExchange(env);
  _greedy = IloImprove(env);

}
void RoutingSolver::printInformation() const {
  _solver.printInformation();
  _dispatcher.printInformation();
  _solver.out() << "===============" << endl
    << "Cost         : " << _dispatcher.getTotalCost() << endl
    << "Number of vehicles used : "
    << _dispatcher.getNumberOfVehiclesUsed() << endl
    << "Solution     : " << endl
    << _dispatcher << endl;
}

// Solving : find first solution
IloBool RoutingSolver::findFirstSolution() {
  if (!_solver.solve(_fsGoal && _instantiateCost)) {
    _solver.error() << "Infeasible Routing Plan" << endl;
    return IloFalse;
  }
  _solution.store(_solver);
  _solver.out() << "First solution with cost: "
                << _solution.getObjectiveValue()
```

```
              << endl;
  return IloTrue;
}

IloBool RoutingSolver::improveDepots() {
  IloEnv env = getEnv();
  _move =
    IloSingleMove(env, _solution, _nhood, _greedy, _instantiateCost);

  _solver.out() << endl << “Improvement loop” << endl;
  _solver.out() << “================” << endl;
  IloBool improved = IloTrue;
  IloInt nbOfDepots = _routing.getNumberOfDepots();
  improved = IloFalse;
  for (IloInt d = 0; d < nbOfDepots; ++d) {
    Depot* depot = _routing.getDepot(d);
    depot->fillModel(_solution);
    syncSolution(depot->getModel(), _solution, _instantiateCost);
    if ( depot->improve(_solver, _solution, _instantiateCost) ) {
      improved = IloTrue;
    }
  }

  // sync back to full model.
  syncSolution(_mdl, _solution, _instantiateCost);
  return improved;
}

IloBool RoutingSolver::improvePlan() {
  _solver.out() << “Optimizing plan “
              << _solution.getSolution().getObjectiveValue()
              << flush;
  _nhood.reset();
  _greedy.reset();
  IloInt nbImproved = 0;
  while ( _solver.solve(_move) ) { ++nbImproved;}
  _solver.out() << “ ---> “
             << _solution.getSolution().getObjectiveValue() << endl;
  return ( nbImproved > 0 );
}


void RoutingSolver::syncSolution(IloModel mdl, IloSolution s, IloGoal g) {
  _solver.extract(mdl);
  if ( !_solver.solve( IloRestoreSolution( _solver.getEnv(), s) && g))
    _solver.out() << “Synchronization failed” << endl;
  else
    s.store(_solver);
}

void RoutingSolver::syncSolution() {
  syncSolution(_mdl, _solution, _instantiateCost);
}

int main(int argc, char* argv[]) {
  IloEnv env;
  try {
    RoutingModel routingModel(env);
    routingModel.parse(argc, argv);
    routingModel.createModel();

    RoutingSolver rsolver(routingModel);
    if ( rsolver.findFirstSolution() ) {
      IloBool improved = IloTrue;
      do {
        improved =
          rsolver.improveDepots() && rsolver.improvePlan();
      } while ( improved );
    }
    rsolver.syncSolution();
    rsolver.printInformation();
  } catch (IloException& ex) {
    env.out() << “Error: “ << ex << endl;
  }
  env.end();
  return 0;
}
```

Complete Output

```
/**
 * Complete program output:
First solution with cost: 1397.52

Improvement loop
================
  Optimizing depot Depot 0 509.991 ---> 411.418
  Optimizing depot Depot 1 385.144 ---> 351.056
  Optimizing depot Depot 2 502.382 ---> 345.504
Optimizing plan 1107.98 ---> 1073.71

Improvement loop
================
  Optimizing depot Depot 0 412.393 ---> 391.898
  Optimizing depot Depot 1 325.838 ---> 323.948
  Optimizing depot Depot 2 335.482 ---> 310.54
Optimizing plan 1026.39 ---> 992.007

Improvement loop
================
  Optimizing depot Depot 0 385.09 ---> 378.142
  Optimizing depot Depot 1 323.948 ---> 323.948
  Optimizing depot Depot 2 282.968 ---> 281.295
  Optimizing plan 983.385 ---> 983.385
Number of fails              : 0
Number of choice points      : 0
Number of variables          : 7354
Number of constraints        : 1280
Reversible stack (bytes)     : 944724
Solver heap (bytes)          : 2509244
Solver global heap (bytes)   : 2446204
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
```

```
Constraint queue (bytes)      : 20176
Total memory used (bytes)     : 5988760
Elapsed time since creation   : 0.461
Number of nodes               : 103
Number of visits              : 260
Number of vehicles            : 30
Number of dimensions          : 3
Number of accepted moves      : 141
===============
Cost          : 983.385
Number of vehicles used : 15
Solution      :
Unperformed visits : None
Vehicle 0 of Depot Depot 0 : Unused
Vehicle 1 of Depot Depot 0 :
 -> Truck 1 leaving Depot 0 weight[0..200] time[0..4.62966] distance[0] ->
Pvisit21 weight[0] time[18.0278..22.6574] distance[18.0278] -> visit21
weight[11] time[18.0278..22.6574] distance[18.0278] -> Pvisit72 weight[0]
time[32.4999..37.1296] distance[22.4999] -> visit72 weight[25]
time[32.4999..37.1296] distance[22.4999] -> Pvisit75 weight[0]
time[47.8851..52.5147] distance[27.8851..27.8851] -> visit75 weight[18]
time[47.8851..52.5147] distance[27.8851..27.8851] -> Pvisit23 weight[0]
time[66.3703..71] distance[36.3703..36.3703] -> visit23 weight[29] time[68..71]
distance[36.3703..36.3703] -> Pvisit67 weight[0] time[90..93]
distance[48.3703..48.3703] -> visit67 weight[25] time[90..93]
distance[48.3703..48.3703] -> Pvisit39 weight[0] time[109.899..123.401]
distance[58.2698..58.2698] -> visit39 weight[31] time[109.899..123.401]
distance[58.2698..58.2698] -> Pvisit56 weight[0] time[126.899..140.401]
distance[65.2698..65.2698] -> visit56 weight[6] time[126.899..140.401]
distance[65.2698..65.2698] -> Pvisit74 weight[0] time[143.971..157.472]
distance[72.3409..72.3409] -> visit74 weight[8] time[149..157.472]
distance[72.3409..72.3409] -> Pvisit73 weight[0] time[163.472..171.945]
distance[76.813..76.813] -> visit73 weight[9] time[163.472..171.945]
distance[76.813..76.813] -> Pvisit2 weight[0] time[182.472..190.945]
distance[85.813..85.813] -> visit2 weight[7] time[182.472..190.945]
distance[85.813..85.813] -> Pvisit58 weight[0] time[201.528..210]
distance[94.8684..94.8684] -> visit58 weight[18] time[201.528..210]
distance[94.8684..94.8684] -> Truck 1 returning to Depot 0 weight[0]
time[220.583..230] distance[103.924..103.924]
Vehicle 2 of Depot Depot 0 :
 -> Truck 2 leaving Depot 0 weight[0..200] time[0..68.1115] distance[0] ->
Pvisit89 weight[0] time[9..77.1115] distance[9] -> visit89 weight[15]
time[9..77.1115] distance[9] -> Pvisit8 weight[0] time[36.8885..105]
distance[26.8885] -> visit8 weight[9] time[95..105] distance[26.8885] ->
Pvisit18 weight[0] time[115.44..204] distance[37.3289..37.3289] -> visit18
weight[12] time[115.44..204] distance[37.3289..37.3289] -> Truck 2 returning to
Depot 0 weight[0] time[141.252..230] distance[53.1402..53.1402]
Vehicle 3 of Depot Depot 0 :
 -> Truck 3 leaving Depot 0 weight[0..200] time[0..37.7199] distance[0] ->
Pvisit27 weight[0] time[5..42.7199] distance[5] -> visit27 weight[16]
time[5..42.7199] distance[5] -> Pvisit69 weight[0] time[22.2801..60]
distance[12.2801..12.2801] -> visit69 weight[6] time[50..60]
distance[12.2801..12.2801] -> Pvisit31 weight[0] time[67.8102..185]
distance[20.0904..20.0904] -> visit31 weight[27] time[67.8102..185]
distance[20.0904..20.0904] -> Pvisit88 weight[0] time[82.8102..200]
distance[25.0904..25.0904] -> visit88 weight[9] time[82.8102..200]
distance[25.0904..25.0904] -> Truck 3 returning to Depot 0 weight[0]
time[112.046..230] distance[44.3257..44.3257]
```

```
Vehicle 4 of Depot Depot 0 :
 -> Truck 4 leaving Depot 0 weight[0..200] time[0..8.78027] distance[0] ->
Pvisit52 weight[0] time[11.3137..20.094] distance[11.3137] -> visit52 weight[9]
time[11.3137..20.094] distance[11.3137] -> Pvisit7 weight[0]
time[31.2132..39.9935] distance[21.2132] -> visit7 weight[5]
time[31.2132..39.9935] distance[21.2132] -> Pvisit62 weight[0]
time[50.1575..58.9377] distance[30.1575] -> visit62 weight[19]
time[58..58.9377] distance[30.1575] -> Pvisit11 weight[0] time[76.0623..77]
distance[38.2197] -> visit11 weight[12] time[76.0623..77] distance[38.2197] ->
Pvisit19 weight[0] time[93.1333..96.0544] distance[45.2908] -> visit19
weight[17] time[93.1333..96.0544] distance[45.2908] -> Pvisit47 weight[0]
time[111.196..114.117] distance[53.3531] -> visit47 weight[27]
time[111.196..114.117] distance[53.3531] -> Pvisit36 weight[0]
time[128.407..131.328] distance[60.5642..60.5642] -> visit36 weight[5]
time[128.407..131.328] distance[60.5642..60.5642] -> Pvisit49 weight[0]
time[147.351..150.272] distance[69.5084..69.5084] -> visit49 weight[30]
time[147.351..150.272] distance[69.5084..69.5084] -> Pvisit64 weight[0]
time[170.079..173] distance[82.2364..82.2364] -> visit64 weight[9]
time[170.079..173] distance[82.2364..82.2364] -> Truck 4 returning to Depot 0
weight[0] time[226.598..230] distance[128.755..128.755]
Vehicle 5 of Depot Depot 0 : Unused
Vehicle 6 of Depot Depot 0 :
 -> Truck 6 leaving Depot 0 weight[0..200] time[0..92.1834] distance[0] ->
Pvisit53 weight[0] time[4.47214..96.6556] distance[4.47214] -> visit53
weight[14] time[95..96.6556] distance[4.47214] -> Pvisit28 weight[0]
time[112.211..113.867] distance[11.6832..11.6832] -> visit28 weight[16]
time[112.211..113.867] distance[11.6832..11.6832] -> Pvisit26 weight[0]
time[130.273..131.929] distance[19.7455..19.7455] -> visit26 weight[17]
time[130.273..131.929] distance[19.7455..19.7455] -> Pvisit40 weight[0]
time[147.344..149] distance[26.8166..26.8166] -> visit40 weight[9]
time[147.344..149] distance[26.8166..26.8166] -> Pvisit13 weight[0]
time[167.344..169] distance[36.8166..36.8166] -> visit13 weight[23]
time[167.344..169] distance[36.8166..36.8166] -> Truck 6 returning to Depot 0
weight[0] time[188.525..230] distance[47.9969..47.9969]
Vehicle 7 of Depot Depot 0 : Unused
Vehicle 8 of Depot Depot 0 : Unused
Vehicle 9 of Depot Depot 0 : Unused
Vehicle 0 of Depot Depot 1 :
 -> Truck 0 leaving Depot 1 weight[0..200] time[0..77.1716] distance[0] ->
Pvisit33 weight[0] time[1..78.1716] distance[1] -> visit33 weight[11]
time[1..78.1716] distance[1] -> Pvisit81 weight[0] time[13.8284..91]
distance[3.82843] -> visit81 weight[26] time[13.8284..91] distance[3.82843] ->
Pvisit9 weight[0] time[29.8284..107] distance[9.82843..9.82843] -> visit9
weight[16] time[97..107] distance[9.82843..9.82843] -> Pvisit51 weight[0]
time[113.325..147.821] distance[16.153..16.153] -> visit51 weight[10]
time[113.325..147.821] distance[16.153..16.153] -> Pvisit20 weight[0]
time[131.387..165.884] distance[24.2152..24.2152] -> visit20 weight[9]
time[131.387..165.884] distance[24.2152..24.2152] -> Pvisit30 weight[0]
time[148.458..182.955] distance[31.2863..31.2863] -> visit30 weight[21]
time[148.458..182.955] distance[31.2863..31.2863] -> Pvisit1 weight[0]
time[169.503..204] distance[42.3317..42.3317] -> visit1 weight[10]
time[169.503..204] distance[42.3317..42.3317] -> Truck 0 returning to Depot 1
weight[0] time[191.669..230] distance[54.4972..54.4972]
Vehicle 1 of Depot Depot 1 :
 -> Truck 1 leaving Depot 1 weight[0..200] time[0..97] distance[0] -> Pvisit79
weight[0] time[5..102] distance[5] -> visit79 weight[23] time[92..102]
distance[5] -> Pvisit3 weight[0] time[105.606..176.172]
distance[8.60555..8.60555] -> visit3 weight[13] time[105.606..176.172]
```

```
distance[8.60555..8.60555] -> Pvisit77 weight[0] time[118.434..189]
distance[11.434..11.434] -> visit77 weight[14] time[179..189]
distance[11.434..11.434] -> Truck 1 returning to Depot 1 weight[0]
time[197..230] distance[19.434..19.434]
Vehicle 2 of Depot Depot 1 :
 -> Truck 2 leaving Depot 1 weight[0..200] time[0..60.4037] distance[0] ->
Pvisit50 weight[0] time[7.2111..67.6148] distance[7.2111] -> visit50 weight[13]
time[7.2111..67.6148] distance[7.2111] -> Pvisit76 weight[0] time[22.5963..83]
distance[12.5963] -> visit76 weight[13] time[73..83] distance[12.5963] ->
Pvisit12 weight[0] time[90.0711..133.159] distance[19.6673..19.6673] -> visit12
weight[19] time[90.0711..133.159] distance[19.6673..19.6673] -> Pvisit4
weight[0] time[115.882..158.971] distance[35.4787..35.4787] -> visit4
weight[19] time[149..158.971] distance[35.4787..35.4787] -> Pvisit80 weight[0]
time[176.029..186] distance[52.5081..52.5081] -> visit80 weight[6]
time[176.029..186] distance[52.5081..52.5081] -> Pvisit68 weight[0]
time[188.029..198] distance[54.5081..54.5081] -> visit68 weight[36]
time[188.029..198] distance[54.5081..54.5081] -> Truck 2 returning to Depot 1
weight[0] time[210.399..230] distance[66.8774..66.8774]
Vehicle 3 of Depot Depot 1 :
 -> Truck 3 leaving Depot 1 weight[0..200] time[0..3.08082] distance[0] ->
Pvisit78 weight[0] time[8.06226..11.1431] distance[8.06226] -> visit78
weight[3] time[8.06226..11.1431] distance[8.06226] -> Pvisit34 weight[0]
time[23.0623..26.1431] distance[13.0623] -> visit34 weight[14]
time[23.0623..26.1431] distance[13.0623] -> Pvisit35 weight[0]
time[43.2603..46.3411] distance[23.2603] -> visit35 weight[8]
time[43.2603..46.3411] distance[23.2603] -> Pvisit71 weight[0]
time[59.9685..63.0493] distance[29.9685] -> visit71 weight[15]
time[59.9685..63.0493] distance[29.9685] -> Pvisit65 weight[0]
time[80.2641..83.3449] distance[40.2641] -> visit65 weight[20]
time[80.2641..83.3449] distance[40.2641] -> Pvisit66 weight[0]
time[103.866..106.946] distance[53.8656..53.8656] -> visit66 weight[25]
time[103.866..106.946] distance[53.8656..53.8656] -> Pvisit32 weight[0]
time[128.426..131.507] distance[68.4258..68.4258] -> visit32 weight[23]
time[128.426..131.507] distance[68.4258..68.4258] -> Pvisit90 weight[0]
time[142.898..145.979] distance[72.898..72.898] -> visit90 weight[3]
time[142.898..145.979] distance[72.898..72.898] -> Pvisit63 weight[0]
time[157.37..160.451] distance[77.3701..77.3701] -> visit63 weight[10]
time[157.37..160.451] distance[77.3701..77.3701] -> Pvisit10 weight[0]
time[176.857..179.938] distance[86.8569..86.8569] -> visit10 weight[16]
time[176.857..179.938] distance[86.8569..86.8569] -> Pvisit70 weight[0]
time[194.919..198] distance[94.9192..94.9192] -> visit70 weight[5]
time[194.919..198] distance[94.9192..94.9192] -> Truck 3 returning to Depot 1
weight[0] time[221.682..230] distance[111.682..111.682]
Vehicle 4 of Depot Depot 1 :
 -> Truck 4 leaving Depot 1 weight[0..200] time[0..92.5507] distance[0] ->
Pvisit29 weight[0] time[14.2127..106.763] distance[14.2127] -> visit29
weight[9] time[14.2127..106.763] distance[14.2127] -> Pvisit24 weight[0]
time[31.2837..123.834] distance[21.2837..21.2837] -> visit24 weight[3]
time[31.2837..123.834] distance[21.2837..21.2837] -> Pvisit55 weight[0]
time[53.4493..146] distance[33.4493..33.4493] -> visit55 weight[2]
time[136..146] distance[33.4493..33.4493] -> Pvisit25 weight[0]
time[149.606..174.958] distance[37.0548..37.0548] -> visit25 weight[6]
time[172..174.958] distance[37.0548..37.0548] -> Pvisit54 weight[0]
time[194.042..197] distance[49.0964..49.0964] -> visit54 weight[18]
time[194.042..197] distance[49.0964..49.0964] -> Truck 4 returning to Depot 1
weight[0] time[226.402..230] distance[71.4571..71.4571]
Vehicle 5 of Depot Depot 1 : Unused
Vehicle 6 of Depot Depot 1 : Unused
Vehicle 7 of Depot Depot 1 : Unused
Vehicle 8 of Depot Depot 1 : Unused
Vehicle 9 of Depot Depot 1 : Unused
Vehicle 0 of Depot Depot 2 : Unused
Vehicle 1 of Depot Depot 2 :
 -> Truck 1 leaving Depot 2 weight[0..200] time[0..71.4822] distance[0] ->
Pvisit98 weight[0] time[4.12311..75.6054] distance[4.12311] -> visit98
weight[10] time[4.12311..75.6054] distance[4.12311] -> Pvisit99 weight[0]
time[19.2221..90.7044] distance[9.22213..9.22213] -> visit99 weight[9]
time[83..90.7044] distance[9.22213..9.22213] -> Pvisit84 weight[0]
time[103.296..111] distance[19.5178..19.5178] -> visit84 weight[7]
time[103.296..111] distance[19.5178..19.5178] -> Pvisit5 weight[0]
time[117.419..162.292] distance[23.6409..23.6409] -> visit5 weight[26]
time[117.419..162.292] distance[23.6409..23.6409] -> Pvisit93 weight[0]
time[134.127..179] distance[30.3491..30.3491] -> visit93 weight[22]
time[134.127..179] distance[30.3491..30.3491] -> Pvisit100 weight[0]
time[150.127..195] distance[36.3491..36.3491] -> visit100 weight[17]
time[185..195] distance[36.3491..36.3491] -> Truck 1 returning to Depot 2
weight[0] time[196..230] distance[37.3491..37.3491]
Vehicle 2 of Depot Depot 2 :
 -> Truck 2 leaving Depot 2 weight[0..200] time[0..11.566] distance[0] ->
Pvisit37 weight[0] time[3.60555..15.1716] distance[3.60555] -> visit37
weight[8] time[3.60555..15.1716] distance[3.60555] -> Pvisit92 weight[0]
time[16.434..28] distance[6.43398] -> visit92 weight[2] time[18..28]
distance[6.43398] -> Pvisit94 weight[0] time[34.4031..62.5951]
distance[12.8371] -> visit94 weight[27] time[34.4031..62.5951]
distance[12.8371] -> Pvisit95 weight[0] time[47.5654..75.7574]
distance[15.9994..15.9994] -> visit95 weight[20] time[47.5654..75.7574]
distance[15.9994..15.9994] -> Pvisit97 weight[0] time[60.5654..88.7574]
distance[18.9994..18.9994] -> visit97 weight[12] time[60.5654..88.7574]
distance[18.9994..18.9994] -> Pvisit87 weight[0] time[74.808..103]
distance[23.242..23.242] -> visit87 weight[26] time[93..103]
distance[23.242..23.242] -> Truck 2 returning to Depot 2 weight[0]
time[113.05..230] distance[33.2919..33.2919]
Vehicle 3 of Depot Depot 2 :
 -> Truck 3 leaving Depot 2 weight[0..200] time[0..6.359] distance[0] ->
Pvisit85 weight[0] time[5.38516..11.7442] distance[5.38516] -> visit85
weight[41] time[5.38516..11.7442] distance[5.38516] -> Pvisit61 weight[0]
time[19.8573..26.2163] distance[9.8573..9.8573] -> visit61 weight[13]
time[19.8573..26.2163] distance[9.8573..9.8573] -> Pvisit17 weight[0]
time[39.0768..45.4358] distance[19.0768..19.0768] -> visit17 weight[2]
time[39.0768..45.4358] distance[19.0768..19.0768] -> Pvisit45 weight[0]
time[57.1391..63.4981] distance[27.1391..27.1391] -> visit45 weight[16]
time[57.1391..63.4981] distance[27.1391..27.1391] -> Pvisit46 weight[0]
time[77.9094..84.2684] distance[37.9094..37.9094] -> visit46 weight[1]
time[77.9094..84.2684] distance[37.9094..37.9094] -> Pvisit48 weight[0]
time[99.6141..105.973] distance[49.6141..49.6141] -> visit48 weight[36]
time[99.6141..105.973] distance[49.6141..49.6141] -> Pvisit82 weight[0]
time[114.999..121.358] distance[54.9993..54.9993] -> visit82 weight[16]
time[114.999..121.358] distance[54.9993..54.9993] -> Pvisit83 weight[0]
time[135.049..141.408] distance[65.0492..65.0492] -> visit83 weight[11]
time[135.049..141.408] distance[65.0492..65.0492] -> Pvisit60 weight[0]
time[149.292..155.651] distance[69.2918..69.2918] -> visit60 weight[3]
time[149.292..155.651] distance[69.2918..69.2918] -> Pvisit6 weight[0]
time[168.236..174.595] distance[78.2361..78.2361] -> visit6 weight[3]
time[168.236..174.595] distance[78.2361..78.2361] -> Pvisit96 weight[0]
time[182.479..188.838] distance[82.4787..82.4787] -> visit96 weight[11]
time[182.479..188.838] distance[82.4787..82.4787] -> Pvisit59 weight[0]
```

```
time[195.641..202] distance[85.641..85.641] -> visit59 weight[28]
time[195.641..202] distance[85.641..85.641] -> Truck 3 returning to Depot 2
weight[0] time[213.257..230] distance[93.2568..93.2568]
Vehicle 4 of Depot Depot 2 :
 -> Truck 4 leaving Depot 2 weight[0..200] time[0..64.5676] distance[0] ->
Pvisit44 weight[0] time[7.61577..72.1833] distance[7.61577] -> visit44
weight[18] time[69..72.1833] distance[7.61577] -> Pvisit38 weight[0]
time[89.8167..93] distance[18.4324..18.4324] -> visit38 weight[16]
time[89.8167..93] distance[18.4324..18.4324] -> Pvisit86 weight[0]
time[112.855..162.576] distance[31.4708..31.4708] -> visit86 weight[35]
time[112.855..162.576] distance[31.4708..31.4708] -> Pvisit16 weight[0]
time[129.18..178.901] distance[37.7954..37.7954] -> visit16 weight[19]
time[129.18..178.901] distance[37.7954..37.7954] -> Pvisit91 weight[0]
time[144.279..194] distance[42.8944..42.8944] -> visit91 weight[1]
time[144.279..194] distance[42.8944..42.8944] -> Truck 4 returning to Depot 2
weight[0] time[157.884..230] distance[46.5..46.5]
Vehicle 5 of Depot Depot 2 :
 -> Truck 5 leaving Depot 2 weight[0..200] time[0..25.474] distance[0] ->
Pvisit14 weight[0] time[7.61577..33.0897] distance[7.61577] -> visit14
weight[20] time[7.61577..33.0897] distance[7.61577] -> Pvisit43 weight[0]
time[28.2459..53.7199] distance[18.2459] -> visit43 weight[7]
time[28.2459..53.7199] distance[18.2459] -> Pvisit15 weight[0] time[45.526..71]
distance[25.526..25.526] -> visit15 weight[8] time[61..71]
distance[25.526..25.526] -> Pvisit41 weight[0] time[83.1655..138.604]
distance[37.6916..37.6916] -> visit41 weight[5] time[83.1655..138.604]
distance[37.6916..37.6916] -> Pvisit22 weight[0] time[97.4082..152.847]
distance[41.9342..41.9342] -> visit22 weight[18] time[97.4082..152.847]
distance[41.9342..41.9342] -> Pvisit57 weight[0] time[120.561..176]
distance[55.0871..55.0871] -> visit57 weight[7] time[120.561..176]
distance[55.0871..55.0871] -> Pvisit42 weight[0] time[138.561..194]
distance[63.0871..63.0871] -> visit42 weight[5] time[138.561..194]
distance[63.0871..63.0871] -> Truck 5 returning to Depot 2 weight[0]
time[156.371..230] distance[70.8974..70.8974]
Vehicle 6 of Depot Depot 2 : Unused
Vehicle 7 of Depot Depot 2 : Unused
Vehicle 8 of Depot Depot 2 : Unused
Vehicle 9 of Depot Depot 2 : Unused

*/
```

C H A P T E R

11

Modeling Complex Costs

In this lesson, you will learn how to:

◆ model vehicle costs with variable *cost coefficients*

◆ use the member function `setCost` to apply cost coefficient functions

◆ use the function `IloNumToNumSegmentFunction`

In this lesson, you model problems which apply complex costs via variable cost coefficient functions. In previous lessons, costs were typically based on linear combinations of the route distance, time, or vehicle load. This lesson introduces variable cost coefficients, which are values that can be applied to route dimensions in order to determine the route cost. These variable coefficient functions allow you to model vehicle costs with greater flexibility, and thereby extend your ability to apply a wide variety of cost calculations to your routing problems.

The first example in this lesson, `cost1.cpp`, models a last-visit-dependent cost, and it starts with “Describe cost1” on page 280. The second example, `cost2.cpp`, applies higher transit costs as route distance thresholds are passed, and that lesson begins with “Describe cost2” on page 282.

Describe cost1

Routing problem costs may be highly dependent on the location of specific customer visits. This problem examines the case where the location of the last customer visit determines the cost coefficient of the entire route. This type of problem could represent a business need to have the last customer visit as close to the depot as possible.

Step 1 Describe the problem

The first step is to write a natural language description of the problem. The constraints and objectives for `cost1.cpp` are the same as for the PDP; in particular, the objective is to minimize the costs of the delivery of all the parcels. The difference in this lesson lies in the way the costs are calculated with cost coefficients. In `cost1.cpp`, vehicle costs are derived from a calculation based on the x-coordinate of the vehicle's last visit before returning to the depot.

Model cost1

Once you have written a description of your problem, you can use Dispatcher classes to model it.

Step 2 Open the example file

Open the example file `YourDispatcherHome/examples/src/tutorial/cost1_partial.cpp` in your development environment.

This problem is modeled as for a PDP, with only the creation and addition of a cost coefficient function.

Step 3 Create the LastVisitCostCoef function

Add the following code after the comment
`//Create the LastVisitCostCoef function`.

```
IloNum LastVisitCostCoef(IloVisit visit) {
  IloNode node = visit.getNode();
  return ((IloInt)(node.getX()) % 5) + 1;
}
```

The `LastVisitCostCoef` is a function of the type `IloSimpleVisitToNumFunction`, and it creates the cost coefficient from the x-coordinate of `visit`.

Next, you set the vehicle cost in the `createVehicles` function.

Step 4 Add the cost function

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

```
IloVisitToNumFunction coefFunction(_env, LastVisitCostCoef);
vehicle.setCost(distance,
                coefFunction,
                vehicle.getLastVisit().getPrevVar());
```

These lines in `createVehicles` create the vehicle cost. `coefFunction` is created as an `IloVisitToNumFunction` which allows you to associate `IloNum` values with visits.

The member function `IloVehicle::setCost` is used here to associate a visit-dependent proportional cost with `vehicle`. The parameter `vehicle.getLastVisit().getPrevVar()` returns the location variable with which `coefFunction` calculates the cost applied to `distance`. Note that the visit returned by `vehicle.getLastVisit().getPrevVar()` corresponds to the last customer visit performed before returning to the depot.

Solve cost1

The solution for `cost1.cpp` is computed, improved, and displayed by methods previously presented for the PDP problem.

Step 5 Compile and run the cost1 program

The solution improvement phase finds a solution using 10 vehicles with a cost of 1203.79 units:

```
  2286.23
Improving solution
Number of fails              : 0
Number of choice points      : 0
Number of variables          : 2344
Number of constraints        : 440
Reversible stack (bytes)     : 201024
Solver heap (bytes)          : 1017340
Solver global heap (bytes)   : 144384
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11160
Total memory used (bytes)    : 1442320
Elapsed time since creation  : 0.02
Number of nodes              : 51
Number of visits             : 80
Number of vehicles           : 15
Number of dimensions         : 3
Number of accepted moves     : 24
===============
Cost         : 1203.79
Number of vehicles used : 10
```

The complete `cost1.cpp` program and output is presented in "Complete cost1 Program" on page 286.

Describe cost2

There are numerous reasons why you may want to put a "penalty" cost on high mileage in a route solution. For example, there may be a desire to simply limit the amount of driving time, or a desire to keep all the vehicles in a solution near the same mileage. One way to achieve this is to apply higher costs when the vehicle mileage for any particular route passes certain thresholds. In `cost2.cpp`, vehicle costs are calculated from the length of the trip; longer trips result in higher costs.

Step 1 Describe the problem

The first step is to write a natural language description of the problem. The constraints and objectives for `cost2.cpp` are the same as for the PDP; in particular, the objective is to minimize the costs of the delivery of all the parcels. The difference in this lesson lies in the way the costs are calculated. In cost2.cpp, the cost is determined from the total length of the vehicle route, which then determines the value returned by the segmented function.

Model cost2

Once you have written a description of your problem, you can use Dispatcher classes to model it.

Step 2 Open the example file

Open the example file YourDispatcherHome/examples/src/tutorial/cost2_partial.cpp in your development environment.

In this example, the cost function is added as an IloNumToNumSegmentFunction in the createVehicles function.

Step 3 Add the vehicle cost function

Add the following code after the comment //Add the vehicle cost function.

```
vehicle.setCost(distance, 1);
IloNumToNumSegmentFunction costFunction(_env);
costFunction.setSlope(75, 100, 37.5, 0.5);
costFunction.setSlope(100, IloInfinity, 100, 1);
vehicle.setCost(distance, costFunction);
```

First, a standard proportional cost is applied on the distance traveled (cost per unit distance is 1). Then, costFunction is created with a default value of 0. The member function setSlope is then used to change the value of the function, depending on the range given.

Solve cost2

The solution for cost2.cpp is computed, improved, and displayed by methods previously presented for the PDP problem.

Step 4 Compile and run the cost2 program

The solution improvement phase finds a solution using 13 vehicles with a cost of 1900.16 units:

```
  2583.49
Improving solution
Number of fails              : 0
Number of choice points      : 0
Number of variables          : 2359
Number of constraints        : 440
Reversible stack (bytes)     : 201024
Solver heap (bytes)          : 1025380
Solver global heap (bytes)   : 269064
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11160
Total memory used (bytes)    : 1575040
Elapsed time since creation  : 0.01
Number of nodes              : 51
Number of visits             : 80
Number of vehicles           : 15
Number of dimensions         : 3
Number of accepted moves     : 28
===============
Cost         : 1900.16
Number of vehicles used : 13
```

The complete `cost2.cpp` program and output is presented in "Complete cost2 Program" on page 293.

Review Exercises

For answers, see "Suggested Answers" on page 285.

1. What member function is used to associate a visit-dependent proportional cost with a vehicle?
2. What Dispatcher class is used to associate `IloNum` values with vehicle visits?
3. The following section of code is from the `createVehicles` function of `cost1.cpp`. If the code `.getPrevVar()` were removed, all the other code in `cost1.cpp` remained the same, and the depot were located at (0,0), what would be the resulting cost coefficient?

```
IloVisitToNumFunction coefFunction(_env, LastVisitCostCoef);
vehicle.setCost(distance,
                coefFunction,
                vehicle.getLastVisit().getPrevVar());
```

Suggested Answers

Exercise 1

What member function is used to associate a visit-dependent proportional cost with a vehicle?

Suggested Answer

`IloVehicle::setCost`.

Exercise 2

What Dispatcher class is used to associate `IloNum` values with vehicle visits?

Suggested Answer

`IloVisitToNumFunction.`

Exercise 3

The following section of code is from the `createVehicles` function of `cost1.cpp`. If the code `.getPrevVar()` were removed, all the other code in `cost1.cpp` remained the same, and the depot is located at (0,0), what would be the resulting cost coefficient?

```
    IloVisitToNumFunction coefFunction(_env, LastVisitCostCoef);
    vehicle.setCost(distance,
                    coefFunction,
                    vehicle.getLastVisit().getPrevVar());
```

Suggested Answer

Removing `.getPrevVar()` would mean that `coefFunction` uses the very last visit—the return to the depot visit—as the visit to use in calculating the cost coefficient. The coefficient is based on the x-coordinate of this visit; so if the depot were located at (0,0), then the cost would be based on a coefficient of 1. See the following code for the function `LastVisitCostCoef`.

```
IloNum LastVisitCostCoef(IloVisit visit) {
  IloNode node = visit.getNode();
  return ((IloInt)(node.getX()) % 5) + 1;
}
```

Complete cost1 Program

The complete program and output for `cost1.cpp` follows. You can also view it online in the file `YourDispatcherHome/examples/src/cost1.cpp`.

```
// -------------------------------------------------------------- -*- C++ -*-
// File: examples/src/cost1.cpp
// --------------------------------------------------------------------------


#include <ildispat/ilodispatcher.h>

ILOSTLBEGIN

///////////////////////////////////////////////////////////////////////////////
// Modeling
class RoutingModel {
  IloEnv              _env;
  IloModel            _mdl;

  void createDimensions();
  void createIloNodes(char* nodeFileName);
  void createVehicles(char* vehicleFileName);
  void createVisits(char* visitsFileName);
public:
  RoutingModel(IloEnv env,
               int argc,
               char* argv[]);
  ~RoutingModel() {}
  IloEnv getEnv() const { return _env; }
  IloModel getModel() const { return _mdl; }
};

RoutingModel::RoutingModel(IloEnv env,
                                       int argc,
                                       char * argv[])
  : _env(env), _mdl(env){

  createDimensions();

  char* nodeFileName;
  if(argc < 4)
    nodeFileName = (char *) "../../../examples/data/pdp/nodes.csv";
  else
    nodeFileName = argv[3];
  createIloNodes(nodeFileName);

  char* vehiclesFileName;
  if (argc < 2)
    vehiclesFileName = (char *) "../../../examples/data/pdp/vehicles.csv";
  else
    vehiclesFileName = argv[1];
  createVehicles(vehiclesFileName);

  char* visitsFileName;
  if (argc < 3)
```

```
    visitsFileName =
      (char*) “../../../examples/data/pdp/visits.csv”;
  else
    visitsFileName = argv[2];
  createVisits(visitsFileName);
}

// Create dimensions
void RoutingModel::createDimensions() {
  IloDimension1 weight(_env, “Weight”);
  weight.setKey(“Weight”);
  _mdl.add(weight);
  IloDimension2 time(_env, IloEuclidean, “Time”);
  time.setKey(“Time”);
  _mdl.add(time);
  IloDimension2 distance(_env, IloEuclidean, IloFalse, “Distance”);
  distance.setKey(“Distance”);
  _mdl.add(distance);
}

// Create IloNodes
void RoutingModel::createIloNodes(char* nodeFileName) {
  IloCsvReader csvNodeReader(_env, nodeFileName);
  IloCsvReader::LineIterator  it(csvNodeReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char* name = line.getStringByHeader(“name”);
    IloNode node(_env,
                 line.getFloatByHeader(“x”),
                 line.getFloatByHeader(“y”),
                 0,
                 name);
    node.setKey(name);
    ++it;
  }
  csvNodeReader.end();
}

IloNum LastVisitCostCoef(IloVisit visit) {
  IloNode node = visit.getNode();
  return ((IloInt)(node.getX()) % 5) + 1;
}

// Create vehicles
void RoutingModel::createVehicles(char* vehicleFileName) {
  IloDimension1 weight = IloDimension1::Find(_env, “Weight”);
  IloDimension2 time = IloDimension2::Find(_env, “Time”);
  IloDimension2 distance = IloDimension2::Find(_env, “Distance”);
  IloCsvReader csvVehicleReader(_env, vehicleFileName);
  IloCsvReader::LineIterator it(csvVehicleReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char * namefirst = line.getStringByHeader(“first”);
    char * namelast = line.getStringByHeader(“last”);
    char * name = line.getStringByHeader(“name”);
    IloNum capacity = line.getFloatByHeader(“capacity”);
    IloNum openTime = line.getFloatByHeader(“open”);
    IloNum closeTime = line.getFloatByHeader(“close”);
    IloNode node1 = IloNode::Find(_env, namefirst);
    IloNode node2 = IloNode::Find(_env, namelast);

    IloVisit first(node1, “depot”);
    _mdl.add(first.getCumulVar(weight) == 0);
    _mdl.add(first.getCumulVar(time) >= openTime);

    IloVisit last(node2, “depot”);
    _mdl.add(last.getCumulVar(time) <= closeTime);
    IloVehicle vehicle(first, last, name);

    IloVisitToNumFunction coefFunction(_env, LastVisitCostCoef);
    vehicle.setCost(distance,
                    coefFunction,
                    vehicle.getLastVisit().getPrevVar());

    vehicle.setCapacity(weight, capacity);
    _mdl.add(vehicle);
    ++it;
  }
  csvVehicleReader.end();
}

// Create visits
void RoutingModel::createVisits(char* visitsFileName) {
  IloDimension1 weight = IloDimension1::Find(_env, “Weight”);
  IloDimension2 time = IloDimension2::Find(_env, “Time”);
  IloCsvReader csvVisitReader(_env, visitsFileName);
  IloCsvReader::LineIterator it(csvVisitReader);
  while(it.ok()){
    IloCsvLine line = *it;
    //read visit data from files
    char * pickupVisitName = line.getStringByHeader(“pickup”);
    char * pickupNodeName = line.getStringByHeader(“pickupNode”);
    char * deliveryVisitName = line.getStringByHeader(“delivery”);
    char * deliveryNodeName = line.getStringByHeader(“deliveryNode”);
    IloNum quantity = line.getFloatByHeader(“quantity”);
    IloNum pickupMinTime  = line.getFloatByHeader(“pickupMinTime”);
    IloNum pickupMaxTime  = line.getFloatByHeader(“pickupMaxTime”);
    IloNum deliveryMinTime  = line.getFloatByHeader(“deliveryMinTime”);
    IloNum deliveryMaxTime  = line.getFloatByHeader(“deliveryMaxTime”);
    IloNum dropTime = line.getFloatByHeader(“dropTime”);
    IloNum pickupTime = line.getFloatByHeader(“pickupTime”);

    // read data nodes from the file nodes.csv
    // and create pickup and delivery nodes

    IloNode pickupNode = IloNode::Find(_env, pickupNodeName);
    IloNode deliveryNode = IloNode::Find(_env, deliveryNodeName);

    //create and add pickup and delivery visits
    IloVisit pickup(pickupNode, pickupVisitName);
    _mdl.add(pickup.getDelayVar(time) == pickupTime);
    _mdl.add(pickup.getTransitVar(weight) == quantity);
    _mdl.add(pickupMinTime <= pickup.getCumulVar(time) <= pickupMaxTime);
    _mdl.add(pickup);
    IloVisit delivery(deliveryNode, deliveryVisitName);
    _mdl.add(delivery.getDelayVar(time) == dropTime);
    _mdl.add(delivery.getTransitVar(weight) == -quantity);
    _mdl.add(deliveryMinTime <= delivery.getCumulVar(time) <= deliveryMaxTime);
    _mdl.add(delivery);
    //add pickup and delivery order constraint
    _mdl.add(IloOrderedVisitPair(_env, pickup, delivery));
    ++it;
  }
  csvVisitReader.end();
}

///////////////////////////////////////////////////////////////////////////////
// Solving
class RoutingSolver {
  IloModel           _mdl;
  IloSolver          _solver;
  IloRoutingSolution  _solution;

  IloBool findFirstSolution(IloGoal goal);
  void greedyImprove(IloNHood nhood, IloGoal subGoal);
  void improve(IloGoal subgoal);
public:
  RoutingSolver(RoutingModel mdl);
  ~RoutingSolver() {}
  IloRoutingSolution getSolution() const { return _solution; }
  void printInformation() const;
  void solve();
};

RoutingSolver::RoutingSolver(RoutingModel mdl)
  : _mdl(mdl.getModel()), _solver(mdl.getModel()), _solution(mdl.getModel()) {}

// Solving : find first solution
IloBool RoutingSolver::findFirstSolution(IloGoal goal) {
  if (!_solver.solve(goal)) {
    _solver.error() << “Infeasible Routing Plan” << endl;
    return IloFalse;
  }
  IloDispatcher dispatcher(_solver);
  _solver.out() << dispatcher.getTotalCost() << endl;
  _solution.store(_solver);
  return IloTrue;
}

// Improve solution using nhood
void RoutingSolver::greedyImprove(IloNHood nhood, IloGoal subGoal) {
  _solver.out() << “Improving solution” << endl;
  IloEnv env = _solver.getEnv();
  nhood.reset();
  IloGoal improve = IloSingleMove(env,
                                  _solution,
                                  nhood,
                                  IloImprove(env),
                                  subGoal);
  while (_solver.solve(improve)) {}
}

// Improve solution
void RoutingSolver::improve(IloGoal subGoal) {
  IloEnv env = _solver.getEnv();
  IloNHood nhood = IloTwoOpt(env)
                 + IloOrOpt(env)
                 + IloRelocate(env)
                 + IloCross(env)
                 + IloExchange(env);
  greedyImprove(nhood, subGoal);
}

// Display Dispatcher information
void RoutingSolver::printInformation() const {
  IloDispatcher dispatcher(_solver);
  _solver.printInformation();
  dispatcher.printInformation();
  _solver.out() << "===============" << endl
                << "Cost         : " << dispatcher.getTotalCost() << endl
                << "Number of vehicles used : "
                << dispatcher.getNumberOfVehiclesUsed() << endl
                << "Solution     : " << endl
                << dispatcher << endl;
}

// Solving
void RoutingSolver::solve() {
  IloDispatcher dispatcher(_solver);
  IloEnv env = _solver.getEnv();
  // Subgoal
  IloGoal instantiateCost = IloDichotomize(env,
                                          dispatcher.getCostVar(),
                                          IloFalse);
  IloGoal restoreSolution = IloRestoreSolution(env, _solution);
  IloGoal goal = IloSavingsGenerate(env) && instantiateCost;

  // Solving
  if (findFirstSolution(goal)) {
    improve(instantiateCost);
    _solver.solve(restoreSolution);
  }
}

///////////////////////////////////////////////////////////////////////////////

int main(int argc, char * argv[]) {
  IloEnv env;
  try {
    RoutingModel mdl(env, argc, argv);
    RoutingSolver solver(mdl);
    solver.solve();
    solver.printInformation();
  } catch(IloException& ex) {
    cerr << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Complete cost1 Output

```
/**
2286.23
Improving solution
Number of fails              : 0
Number of choice points      : 0
Number of variables          : 2344
Number of constraints        : 440
Reversible stack (bytes)     : 201024
Solver heap (bytes)          : 1017340
Solver global heap (bytes)   : 144384
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11160
Total memory used (bytes)    : 1442320
Elapsed time since creation  : 0.02
Number of nodes              : 51
Number of visits             : 80
Number of vehicles           : 15
Number of dimensions         : 3
Number of accepted moves     : 24
===============
Cost         : 1203.79
Number of vehicles used : 10
Solution     :
Unperformed visits : None
vehicle1 :
 -> depot Weight[0] Time[0..12.414] Distance[0..Inf) -> visit35 Weight[0..175]
Time[41.0366..53.4506] Distance[0..Inf) -> visit19 Weight[8..183]
Time[99.2963..111.71] Distance[0..Inf) -> visit36 Weight[25..200]
Time[122.296..134.71] Distance[0..Inf) -> visit20 Weight[17..192]
Time[175.586..188] Distance[0..Inf) -> depot Weight[0..175] Time[217.209..230]
Distance[0..Inf)
vehicle2 :
 -> depot Weight[0] Time[0..18.7919] Distance[0..Inf) -> visit21 Weight[0..140]
Time[18.0278..36.8197] Distance[0..Inf) -> visit22 Weight[11..151]
Time[38.0278..56.8197] Distance[0..Inf) -> visit23 Weight[0..140] Time[68..78]
Distance[0..Inf) -> visit39 Weight[29..169] Time[86.6023..137.537]
Distance[0..Inf) -> visit24 Weight[60..200] Time[120.14..171.074]
Distance[0..Inf) -> visit40 Weight[31..171] Time[157.065..208] Distance[0..Inf)
-> depot Weight[0..140] Time[178.246..230] Distance[0..Inf)
vehicle3 :
 -> depot Weight[0] Time[0..47.076] Distance[0..Inf) -> visit5 Weight[0..145]
Time[20.6155..67.6915] Distance[0..Inf) -> visit17 Weight[26..171]
Time[40.6155..87.6915] Distance[0..Inf) -> visit45 Weight[28..173]
Time[58.6778..105.754] Distance[0..Inf) -> visit46 Weight[44..189]
Time[79.4481..126.524] Distance[0..Inf) -> visit47 Weight[28..173]
Time[99.4481..146.524] Distance[0..Inf) -> visit48 Weight[55..200]
Time[115.851..162.927] Distance[0..Inf) -> visit18 Weight[28..173]
Time[139.744..186.82] Distance[0..Inf) -> visit6 Weight[26..171]
Time[160.924..208] Distance[0..Inf) -> depot Weight[0..145] Time[182.104..230]
Distance[0..Inf)
vehicle4 :
 -> depot Weight[0] Time[0..13.1563] Distance[0..Inf) -> visit33 Weight[0..175]
Time[24.7588..37.9152] Distance[0..Inf) -> visit29 Weight[11..186]
Time[49.6249..62.7813] Distance[0..Inf) -> visit34 Weight[20..195]
```

```
Time[72.6633..85.8197] Distance[0..Inf) -> visit9 Weight[9..184] Time[97..107]
Distance[0..Inf) -> visit30 Weight[25..200] Time[122..174] Distance[0..Inf) ->
visit10 Weight[16..191] Time[142..194] Distance[0..Inf) -> depot Weight[0..175]
Time[177.495..230] Distance[0..Inf)
vehicle5 :
 -> depot Weight[0] Time[0..22.3215] Distance[0..Inf) -> visit27 Weight[0..145]
Time[5..27.3215] Distance[0..Inf) -> visit31 Weight[16..161]
Time[27.6491..49.9706] Distance[0..Inf) -> visit11 Weight[43..188] Time[67..77]
Distance[0..Inf) -> visit32 Weight[55..200] Time[92.5242..143.223]
Distance[0..Inf) -> visit28 Weight[28..173] Time[135.082..185.78]
Distance[0..Inf) -> visit12 Weight[12..157] Time[154.301..205] Distance[0..Inf)
-> depot Weight[0..145] Time[179.301..230] Distance[0..Inf)
vehicle6 :
 -> depot Weight[0] Time[0..19.9727] Distance[0..Inf) -> visit41 Weight[0..164]
Time[28.8617..48.8345] Distance[0..Inf) -> visit15 Weight[5..169] Time[61..71]
Distance[0..Inf) -> visit13 Weight[13..177] Time[109..125.282] Distance[0..Inf)
-> visit42 Weight[36..200] Time[133.318..149.6] Distance[0..Inf) -> visit14
Weight[31..195] Time[152.537..168.82] Distance[0..Inf) -> visit16
Weight[8..172] Time[173.718..190] Distance[0..Inf) -> depot Weight[0..164]
Time[212.872..230] Distance[0..Inf)
vehicle7 :
 -> depot Weight[0] Time[0..18.5452] Distance[0..Inf) -> visit43 Weight[0..193]
Time[34.176..52.7212] Distance[0..Inf) -> visit44 Weight[7..200] Time[69..79]
Distance[0..Inf) -> visit25 Weight[0..193] Time[172..175.639] Distance[0..Inf)
-> visit26 Weight[6..199] Time[204.361..208] Distance[0..Inf) -> depot
Weight[0..193] Time[225.541..230] Distance[0..Inf)
vehicle8 :
 -> depot Weight[0] Time[0..0.750223] Distance[0..Inf) -> visit1 Weight[0..182]
Time[15.2315..15.9818] Distance[0..Inf) -> visit37 Weight[10..192]
Time[61.0366..61.7868] Distance[0..Inf) -> visit38 Weight[18..200]
Time[92.2498..93] Distance[0..Inf) -> visit2 Weight[10..192] Time[134.561..202]
Distance[0..Inf) -> depot Weight[0..182] Time[162.561..230] Distance[0..Inf)
vehicle9 :
 -> depot Weight[0] Time[0..61.5802] Distance[0..Inf) -> visit7 Weight[0..195]
Time[21.2132..82.7934] Distance[0..Inf) -> visit8 Weight[5..200] Time[95..105]
Distance[0..Inf) -> depot Weight[0..195] Time[131.249..230] Distance[0..Inf)
vehicle10 :
 -> depot Weight[0] Time[0..5.75685] Distance[0..Inf) -> visit49 Weight[0..170]
Time[43.9318..49.6886] Distance[0..Inf) -> visit50 Weight[30..200]
Time[99.9969..105.754] Distance[0..Inf) -> visit3 Weight[0..170]
Time[118.243..124] Distance[0..Inf) -> visit4 Weight[13..183]
Time[153.243..159] Distance[0..Inf) -> depot Weight[0..170] Time[188.243..230]
Distance[0..Inf)
vehicle11 : Unused
vehicle12 : Unused
vehicle13 : Unused
vehicle14 : Unused
vehicle15 : Unused

*/
```

Complete cost2 Program

The complete program and output for `cost2.cpp` follows. You can also view it online in the file `YourDispatcherHome/examples/src/cost2.cpp.`

```
// -------------------------------------------------------------- -*- C++ -*-
// File: examples/src/cost2.cpp
// --------------------------------------------------------------------------


#include <ildispat/ilodispatcher.h>

ILOSTLBEGIN
////////////////////////////////////////////////////////////////////////////////
// Modeling
class RoutingModel {
  IloEnv             _env;
  IloModel           _mdl;

  void createDimensions();
  void createIloNodes(char* nodeFileName);
  void createVehicles(char* vehicleFileName);
  void createVisits(char* visitsFileName);
public:
  RoutingModel(IloEnv env,
               int argc,
               char* argv[]);
  ~RoutingModel() {}
  IloEnv getEnv() const { return _env; }
  IloModel getModel() const { return _mdl; }
};

RoutingModel::RoutingModel(IloEnv env,
                           int argc,
                           char * argv[])
  : _env(env), _mdl(env){

  createDimensions();

  char* nodeFileName;
  if(argc < 4)
    nodeFileName = (char *) “../../../examples/data/pdp/nodes.csv”;
  else
    nodeFileName = argv[3];
  createIloNodes(nodeFileName);

  char* vehiclesFileName;
  if (argc < 2)
    vehiclesFileName = (char *) “../../../examples/data/pdp/vehicles.csv”;
  else
    vehiclesFileName = argv[1];
  createVehicles(vehiclesFileName);

  char* visitsFileName;
  if (argc < 3)
    visitsFileName =
      (char*) “../../../examples/data/pdp/visits.csv”;
  else
    visitsFileName = argv[2];
  createVisits(visitsFileName);
}

// Create dimensions
void RoutingModel::createDimensions() {
  IloDimension1 weight(_env, “Weight”);
  weight.setKey(“Weight”);
  _mdl.add(weight);
  IloDimension2 time(_env, IloEuclidean, “Time”);
  time.setKey(“Time”);
  _mdl.add(time);
  IloDimension2 distance(_env, IloEuclidean, IloFalse, “Distance”);
  distance.setKey(“Distance”);
  _mdl.add(distance);
}

// Create IloNodes
void RoutingModel::createIloNodes(char* nodeFileName) {
  IloCsvReader csvNodeReader(_env, nodeFileName);
  IloCsvReader::LineIterator  it(csvNodeReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char* name = line.getStringByHeader(“name”);
    IloNode node(_env,
                 line.getFloatByHeader(“x”),
                 line.getFloatByHeader(“y”),
                 0,
                 name);
    node.setKey(name);
    ++it;
  }
  csvNodeReader.end();
}

// Create vehicles
void RoutingModel::createVehicles(char* vehicleFileName) {
  IloDimension1 weight = IloDimension1::Find(_env, “Weight”);
  IloDimension2 time = IloDimension2::Find(_env, “Time”);
  IloDimension2 distance = IloDimension2::Find(_env, “Distance”);
  IloCsvReader csvVehicleReader(_env, vehicleFileName);
  IloCsvReader::LineIterator it(csvVehicleReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char * namefirst = line.getStringByHeader(“first”);
    char * namelast = line.getStringByHeader(“last”);
    char * name = line.getStringByHeader(“name”);
    IloNum capacity = line.getFloatByHeader(“capacity”);
    IloNum openTime = line.getFloatByHeader(“open”);
    IloNum closeTime = line.getFloatByHeader(“close”);
    IloNode node1 = IloNode::Find(_env, namefirst);
    IloNode node2 = IloNode::Find(_env, namelast);

    IloVisit first(node1, “depot”);
    _mdl.add(first.getCumulVar(weight) == 0);
    _mdl.add(first.getCumulVar(time) >= openTime);
    IloVisit last(node2, “depot”);
    _mdl.add(last.getCumulVar(time) <= closeTime);
    IloVehicle vehicle(first, last, name);

    vehicle.setCost(distance, 1);
    IloNumToNumSegmentFunction costFunction(_env);
    costFunction.setSlope(75, 100, 37.5, 0.5);
    costFunction.setSlope(100, IloInfinity, 100, 1);
    vehicle.setCost(distance, costFunction);

    vehicle.setCapacity(weight, capacity);
    _mdl.add(vehicle);
    ++it;
  }
  csvVehicleReader.end();
}

// Create visits
void RoutingModel::createVisits(char* visitsFileName) {
  IloDimension1 weight = IloDimension1::Find(_env, “Weight”);
  IloDimension2 time = IloDimension2::Find(_env, “Time”);
  IloCsvReader csvVisitReader(_env, visitsFileName);
  IloCsvReader::LineIterator it(csvVisitReader);
  while(it.ok()){
    IloCsvLine line = *it;
    //read visit data from files
    char * pickupVisitName = line.getStringByHeader(“pickup”);
    char * pickupNodeName = line.getStringByHeader(“pickupNode”);
    char * deliveryVisitName = line.getStringByHeader(“delivery”);
    char * deliveryNodeName = line.getStringByHeader(“deliveryNode”);
    IloNum quantity = line.getFloatByHeader(“quantity”);
    IloNum pickupMinTime  = line.getFloatByHeader(“pickupMinTime”);
    IloNum pickupMaxTime  = line.getFloatByHeader(“pickupMaxTime”);
    IloNum deliveryMinTime  = line.getFloatByHeader(“deliveryMinTime”);
    IloNum deliveryMaxTime  = line.getFloatByHeader(“deliveryMaxTime”);
    IloNum dropTime = line.getFloatByHeader(“dropTime”);
    IloNum pickupTime = line.getFloatByHeader(“pickupTime”);

    // read data nodes from the file nodes.csv
    // and create pickup and delivery nodes

    IloNode pickupNode = IloNode::Find(_env, pickupNodeName);
    IloNode deliveryNode = IloNode::Find(_env, deliveryNodeName);

    //create and add pickup and delivery visits
    IloVisit pickup(pickupNode, pickupVisitName);
    _mdl.add(pickup.getDelayVar(time) == pickupTime);
    _mdl.add(pickup.getTransitVar(weight) == quantity);
    _mdl.add(pickupMinTime <= pickup.getCumulVar(time) <= pickupMaxTime);
    _mdl.add(pickup);
    IloVisit delivery(deliveryNode, deliveryVisitName);
    _mdl.add(delivery.getDelayVar(time) == dropTime);
    _mdl.add(delivery.getTransitVar(weight) == -quantity);
    _mdl.add(deliveryMinTime <= delivery.getCumulVar(time) <= deliveryMaxTime);
    _mdl.add(delivery);
    //add pickup and delivery order constraint
    _mdl.add(IloOrderedVisitPair(_env, pickup, delivery));
    ++it;
  }
  csvVisitReader.end();
}

////////////////////////////////////////////////////////////////////////////////
// Solving
class RoutingSolver {
  IloModel           _mdl;
  IloSolver          _solver;
  IloRoutingSolution  _solution;

  IloBool findFirstSolution(IloGoal goal);
  void greedyImprove(IloNHood nhood, IloGoal subGoal);
  void improve(IloGoal subgoal);
public:
  RoutingSolver(RoutingModel mdl);
  ~RoutingSolver() {}
  IloRoutingSolution getSolution() const { return _solution; }
  void printInformation() const;
  void solve();
};

RoutingSolver::RoutingSolver(RoutingModel mdl)
  : _mdl(mdl.getModel()), _solver(mdl.getModel()), _solution(mdl.getModel()) {}

// Solving : find first solution
IloBool RoutingSolver::findFirstSolution(IloGoal goal) {
  if (!_solver.solve(goal)) {
    _solver.error() << “Infeasible Routing Plan” << endl;
    return IloFalse;
  }
  IloDispatcher dispatcher(_solver);
  _solver.out() << dispatcher.getTotalCost() << endl;
  _solution.store(_solver);
  return IloTrue;
}

// Improve solution using nhood
void RoutingSolver::greedyImprove(IloNHood nhood, IloGoal subGoal) {
  _solver.out() << “Improving solution” << endl;
  IloEnv env = _solver.getEnv();
  nhood.reset();
  IloGoal improve = IloSingleMove(env,
                                  _solution,
                                  nhood,
                                  IloImprove(env),
                                  subGoal);
  while (_solver.solve(improve)) {}
}

// Improve solution
void RoutingSolver::improve(IloGoal subGoal) {
  IloEnv env = _solver.getEnv();
  IloNHood nhood = IloTwoOpt(env)
                 + IloOrOpt(env)
                 + IloRelocate(env)
                 + IloCross(env)
                 + IloExchange(env);
  greedyImprove(nhood, subGoal);
}

// Display Dispatcher information
void RoutingSolver::printInformation() const {
  IloDispatcher dispatcher(_solver);
  _solver.printInformation();
  dispatcher.printInformation();
  _solver.out() << "===============" << endl
                << "Cost         : " << dispatcher.getTotalCost() << endl
                << "Number of vehicles used : "
                << dispatcher.getNumberOfVehiclesUsed() << endl
                << "Solution     : " << endl
                << IloVerbose(dispatcher) << endl;
}

// Solving
void RoutingSolver::solve() {
  IloDispatcher dispatcher(_solver);
  IloEnv env = _solver.getEnv();
  // Subgoal
  IloGoal instantiateCost = IloDichotomize(env,
                                           dispatcher.getCostVar(),
                                           IloFalse);
  IloGoal restoreSolution = IloRestoreSolution(env, _solution);
  IloGoal goal = IloSavingsGenerate(env) && instantiateCost;

  // Solving
  if (findFirstSolution(goal)) {
    improve(instantiateCost);
    _solver.solve(restoreSolution);
  }
}
///////////////////////////////////////////////////////////////////////////////

int main(int argc, char * argv[]) {
  IloEnv env;
  try {
    RoutingModel mdl(env, argc, argv);
    RoutingSolver solver(mdl);
    solver.solve();
    solver.printInformation();
  } catch(IloException& ex) {
    cerr << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Complete cost2 Output

```
/**
2583.49
Improving solution
Number of fails              : 0
Number of choice points       : 0
Number of variables           : 2359
Number of constraints         : 440
Reversible stack (bytes)      : 201024
Solver heap (bytes)           : 1025380
Solver global heap (bytes)    : 269064
And stack (bytes)             : 20124
Or stack (bytes)              : 44244
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 11160
Total memory used (bytes)     : 1575040
Elapsed time since creation   : 0.01
Number of nodes               : 51
Number of visits              : 80
Number of vehicles            : 15
Number of dimensions          : 3
Number of accepted moves      : 28
===============
Cost          : 1900.16
Number of vehicles used : 13
Solution      :
Unperformed visits : None

vehicle1
Cost coefficients : Distance[2]
     Route : depot -> visit35 -> visit49 -> visit36 -> visit50 -> depot
    Weight : depot [0], quantity [0..162] -> visit35 [0..162], quantity [8] ->
visit49 [8..170], quantity [30] -> visit36 [38..200], quantity [-8] -> visit50
[30..192], quantity [-30] -> depot [0..162], quantity (-Inf..Inf) TransitSum (-
Inf..Inf)
      Time : depot [0..19.1001], delay [0..19.1001] -> travel [41.0366], wait
[0..19.1001] -> visit35 [41.0366..60.1367], delay [10] -> travel [57.0789],
wait [0..19.1001] -> visit49 [108.115..127.216], delay [10] -> travel
[8.94427], wait [0..19.1001] -> visit36 [127.06..146.16], delay [10] -> travel
[46.8402], wait [0..19.1001] -> visit50 [183.9..203], delay [10] -> travel
[16.9706], wait [0..19.1295] -> depot [210.87..230], delay [0..Inf) -> travel
[0], wait [0..Inf) TransitSum [210.87..Inf)
  Distance : depot [0..Inf), delay [0..Inf) -> travel [41.0366], wait [0..Inf)
-> visit35 [0..Inf), delay [0..Inf) -> travel [57.0789], wait [0..Inf) ->
visit49 [0..Inf), delay [0..Inf) -> travel [8.94427], wait [0..Inf) -> visit36
[0..Inf), delay [0..Inf) -> travel [46.8402], wait [0..Inf) -> visit50
[0..Inf), delay [0..Inf) -> travel [16.9706], wait [0..Inf) -> depot [0..Inf),
delay [0..Inf) -> travel [0], wait [0..Inf) TransitSum [170.87..Inf)

vehicle2
Cost coefficients : Distance[2]
     Route : depot -> visit21 -> visit22 -> visit23 -> visit39 -> visit24 ->
visit40 -> depot
    Weight : depot [0], quantity [0..140] -> visit21 [0..140], quantity [11] ->
visit22 [11..151], quantity [-11] -> visit23 [0..140], quantity [29] -> visit39
[29..169], quantity [31] -> visit24 [60..200], quantity [-29] -> visit40
[31..171], quantity [-31] -> depot [0..140], quantity (-Inf..Inf) TransitSum (-
Inf..Inf)
      Time : depot [0..18.7919], delay [0..18.7919] -> travel [18.0278], wait
[0..18.7919] -> visit21 [18.0278..36.8197], delay [10] -> travel [10], wait
[0..18.7919] -> visit22 [38.0278..56.8197], delay [10] -> travel [11.1803],
wait [0..18.7919] -> visit23 [68..78], delay [10] -> travel [8.60233], wait
[0..50.9346] -> visit39 [86.6023..137.537], delay [10] -> travel [23.5372],
wait [0..50.9346] -> visit24 [120.14..171.074], delay [10] -> travel [26.9258],
wait [0..50.9346] -> visit40 [157.065..208], delay [10] -> travel [11.1803],
wait [0..51.7543] -> depot [178.246..230], delay [0..Inf) -> travel [0], wait
[0..Inf) TransitSum [169.454..Inf)
  Distance : depot [0..Inf), delay [0..Inf) -> travel [18.0278], wait [0..Inf)
-> visit21 [0..Inf), delay [0..Inf) -> travel [10], wait [0..Inf) -> visit22
[0..Inf), delay [0..Inf) -> travel [11.1803], wait [0..Inf) -> visit23
[0..Inf), delay [0..Inf) -> travel [8.60233], wait [0..Inf) -> visit39
[0..Inf), delay [0..Inf) -> travel [23.5372], wait [0..Inf) -> visit24
[0..Inf), delay [0..Inf) -> travel [26.9258], wait [0..Inf) -> visit40
[0..Inf), delay [0..Inf) -> travel [11.1803], wait [0..Inf) -> depot [0..Inf),
delay [0..Inf) -> travel [0], wait [0..Inf) TransitSum [109.454..Inf)

vehicle3
Cost coefficients : Distance[1.5..2]
     Route : depot -> visit7 -> visit8 -> visit45 -> visit46 -> visit47 ->
visit48 -> depot
    Weight : depot [0], quantity [0..173] -> visit7 [0..173], quantity [5] ->
visit8 [5..178], quantity [-5] -> visit45 [0..173], quantity [16] -> visit46
[16..189], quantity [-16] -> visit47 [0..173], quantity [27] -> visit48
[27..200], quantity [-27] -> depot [0..173], quantity (-Inf..Inf) TransitSum (-
Inf..Inf)
      Time : depot [0..61.5802], delay [0..61.5802] -> travel [21.2132], wait
[0..61.5802] -> visit7 [21.2132..82.7934], delay [10] -> travel [12.2066], wait
[0..61.5802] -> visit8 [95..105], delay [10] -> travel [6.40312], wait
[0..23.4234] -> visit45 [111.403..134.827], delay [10] -> travel [10.7703],
wait [0..23.4234] -> visit46 [132.173..155.597], delay [10] -> travel [10],
wait [0..23.4234] -> visit47 [152.173..175.597], delay [10] -> travel
[6.40312], wait [0..23.4234] -> visit48 [168.577..192], delay [10] -> travel
[27.8029], wait [0..23.6205] -> depot [206.379..230], delay [0..Inf) -> travel
[0], wait [0..Inf) TransitSum [154.799..Inf)
  Distance : depot [0..Inf), delay [0..Inf) -> travel [21.2132], wait [0..Inf)
-> visit7 [0..Inf), delay [0..Inf) -> travel [12.2066], wait [0..Inf) -> visit8
[0..Inf), delay [0..Inf) -> travel [6.40312], wait [0..Inf) -> visit45
[0..Inf), delay [0..Inf) -> travel [10.7703], wait [0..Inf) -> visit46
[0..Inf), delay [0..Inf) -> travel [10], wait [0..Inf) -> visit47 [0..Inf),
delay [0..Inf) -> travel [6.40312], wait [0..Inf) -> visit48 [0..Inf), delay
[0..Inf) -> travel [27.8029], wait [0..Inf) -> depot [0..Inf), delay [0..Inf) -
> travel [0], wait [0..Inf) TransitSum [94.7992..Inf)

vehicle4
Cost coefficients : Distance[2]
     Route : depot -> visit33 -> visit29 -> visit34 -> visit9 -> visit30 ->
visit10 -> depot
    Weight : depot [0], quantity [0..175] -> visit33 [0..175], quantity [11] ->
visit29 [11..186], quantity [9] -> visit34 [20..195], quantity [-11] -> visit9
[9..184], quantity [16] -> visit30 [25..200], quantity [-9] -> visit10
[16..191], quantity [-16] -> depot [0..175], quantity (-Inf..Inf) TransitSum (-
Inf..Inf)
      Time : depot [0..13.1563], delay [0..13.1563] -> travel [24.7588], wait
[0..13.1563] -> visit33 [24.7588..37.9152], delay [10] -> travel [14.8661],
wait [0..13.1563] -> visit29 [49.6249..62.7813], delay [10] -> travel
[13.0384], wait [0..13.1563] -> visit34 [72.6633..85.8197], delay [10] ->
travel [11.1803], wait [0..13.1563] -> visit9 [97..107], delay [10] -> travel
[15], wait [0..52] -> visit30 [122..174], delay [10] -> travel [10], wait
[0..52] -> visit10 [142..194], delay [10] -> travel [25.4951], wait
[0..52.5049] -> depot [177.495..230], delay [0..Inf) -> travel [0], wait
[0..Inf) TransitSum [174.339..Inf)
```

```
  Distance : depot [0..Inf), delay [0..Inf) -> travel [24.7588], wait [0..Inf)
-> visit33 [0..Inf), delay [0..Inf) -> travel [14.8661], wait [0..Inf) ->
visit29 [0..Inf), delay [0..Inf) -> travel [13.0384], wait [0..Inf) -> visit34
[0..Inf), delay [0..Inf) -> travel [11.1803], wait [0..Inf) -> visit9 [0..Inf),
delay [0..Inf) -> travel [15], wait [0..Inf) -> visit30 [0..Inf), delay
[0..Inf) -> travel [10], wait [0..Inf) -> visit10 [0..Inf), delay [0..Inf) ->
travel [25.4951], wait [0..Inf) -> depot [0..Inf), delay [0..Inf) -> travel
[0], wait [0..Inf) TransitSum [114.339..Inf)

vehicle5
Cost coefficients : Distance[2]
     Route : depot -> visit31 -> visit19 -> visit11 -> visit32 -> visit20 ->
visit12 -> depot
    Weight : depot [0], quantity [0..144] -> visit31 [0..144], quantity [27] ->
visit19 [27..171], quantity [17] -> visit11 [44..188], quantity [12] -> visit32
[56..200], quantity [-27] -> visit20 [29..173], quantity [-17] -> visit12
[12..156], quantity [-12] -> depot [0..144], quantity (-Inf..Inf) TransitSum (-
Inf..Inf)
      Time : depot [0..14.5761], delay [0..14.5761] -> travel [17.4642], wait
[0..14.5761] -> visit31 [17.4642..32.0404], delay [10] -> travel [17.8885],
wait [0..14.5761] -> visit19 [45.3528..59.9289], delay [10] -> travel
[7.07107], wait [0..14.5761] -> visit11 [67..77], delay [10] -> travel
[15.5242], wait [0..51.2917] -> visit32 [92.5242..143.816], delay [10] ->
travel [10.7703], wait [0..51.2917] -> visit20 [113.295..164.586], delay [10] -
> travel [30.4138], wait [0..51.2917] -> visit12 [153.708..205], delay [10] ->
travel [15], wait [0..51.2917] -> depot [178.708..230], delay [0..Inf) ->
travel [0], wait [0..Inf) TransitSum [174.132..Inf)
  Distance : depot [0..Inf), delay [0..Inf) -> travel [17.4642], wait [0..Inf)
-> visit31 [0..Inf), delay [0..Inf) -> travel [17.8885], wait [0..Inf) ->
visit19 [0..Inf), delay [0..Inf) -> travel [7.07107], wait [0..Inf) -> visit11
[0..Inf), delay [0..Inf) -> travel [15.5242], wait [0..Inf) -> visit32
[0..Inf), delay [0..Inf) -> travel [10.7703], wait [0..Inf) -> visit20
[0..Inf), delay [0..Inf) -> travel [30.4138], wait [0..Inf) -> visit12
[0..Inf), delay [0..Inf) -> travel [15], wait [0..Inf) -> depot [0..Inf), delay
[0..Inf) -> travel [0], wait [0..Inf) TransitSum [114.132..Inf)

vehicle6
Cost coefficients : Distance[1.5..2]
     Route : depot -> visit41 -> visit15 -> visit42 -> visit16 -> visit5 ->
visit6 -> depot
    Weight : depot [0], quantity [0..174] -> visit41 [0..174], quantity [5] ->
visit15 [5..179], quantity [8] -> visit42 [13..187], quantity [-5] -> visit16
[8..182], quantity [-8] -> visit5 [0..174], quantity [26] -> visit6 [26..200],
quantity [-26] -> depot [0..174], quantity (-Inf..Inf) TransitSum (-Inf..Inf)
      Time : depot [0..19.9727], delay [0..19.9727] -> travel [28.8617], wait
[0..19.9727] -> visit41 [28.8617..48.8345], delay [10] -> travel [12.1655],
wait [0..19.9727] -> visit15 [61..71], delay [10] -> travel [9.21954], wait
[0..60.4756] -> visit42 [80.2195..140.695], delay [10] -> travel [16.1245],
wait [0..60.4756] -> visit16 [106.344..166.82], delay [10] -> travel [11.1803],
wait [0..60.4756] -> visit5 [127.524..188], delay [10] -> travel [10], wait
[0..60.4756] -> visit6 [147.524..208], delay [10] -> travel [11.1803], wait
[0..61.2953] -> depot [168.705..230], delay [0..Inf) -> travel [0], wait
[0..Inf) TransitSum [158.732..Inf)
  Distance : depot [0..Inf), delay [0..Inf) -> travel [28.8617], wait [0..Inf)
-> visit41 [0..Inf), delay [0..Inf) -> travel [12.1655], wait [0..Inf) ->
visit15 [0..Inf), delay [0..Inf) -> travel [9.21954], wait [0..Inf) -> visit42
[0..Inf), delay [0..Inf) -> travel [16.1245], wait [0..Inf) -> visit16
[0..Inf), delay [0..Inf) -> travel [11.1803], wait [0..Inf) -> visit5 [0..Inf),
delay [0..Inf) -> travel [10], wait [0..Inf) -> visit6 [0..Inf), delay [0..Inf)
-> travel [11.1803], wait [0..Inf) -> depot [0..Inf), delay [0..Inf) -> travel
[0], wait [0..Inf) TransitSum [98.732..Inf)

vehicle7
Cost coefficients : Distance[1..2]
     Route : depot -> visit27 -> visit28 -> visit25 -> visit26 -> depot
    Weight : depot [0], quantity [0..184] -> visit27 [0..184], quantity [16] ->
visit28 [16..200], quantity [-16] -> visit25 [0..184], quantity [6] -> visit26
[6..190], quantity [-6] -> depot [0..184], quantity (-Inf..Inf) TransitSum (-
Inf..Inf)
      Time : depot [0..114.52], delay [0..114.52] -> travel [5], wait
[0..114.52] -> visit27 [5..119.52], delay [10] -> travel [6.7082], wait
[0..114.52] -> visit28 [21.7082..136.228], delay [10] -> travel [29.4109], wait
[0..114.52] -> visit25 [172..175.639], delay [10] -> travel [22.3607], wait
[0..3.63932] -> visit26 [204.361..208], delay [10] -> travel [11.1803], wait
[0..4.45898] -> depot [225.541..230], delay [0..Inf) -> travel [0], wait
[0..Inf) TransitSum [114.66..Inf)
  Distance : depot [0..Inf), delay [0..Inf) -> travel [5], wait [0..Inf) ->
visit27 [0..Inf), delay [0..Inf) -> travel [6.7082], wait [0..Inf) -> visit28
[0..Inf), delay [0..Inf) -> travel [29.4109], wait [0..Inf) -> visit25
[0..Inf), delay [0..Inf) -> travel [22.3607], wait [0..Inf) -> visit26
[0..Inf), delay [0..Inf) -> travel [11.1803], wait [0..Inf) -> depot [0..Inf),
delay [0..Inf) -> travel [0], wait [0..Inf) TransitSum [74.6601..Inf)

vehicle8
Cost coefficients : Distance[1.5..2]
     Route : depot -> visit37 -> visit38 -> depot
    Weight : depot [0], quantity [0..192] -> visit37 [0..192], quantity [8] ->
visit38 [8..200], quantity [-8] -> depot [0..192], quantity (-Inf..Inf)
TransitSum (-Inf..Inf)
      Time : depot [0..40.5736], delay [0..40.5736] -> travel [21.2132], wait
[0..40.5736] -> visit37 [21.2132..61.7868], delay [10] -> travel [21.2132],
wait [0..40.5736] -> visit38 [83..93], delay [10] -> travel [42.4264], wait
[0..94.5736] -> depot [135.426..230], delay [0..Inf) -> travel [0], wait
[0..Inf) TransitSum [104.853..Inf)
  Distance : depot [0..Inf), delay [0..Inf) -> travel [21.2132], wait [0..Inf)
-> visit37 [0..Inf), delay [0..Inf) -> travel [21.2132], wait [0..Inf) ->
visit38 [0..Inf), delay [0..Inf) -> travel [42.4264], wait [0..Inf) -> depot
[0..Inf), delay [0..Inf) -> travel [0], wait [0..Inf) TransitSum [84.8528..Inf)

vehicle9
Cost coefficients : Distance[1..2]
     Route : depot -> visit3 -> visit4 -> depot
    Weight : depot [0], quantity [0..187] -> visit3 [0..187], quantity [13] ->
visit4 [13..200], quantity [-13] -> depot [0..187], quantity (-Inf..Inf)
TransitSum (-Inf..Inf)
      Time : depot [0..101.639], delay [0..101.639] -> travel [22.3607], wait
[0..101.639] -> visit3 [22.3607..124], delay [10] -> travel [25], wait
[0..101.639] -> visit4 [149..159], delay [10] -> travel [25], wait [0..46] ->
depot [184..230], delay [0..Inf) -> travel [0], wait [0..Inf) TransitSum
[92.3607..Inf)
  Distance : depot [0..Inf), delay [0..Inf) -> travel [22.3607], wait [0..Inf)
-> visit3 [0..Inf), delay [0..Inf) -> travel [25], wait [0..Inf) -> visit4
[0..Inf), delay [0..Inf) -> travel [25], wait [0..Inf) -> depot [0..Inf), delay
[0..Inf) -> travel [0], wait [0..Inf) TransitSum [72.3607..Inf)

vehicle10
```

```
Cost coefficients : Distance[1.5..2]
     Route : depot -> visit43 -> visit44 -> depot
    Weight : depot [0], quantity [0..193] -> visit43 [0..193], quantity [7] ->
visit44 [7..200], quantity [-7] -> depot [0..193], quantity (-Inf..Inf)
TransitSum (-Inf..Inf)
      Time : depot [0..18.5452], delay [0..18.5452] -> travel [34.176], wait
[0..18.5452] -> visit43 [34.176..52.7212], delay [10] -> travel [16.2788], wait
[0..18.5452] -> visit44 [69..79], delay [10] -> travel [31.8904], wait
[0..119.11] -> depot [110.89..230], delay [0..Inf) -> travel [0], wait [0..Inf)
TransitSum [102.345..Inf)
  Distance : depot [0..Inf), delay [0..Inf) -> travel [34.176], wait [0..Inf) -
> visit43 [0..Inf), delay [0..Inf) -> travel [16.2788], wait [0..Inf) ->
visit44 [0..Inf), delay [0..Inf) -> travel [31.8904], wait [0..Inf) -> depot
[0..Inf), delay [0..Inf) -> travel [0], wait [0..Inf) TransitSum [82.3453..Inf)

vehicle11
Cost coefficients : Distance[1..2]
     Route : depot -> visit17 -> visit18 -> depot
    Weight : depot [0], quantity [0..198] -> visit17 [0..198], quantity [2] ->
visit18 [2..200], quantity [-2] -> depot [0..198], quantity (-Inf..Inf)
TransitSum (-Inf..Inf)
      Time : depot [0..145.558], delay [0..145.558] -> travel [30.4138], wait
[0..145.558] -> visit17 [30.4138..175.972], delay [10] -> travel [18.0278],
wait [0..145.558] -> visit18 [58.4416..204], delay [10] -> travel [15.8114],
wait [0..145.747] -> depot [84.253..230], delay [0..Inf) -> travel [0], wait
[0..Inf) TransitSum [84.253..Inf)
  Distance : depot [0..Inf), delay [0..Inf) -> travel [30.4138], wait [0..Inf)
-> visit17 [0..Inf), delay [0..Inf) -> travel [18.0278], wait [0..Inf) ->
visit18 [0..Inf), delay [0..Inf) -> travel [15.8114], wait [0..Inf) -> depot
[0..Inf), delay [0..Inf) -> travel [0], wait [0..Inf) TransitSum [64.253..Inf)

vehicle12
Cost coefficients : Distance[1..2]
     Route : depot -> visit13 -> visit14 -> depot
    Weight : depot [0], quantity [0..177] -> visit13 [0..177], quantity [23] ->
visit14 [23..200], quantity [-23] -> depot [0..177], quantity (-Inf..Inf)
TransitSum (-Inf..Inf)
      Time : depot [0..144.606], delay [0..144.606] -> travel [11.1803], wait
[0..144.606] -> visit13 [109..155.787], delay [10] -> travel [21.2132], wait
[0..46.7868] -> visit14 [140.213..187], delay [10] -> travel [32.0156], wait
[0..47.7712] -> depot [182.229..230], delay [0..Inf) -> travel [0], wait
[0..Inf) TransitSum [84.4092..Inf)
  Distance : depot [0..Inf), delay [0..Inf) -> travel [11.1803], wait [0..Inf)
-> visit13 [0..Inf), delay [0..Inf) -> travel [21.2132], wait [0..Inf) ->
visit14 [0..Inf), delay [0..Inf) -> travel [32.0156], wait [0..Inf) -> depot
[0..Inf), delay [0..Inf) -> travel [0], wait [0..Inf) TransitSum [64.4092..Inf)

vehicle13
Cost coefficients : Distance[1..2]
     Route : depot -> visit1 -> visit2 -> depot
    Weight : depot [0], quantity [0..190] -> visit1 [0..190], quantity [10] ->
visit2 [10..200], quantity [-10] -> depot [0..190], quantity (-Inf..Inf)
TransitSum (-Inf..Inf)
      Time : depot [0..144.211], delay [0..144.211] -> travel [15.2315], wait
[0..144.211] -> visit1 [15.2315..159.442], delay [10] -> travel [32.5576], wait
[0..144.211] -> visit2 [57.7892..202], delay [10] -> travel [18], wait
[0..144.211] -> depot [85.7892..230], delay [0..Inf) -> travel [0], wait
[0..Inf) TransitSum [85.7892..Inf)
```

```
  Distance : depot [0..Inf), delay [0..Inf) -> travel [15.2315], wait [0..Inf)
-> visit1 [0..Inf), delay [0..Inf) -> travel [32.5576], wait [0..Inf) -> visit2
[0..Inf), delay [0..Inf) -> travel [18], wait [0..Inf) -> depot [0..Inf), delay
[0..Inf) -> travel [0], wait [0..Inf) TransitSum [65.7892..Inf)

vehicle14 : Unused

vehicle15 : Unused

*/
```

C H A P T E R

12

Docking Bays: Modeling External Resources

In this lesson, you will learn how to:

- ◆ model a shared external resource that is used by vehicles
- ◆ create an activity that uses that resource
- ◆ incorporate IBM ILOG Scheduler into your Dispatcher programs

Describe

Vehicles and their drivers may occasionally need resources that can be considered “external” to the delivery process itself. For example, there may be handling resources such as people, cranes, or docking bays at a depot that are necessary to load the trucks, or there may be trailers (shared between vehicles) that are used for the deliveries. External resources can be modeled with IBM ILOG Scheduler and incorporated into your Dispatcher program, thus increasing your ability to model various delivery problems.

This lesson demonstrates the use of external resources by starting with a standard Pickup and Delivery Problem (PDP) and adding the requirement to use one of a limited number of depot docking bays to load the trucks. Three docking bays are used to load up to 15 vehicles.

There is a break on each docking bay, which represents a break for the people loading the vehicles.

Step 1 Describe the problem

The first step is to write a natural language description of the problem. A good way to start this process is to analyze the constraints and objectives.

What are the constraints in this problem?

- Delivery trucks must be loaded at a limited number of docking bays at the depot.
- There is a break on the docking bay resource that represents a break for the people loading the trucks.
- Constraints that exist for a regular PDP exist here as well, such as pickups and deliveries with time window constraints, and capacity constraints on the trucks. In this example, a csv data file positions all pickups at the docking bays.

The objective is to minimize the cost of the delivery of all the parcels.

Model

Once you have written a description of this problem, you can use Dispatcher and Scheduler classes to model it.

Step 2 Open the example file

Open the example file `YourDispatcherHome/examples/src/tutorial/bays_partial.cpp` in your development environment.

This lesson requires the IBM ILOG Scheduler library to model the resource and the pickup activity.

Step 3 Include the Scheduler library

Add the following code after the comment `//Include the Scheduler library.`

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

As in the previous examples, you will use a `RoutingModel` class to call the functions that create the dimensions, nodes, vehicles, and visits. In this example, `RoutingModel` includes some Scheduler code and a function to create the docking bays.

Step 4 Declare the RoutingModel class

Add the following code after the comment `//Declare the RoutingModel class`.

```
class RoutingModel {
  IloEnv               _env;
  IloModel             _mdl;
  IloDiscreteResource _bays;
  IloInt               _schedGranularity;

  void createDepotDockingBays(char* baysFileName);
  void createDimensions();
  void createNodes(char* nodeFileName);
  void createVehicles(char* vehicleFileName);
  void createVisits(char* visitsFileName);
public:
  RoutingModel(IloEnv env,
               int argc,
               char* argv[],
               IloInt schedGranularity = 100);
  ~RoutingModel() {}
  IloDiscreteResource getBays() const { return _bays; }
  IloEnv getEnv() const { return _env; }
  IloModel getModel() const { return _mdl; }
};
```

The docking bays are modeled in Scheduler as an instance of `IloDiscreteResource`. A discrete resource has a capacity that can vary over time, but is always available only as a positive integer.

The integer variable `_schedGranularity` is used as a scaling factor to relate floating-point dimension variables of Dispatcher to the integer values of the Scheduler resource time variables.

The function `createDepotDockingBays` creates the docking bays and their associated breaks.

Step 5 Create the docking bays

Add the following code after the comment `//Create the docking bays`.

```
void RoutingModel::createDepotDockingBays(char* baysFileName) {
  // Scheduler environment settings
  IloCsvReader csvBaysReader(_env, baysFileName);
  IloCsvReader::LineIterator it(csvBaysReader);
  while (it.ok()) {
    IloCsvLine line = *it;
    IloSchedulerEnv schedEnv(_env);
    IloInt horizon = line.getIntByHeader("horizon");
    IloInt nbOfBays = line.getIntByHeader("bays");
    IloInt breakStart = line.getIntByHeader("breakStart");
    IloInt breakEnd = line.getIntByHeader("breakEnd");

    schedEnv.setHorizon(horizon * _schedGranularity);
    _bays = IloDiscreteResource(_env, nbOfBays);
    _bays.addBreak(breakStart * _schedGranularity,
                   breakEnd * _schedGranularity);
    ++it;
  }
  csvBaysReader.end();
}
```

Data for the discrete resource is read in from a csv file. The `horizon` is used to set the ending point of the time window over which the resource capacity constraints are enforced. The `horizon` is multiplied by a granularity (`_schedGranularity`) of 100 and then applied to an instance of `IloSchedulerEnv` (`schedEnv`). `IloSchedulerEnv` serves as the repository of all the default parameters used in creating Scheduler modeling objects.

The code `_bays = IloDiscreteResource(_env, nbOfBays);` creates the docking bays as a discrete resource of capacity `nbOfBays` (in this example, “3”).

The break start and end times are also multiplied by `_schedGranularity` and then added to the `_bays` resource.

Here is an example to demonstrate the effect of the granularity: assume that *v* is a visit initially starting after 1.56 and ending before 3.34, with a duration of 1. If `_schedGranularity` == 1, then *v* will start after 2 and end before 3.34, and the resource will be required between 2 and 3. If `_schedGranularity` == 100, then *v* will start after 1.56 and end before 3.34, and the resource will be required at least between 156 and 256.

The dimensions, nodes, and vehicles are created just as for the PDP. The visits require additional code to model the docking bay pickups.

Step 6 Create the docking bay pickups

Add the following code after the comment `//Create the docking bay pickups`.

```
IloNumVar process(_env, 0, pickupMaxTime * _schedGranularity, ILOINT);
IloActivity vehicleLoad(_env, process);
IloNumVar start(_env, 0,  pickupMaxTime * _schedGranularity, ILOINT);
IloNumVar end(_env, 0, pickupMaxTime * _schedGranularity, ILOINT);
_mdl.add(start == vehicleLoad.getStartExpr());
_mdl.add(end == vehicleLoad.getEndExpr());
vehicleLoad.setName(pickupVisitName);
vehicleLoad.setBreakable();
_mdl.add(vehicleLoad.requires(_bays, 1));
_mdl.add(start == pickup.getCumulVar(time) * _schedGranularity);
_mdl.add(end <= (pickup.getCumulVar(time) + pickup.getDelayVar(time))
         * _schedGranularity);
_mdl.add(process >= pickupTime * _schedGranularity);
```

This section of the function `CreateVisits` models the pickup and loading activity at the docking bay as a Scheduler activity `vehicleLoad`. The Scheduler member function `IloActivity::setBreakable` allows `vehicleLoad` to be interrupted by the personnel break. The member function `IloActivity::requires` is a resource constraint that states that the activity `vehicleLoad` requires (or *consumes*) exactly one of the resource `_bays`. For discrete resources, the activity requires the stated resource capacity at all times after the activity's start time (for the entire time the activity is occurring).

The activity `vehicleLoad` is constructed with a start time of `start`, an end time of `end`, and a processing time (total time the activity occurs) of `process`. Note that `start`, `end` and `process` are constructed from the floating point cumul and delay variables, which are then scaled by the granularity.

This section of code links the Dispatcher pickup visits to the Scheduler activities. It is important to have the visit cover the entire time of the activity; in other words, both start at the same time and the visits end after the activity end time (cumulVar + delayVar).

The processing time is constrained to be greater than or equal to the actual time to perform the pickup (it can be more than that due to breaks). The delay variable of the pickup is greater than or equal to the `pickupTime` to take into account the fact that the visit covers the activity, which itself can be interrupted by a break.

Solve

The solution is largely computed, improved, and displayed by methods previously presented for the PDP problem, but with two additions. A small section of code is added to iterate through the Scheduler activity, and a Scheduler subgoal is added to instantiate the starting times of the visits.

Step 7 Add the activity iteration

Add the following code after the comment `//Add the activity iteration.`

```
IlcScheduler sched(_solver);
for (IlcActivityIterator iter(sched); iter.ok(); ++iter) {
  _solver.out() << *iter << endl;
```

This section of the function `RoutingSolver::printInformation` iterates through the activity schedule to retrieve solution information for display.

Step 8 Add the subgoals

Add the following code after the comment `//Add the subgoals.`

```
IloGoal instantiateCost = IloDichotomize(env,
                                         dispatcher.getCostVar(),
                                         IloFalse);
IloGoal schedGoal = IloSetTimesForward(env);
IloGoal subGoal = schedGoal && instantiateCost;
IloGoal restoreSolution = IloRestoreSolution(env, _solution) && subGoal;
IloGoal goal = IloSavingsGenerate(env, schedGoal) && instantiateCost;
```

The predefined Scheduler subgoal `IloSetTimesForward` instantiates the activity variables, which via propagation, in turn instantiates the time variables of the visits. It is necessary to do this to make certain that the current solution is feasible.

Step 9 Compile and run the program

The solution improvement phase finds a solution using 6 vehicles with a cost of 1432.17 units:

```
  1641.01
Improving solution
Number of fails               : 0
Number of choice points       : 1068
Number of variables           : 2215
Number of constraints         : 564
Reversible stack (bytes)      : 385944
Solver heap (bytes)           : 1041460
Solver global heap (bytes)    : 184996
And stack (bytes)             : 20124
Or stack (bytes)              : 44244
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 9148
Total memory used (bytes)     : 1689960
Running time since creation   : 0.023437
Number of nodes               : 51
Number of visits              : 70
Number of vehicles            : 15
Number of dimensions          : 3
Number of accepted moves      : 11
===============
Cost          : 1432.17
Number of vehicles used : 6
```

Review Exercises

For answers, see “Suggested Answers” on page 311.

1. What is a discrete resource?
2. What Scheduler class is used to model a discrete resource?
3. What member function do you use to model an activity using a resource?
4. What member function do you use to allow an activity to be interrupted by breaks?

Suggested Answers

Exercise 1

What is a discrete resource?

Suggested Answer

A discrete resource is a resource that has a *discrete* capacity; that is, the resource is available in units of positive integers, and the resource is either being used in its entirety (1) or it is fully available. Examples include three bricklayers, five trucks, or seven docking bays.

Exercise 2

What Scheduler class is used to model a discrete resource?

Suggested Answer

`IloDiscreteResource`.

Exercise 3

What member function do you use to model an activity using a resource?

Suggested Answer

In this example, you used the IBM ILOG Scheduler member function `IloActivity::requires`. Depending on the type of the resource, there are other member functions available to model resource consumption.

Exercise 4

What member function do you use to allow an activity to be interrupted by breaks?

Suggested Answer

The IBM ILOG Scheduler member function `IloActivity::setBreakable` allows an activity to be interrupted by breaks.

Complete Program

The complete program follows. You can also view it online in the file `YourDispatcherHome/examples/src/bays.cpp`.

```
// -------------------------------------------------------------- -*- C++ -*-
// File: examples/src/bays.cpp
// --------------------------------------------------------------------------


#include <ildispat/ilodispatcher.h>

#include <ilsched/iloscheduler.h>

ILOSTLBEGIN
```

```
///////////////////////////////////////////////////////////////////////////////
// Modeling
class RoutingModel {
  IloEnv              _env;
  IloModel            _mdl;
  IloDiscreteResource _bays;
  IloInt              _schedGranularity;

  void createDepotDockingBays(char* baysFileName);
  void createDimensions();
  void createNodes(char* nodeFileName);
  void createVehicles(char* vehicleFileName);
  void createVisits(char* visitsFileName);
public:
  RoutingModel(IloEnv env,
               int argc,
               char* argv[],
               IloInt schedGranularity = 100);
  ~RoutingModel() {}
  IloDiscreteResource getBays() const { return _bays; }
  IloEnv getEnv() const { return _env; }
  IloModel getModel() const { return _mdl; }
};

RoutingModel::RoutingModel(IloEnv env,
                           int argc,
                           char * argv[],
                           IloInt schedGranularity)
  : _env(env), _mdl(env), _schedGranularity(schedGranularity) {

  createDimensions();

  char* nodeFileName;
  if(argc < 4)
    nodeFileName = (char *) “../../../examples/data/pdp/nodes.csv”;
  else
    nodeFileName = argv[3];
  createNodes(nodeFileName);

  char* baysFileName;
  if (argc < 5)
    baysFileName = (char*) “../../../examples/data/bays.csv”;
  else
    baysFileName = argv[4];
  createDepotDockingBays(baysFileName);

  char* vehiclesFileName;
  if (argc < 2)
    vehiclesFileName = (char *) “../../../examples/data/pdp/vehicles.csv”;
  else
    vehiclesFileName = argv[1];
  createVehicles(vehiclesFileName);

  char* visitsFileName;
  if (argc < 3)
    visitsFileName =
      (char*) “../../../examples/data/pdp/visitsWithPickupInDepot.csv”;
  else
```

```
    visitsFileName = argv[2];
  createVisits(visitsFileName);
}

// Create docking bays
void RoutingModel::createDepotDockingBays(char* baysFileName) {
  // Scheduler environment settings
  IloCsvReader csvBaysReader(_env, baysFileName);
  IloCsvReader::LineIterator it(csvBaysReader);
  while (it.ok()) {
    IloCsvLine line = *it;
    IloSchedulerEnv schedEnv(_env);
    IloInt horizon = line.getIntByHeader("horizon");
    IloInt nbOfBays = line.getIntByHeader("bays");
    IloInt breakStart = line.getIntByHeader("breakStart");
    IloInt breakEnd = line.getIntByHeader("breakEnd");

    schedEnv.setHorizon(horizon * _schedGranularity);
    _bays = IloDiscreteResource(_env, nbOfBays);
    _bays.addBreak(breakStart * _schedGranularity,
                   breakEnd * _schedGranularity);
    ++it;
  }
  csvBaysReader.end();
}

// Create dimensions
void RoutingModel::createDimensions() {
  IloDimension1 weight(_env, "Weight");
  weight.setKey("Weight");
  _mdl.add(weight);
  IloDimension2 time(_env, IloEuclidean, "Time");
  time.setKey("Time");
  _mdl.add(time);
  IloDimension2 distance(_env, IloEuclidean, IloFalse, "Distance");
  distance.setKey("Distance");
  _mdl.add(distance);
}

// Create nodes
void RoutingModel::createNodes(char* nodeFileName) {
  IloCsvReader csvNodeReader(_env, nodeFileName);
  IloCsvReader::LineIterator  it(csvNodeReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char* name = line.getStringByHeader("name");
    IloNode node(_env,
                 line.getFloatByHeader("x"),
                 line.getFloatByHeader("y"),
                 0,
                 name);
    node.setKey(name);
    ++it;
  }
  csvNodeReader.end();
}

// Create vehicles
```

```
void RoutingModel::createVehicles(char* vehicleFileName) {
  IloDimension1 weight = IloDimension1::Find(_env, “Weight”);
  IloDimension2 time = IloDimension2::Find(_env, “Time”);
  IloDimension2 distance = IloDimension2::Find(_env, “Distance”);

  IloCsvReader csvVehicleReader(_env, vehicleFileName);
  IloCsvReader::LineIterator it(csvVehicleReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char * namefirst = line.getStringByHeader(“first”);
    char * namelast = line.getStringByHeader(“last”);
    char * name = line.getStringByHeader(“name”);
    IloNum capacity = line.getFloatByHeader(“capacity”);
    IloNum openTime = line.getFloatByHeader(“open”);
    IloNum closeTime = line.getFloatByHeader(“close”);
    IloNode node1 = IloNode::Find(_env, namefirst);
    IloNode node2 = IloNode::Find(_env, namelast);

    IloVisit first(node1, “depot”);
    _mdl.add(first.getCumulVar(weight) == 0);
    _mdl.add(first.getCumulVar(time) >= openTime);

    IloVisit last(node2, “depot”);
    _mdl.add(last.getCumulVar(time) <= closeTime);
    IloVehicle vehicle(first, last, name);
    vehicle.setCost(distance, 1.0);
    vehicle.setCost(time, 1.0);
    vehicle.setCapacity(weight, capacity);
    _mdl.add(vehicle);
    ++it;
  }
  csvVehicleReader.end();
}

// Create visits
void RoutingModel::createVisits(char* visitsFileName) {
  IloDimension1 weight = IloDimension1::Find(_env, “Weight”);
  IloDimension2 time = IloDimension2::Find(_env, “Time”);

  IloCsvReader csvVisitReader(_env, visitsFileName);
  IloCsvReader::LineIterator it(csvVisitReader);
  while(it.ok()){
    IloCsvLine line = *it;
    //read visit data from files
    char * pickupVisitName = line.getStringByHeader(“pickup”);
    char * pickupNodeName = line.getStringByHeader(“pickupNode”);
    char * deliveryVisitName = line.getStringByHeader(“delivery”);
    char * deliveryNodeName = line.getStringByHeader(“deliveryNode”);
    IloNum quantity = line.getFloatByHeader(“quantity”);
    IloNum pickupMinTime  = line.getFloatByHeader(“pickupMinTime”);
    IloNum pickupMaxTime  = line.getFloatByHeader(“pickupMaxTime”);
    IloNum deliveryMinTime  = line.getFloatByHeader(“deliveryMinTime”);
    IloNum deliveryMaxTime  = line.getFloatByHeader(“deliveryMaxTime”);
    IloNum dropTime = line.getFloatByHeader(“dropTime”);
    IloNum pickupTime = line.getFloatByHeader(“pickupTime”);

    // read data nodes from the file nodes.csv
    // and create pickup and delivery nodes
    IloNode pickupNode = IloNode::Find(_env, pickupNodeName);
    IloNode deliveryNode = IloNode::Find(_env, deliveryNodeName);
    //create and add pickup and delivery visits
    IloVisit pickup(pickupNode, pickupVisitName);
    _mdl.add(pickup.getDelayVar(time) >= pickupTime);
    _mdl.add(pickup.getTransitVar(weight) == quantity);
    _mdl.add(pickupMinTime <= pickup.getCumulVar(time) <= pickupMaxTime);
    _mdl.add(pickup);
    IloVisit delivery(deliveryNode, deliveryVisitName);
    _mdl.add(delivery.getDelayVar(time) == dropTime);
    _mdl.add(delivery.getTransitVar(weight) == -quantity);
    _mdl.add(deliveryMinTime <= delivery.getCumulVar(time) <= deliveryMaxTime);
    _mdl.add(delivery);
    //add pickup and delivery order constraint
    _mdl.add(IloOrderedVisitPair(_env, pickup, delivery));

    // Pickup requiring docking bay

    IloNumVar process(_env, 0, pickupMaxTime * _schedGranularity, ILOINT);
    IloActivity vehicleLoad(_env, process);
    IloNumVar start(_env, 0,  pickupMaxTime * _schedGranularity, ILOINT);
    IloNumVar end(_env, 0, pickupMaxTime * _schedGranularity, ILOINT);
    _mdl.add(start == vehicleLoad.getStartExpr());
    _mdl.add(end == vehicleLoad.getEndExpr());
    vehicleLoad.setName(pickupVisitName);
    vehicleLoad.setBreakable();
    _mdl.add(vehicleLoad.requires(_bays, 1));
    _mdl.add(start == pickup.getCumulVar(time) * _schedGranularity);
    _mdl.add(end <= (pickup.getCumulVar(time) + pickup.getDelayVar(time))
             * _schedGranularity);
    _mdl.add(process >= pickupTime * _schedGranularity);

    ++it;
  }
  csvVisitReader.end();
}

////////////////////////////////////////////////////////////////////////////////
// Solving
class RoutingSolver {
  IloModel            _mdl;
  IloSolver           _solver;
  IloRoutingSolution  _solution;

  IloBool findFirstSolution(IloGoal goal);
  void greedyImprove(IloNHood nhood, IloGoal subGoal);
  void improve(IloGoal subgoal);
public:
  RoutingSolver(RoutingModel mdl);
  ~RoutingSolver() {}
  IloRoutingSolution getSolution() const { return _solution; }
  void printInformation() const;
  void solve();
};

RoutingSolver::RoutingSolver(RoutingModel mdl)
  : _mdl(mdl.getModel()), _solver(mdl.getModel()), _solution(mdl.getModel()) {}
```

```
// Solving : find first solution
IloBool RoutingSolver::findFirstSolution(IloGoal goal) {
  if (!_solver.solve(goal)) {
    _solver.error() << “Infeasible Routing Plan” << endl;
    return IloFalse;
  }
  IloDispatcher dispatcher(_solver);
  _solver.out() << dispatcher.getTotalCost() << endl;
  _solution.store(_solver);
  return IloTrue;
}

// Improve solution using nhood
void RoutingSolver::greedyImprove(IloNHood nhood, IloGoal subGoal) {
  _solver.out() << “Improving solution” << endl;
  IloEnv env = _solver.getEnv();
  nhood.reset();
  IloGoal improve = IloSingleMove(env,
                                  _solution,
                                  nhood,
                                  IloImprove(env),
                                  subGoal);
  while (_solver.solve(improve)) {}
}

// Improve solution
void RoutingSolver::improve(IloGoal subGoal) {
  IloEnv env = _solver.getEnv();
  IloNHood nhood = IloTwoOpt(env)
                 + IloOrOpt(env)
                 + IloRelocate(env)
                 + IloCross(env)
                 + IloExchange(env);
  greedyImprove(nhood, subGoal);
}

// Display Dispatcher information
void RoutingSolver::printInformation() const {
  IloDispatcher dispatcher(_solver);
  _solver.printInformation();
  dispatcher.printInformation();
  _solver.out() << “===============” << endl
                << “Cost         : “ << dispatcher.getTotalCost() << endl
                << “Number of vehicles used : “
                << dispatcher.getNumberOfVehiclesUsed() << endl
                << “Solution     : “ << endl
                << dispatcher << endl;

  IlcScheduler sched(_solver);
  for (IlcActivityIterator iter(sched); iter.ok(); ++iter) {
    _solver.out() << *iter << endl;

  }
}

// Solving
void RoutingSolver::solve() {
  IloDispatcher dispatcher(_solver);
  IloEnv env = _solver.getEnv();

  // Subgoals
  IloGoal instantiateCost = IloDichotomize(env,
                                           dispatcher.getCostVar(),
                                           IloFalse);
  IloGoal schedGoal = IloSetTimesForward(env);
  IloGoal subGoal = schedGoal && instantiateCost;
  IloGoal restoreSolution = IloRestoreSolution(env, _solution) && subGoal;
  IloGoal goal = IloSavingsGenerate(env, schedGoal) && instantiateCost;

  // Solving
  if (findFirstSolution(goal)) {
    improve(subGoal);
    _solver.solve(restoreSolution);
  }
}

///////////////////////////////////////////////////////////////////////////////

int main(int argc, char * argv[]) {
  IloEnv env;
  try {
    IloInt schedGranularity = 100;
    RoutingModel mdl(env, argc, argv, schedGranularity);
    RoutingSolver solver(mdl);
    solver.solve();
    solver.printInformation();
  } catch(IloException& ex) {
    cerr << “Error: “ << ex << endl;
  }
  env.end();
  return 0;
}
```

Complete Output

```
/**
1641.01
Improving solution
Number of fails               : 0
Number of choice points       : 1068
Number of variables           : 2215
Number of constraints         : 564
Reversible stack (bytes)      : 385944
Solver heap (bytes)           : 1041460
Solver global heap (bytes)    : 184996
And stack (bytes)             : 20124
Or stack (bytes)              : 44244
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 9148
Total memory used (bytes)     : 1689960
Running time since creation   : 0.023437
Number of nodes               : 51
Number of visits              : 70
Number of vehicles           : 15
Number of dimensions         : 3
Number of accepted moves     : 11
===============
Cost         : 1432.17
Number of vehicles used : 6
Solution     :
Unperformed visits : None
vehicle1 :
 -> depot Weight[0] Time[0] Distance[0.. Inf) -> pickup11 Weight[0..171]
Time[0] Distance[0.. Inf) -> pickup19 Weight[12..183] Time[10] Distance[0..
Inf) -> delivery19 Weight[29..200] Time[52.0156..59.9289] Distance[0.. Inf) ->
delivery11 Weight[12..183] Time[69.0867..77] Distance[0.. Inf) -> depot
Weight[0..171] Time[112.628..230] Distance[0.. Inf)
vehicle2 :
 -> depot Weight[0] Time[0] Distance[0.. Inf) -> pickup3 Weight[0..143] Time[0]
Distance[0.. Inf) -> pickup9 Weight[13..156] Time[10] Distance[0.. Inf) ->
pickup12 Weight[29..172] Time[20] Distance[0.. Inf) -> pickup20 Weight[48..191]
Time[40] Distance[0.. Inf) -> delivery20 Weight[57..200] Time[81.6228..85.8197]
Distance[0.. Inf) -> delivery9 Weight[48..191] Time[102.803..107] Distance[0..
Inf) -> delivery3 Weight[32..175] Time[127.803..183.82] Distance[0.. Inf) ->
delivery12 Weight[19..162] Time[148.983..205] Distance[0.. Inf) -> depot
Weight[0..143] Time[173.983..230] Distance[0.. Inf)
vehicle3 :
 -> depot Weight[0] Time[0] Distance[0.. Inf) -> pickup7 Weight[0..146] Time[0]
Distance[0.. Inf) -> pickup8 Weight[5..151] Time[10] Distance[0.. Inf) ->
pickup5 Weight[14..160] Time[20] Distance[0.. Inf) -> pickup18 Weight[40..186]
Time[40] Distance[0.. Inf) -> pickup17 Weight[52..198] Time[50] Distance[0..
Inf) -> delivery7 Weight[54..200] Time[81.2132..82.7934] Distance[0.. Inf) ->
delivery8 Weight[49..195] Time[103.42..105] Distance[0.. Inf) -> delivery17
Weight[40..186] Time[127.348..162.82] Distance[0.. Inf) -> delivery5
Weight[38..184] Time[147.348..182.82] Distance[0.. Inf) -> delivery18
Weight[12..158] Time[168.528..204] Distance[0.. Inf) -> depot Weight[0..146]
Time[194.34..230] Distance[0.. Inf)
vehicle4 :
 -> depot Weight[0] Time[0..50] Distance[0.. Inf) -> pickup1 Weight[0..174]
Time[50] Distance[0.. Inf) -> pickup10 Weight[10..184] Time[60] Distance[0..
Inf) -> delivery10 Weight[26..200] Time[95.4951..178.444] Distance[0.. Inf) ->
delivery1 Weight[10..184] Time[121.051..204] Distance[0.. Inf) -> depot
Weight[0..174] Time[146.283..230] Distance[0.. Inf)
vehicle5 :
 -> depot Weight[0] Time[0..20] Distance[0.. Inf) -> pickup15 Weight[0..174]
Time[20] Distance[0.. Inf) -> delivery15 Weight[8..182] Time[70.4138..70.4162]
Distance[0.. Inf) -> pickup4 Weight[0..174] Time[110.83..110.83] Distance[0..
Inf) -> pickup2 Weight[19..193] Time[120.83..120.83] Distance[0.. Inf) ->
delivery4 Weight[26..200] Time[155.83..159] Distance[0.. Inf) -> delivery2
Weight[7..181] Time[186.054..202] Distance[0.. Inf) -> depot Weight[0..174]
Time[214.054..230] Distance[0.. Inf)
vehicle6 :
 -> depot Weight[0] Time[0..40] Distance[0.. Inf) -> pickup6 Weight[0..135]
Time[40] Distance[0.. Inf) -> pickup16 Weight[3..138] Time[50] Distance[0..
Inf) -> pickup14 Weight[22..157] Time[60] Distance[0.. Inf) -> pickup13
Weight[42..177] Time[70] Distance[0.. Inf) -> delivery16 Weight[65..200]
Time[109.155..116.606] Distance[0.. Inf) -> delivery14 Weight[46..181]
Time[130.335..137.787] Distance[0.. Inf) -> delivery13 Weight[26..161]
Time[161.548..169] Distance[0.. Inf) -> delivery6 Weight[3..138]
Time[178.619..208] Distance[0.. Inf) -> depot Weight[0..135] Time[199.8..230]
Distance[0.. Inf)
```

```
vehicle7 : Unused
vehicle8 : Unused
vehicle9 : Unused
vehicle10 : Unused
vehicle11 : Unused
vehicle12 : Unused
vehicle13 : Unused
vehicle14 : Unused
vehicle15 : Unused

pickup20 [4000 -- (1000) 1000 --> 5000]
pickup19 [1000 -- (1000) 1000 --> 2000]
pickup18 [4000 -- (1000) 1000 --> 5000]
pickup17 [5000 -- (1000..1158) 1000..1158 --> 6000..6158]
pickup16 [5000 -- (1000) 1000 --> 6000]
pickup15 [2000 -- (1000) 2000 --> 4000]
pickup14 [6000 -- (1000) 1000 --> 7000]
pickup13 [7000 -- (1000..1745) 1000..1745 --> 8000..8745]
pickup12 [2000 -- (1000) 2000 --> 4000]
pickup11 [0 -- (1000) 1000 --> 1000]
pickup10 [6000 -- (1000..9294) 1000..9294 --> 7000..15294]
pickup9 [1000 -- (1000) 1000 --> 2000]
pickup8 [1000 -- (1000) 1000 --> 2000]
pickup7 [0 -- (1000) 1000 --> 1000]
pickup6 [4000 -- (1000) 1000 --> 5000]
pickup5 [2000 -- (1000) 2000 --> 4000]
pickup4 [11083 -- (1000) 1000 --> 12083]
pickup3 [0 -- (1000) 1000 --> 1000]
pickup2 [12083 -- (1000..1317) 1000..1317 --> 13083..13400]
pickup1 [5000 -- (1000) 1000 --> 6000]

*/
```

Part III

Field Service Solutions

This part consists of the following lessons:

- Chapter 13, *Dispatching Technicians*
- Chapter 14, *Dispatching Technicians II*
- Chapter 15, *CARP: Visiting Arcs Using Multiple Vehicles*

C H A P T E R

13 Dispatching Technicians

In this lesson, you will learn how to:

- ◆ model a technician routing problem
- ◆ match differing technician skill levels with multiple customer skill requirements
- ◆ use intrinsic dimensions to model a skill cost.

Describe

A typical real-world routing problem involves dispatching technicians, such as repair persons or equipment installers, to customer sites. This type of problem can be viewed as a type of Vehicle Routing Problem, except that technicians must be routed along with the vehicles. Customer visits can also require a quantity of goods to be picked up or delivered.

Technicians may have different skill levels, and customer visits require a particular skill set to complete a job. Therefore you need to model these skills and apply constraints and costs to them. One way to model this is with a fixed charge for each technician based on skill level, regardless of the job skill required at the actual customer site. As long as the technician has at least the appropriate level of skill, the visit can occur. However, this can lead to obvious inequities, if highly-trained technicians are performing less-demanding visits.

There is a different and often better way to model these costs that would maximize the quality of service. If a technician type is well suited to performing a certain visit, then the cost for that technician/visit pair should be low. If a technician type is not well suited to performing a visit, the cost for that technician/visit pair should be high.

In this example, five technicians with different skill levels perform customer visits with ten different skill requirements. The costs for a particular visit can vary significantly, depending on the technician skill level used.

Step 1 Describe the problem

The first step is to write a natural language description of the problem. A good way to start this process is to analyze the constraints and objectives.

What are the constraints in this problem?

- There are five technicians with differing skill levels, and numerous customer visits each with one of ten different skill requirements.
- There is one technician per vehicle.
- Each customer visit requires a technician, and any technician can visit any customer. However, the cost for each visit varies depending on both the technician skill level and the skill required at the visit.

The objective is to minimize the cost of supplying technicians to visit all customer sites.

Model

Once you have written a description of your problem, you can use Dispatcher classes to model it.

Step 2 Open the example file

Open the example file `YourDispatcherHome/examples/src/tutorial/technic2_partial.cpp` in your development environment.

As in the previous examples, you will use a `RoutingModel` class to call the functions that create the dimensions, nodes, vehicles (technicians), and visits.

Step 3 Declare the RoutingModel class

Add the following code after the comment `//Declare the RoutingModel class.`

```
class RoutingModel {
  IloEnv              _env;
  IloModel            _mdl;

  void createDimensions();
  void createIloNodes(char* nodeFileName);
  void createTechnicians(char* technicianFileName);
  void createVisits(char* visitsFileName);
  void setVisitsSkillPenalty(char * skillCostsFileName,
                             char * visitSkillsFileName,
                             char * techSkillsFileName);
```

The function `createDimensions` defines the `skillPenalty` dimension. The `createTechnicians` function is similar to the `createVehicles` function of previous lessons, in that it adds an `IloVehicle` to the model. The function `setVisitsSkillPenalty` calculates the cost of each visit and adds it to the model. The other functions are similar to their counterparts in the previous lessons.

Step 4 Create the dimensions

Add the following code after the comment `//Create the dimensions`.

```
void RoutingModel::createDimensions() {
  IloDimension1 skillPenalty(_env, IloFalse, "SkillPenalty");
  _mdl.add(skillPenalty);
  skillPenalty.setKey("SkillPenalty");
  IloDimension2 time(_env, IloEuclidean, "Time");
  time.setKey("Time");
  _mdl.add(time);
}
```

Instances of `IloDimension1` are intrinsic dimensions; this means that the value of the dimension depends only on the object itself (such as an object's weight), and not on any other factors. The dimension `skillPenalty` will be used to assign a cost to the visit of each vehicle, just as weight or capacity were used in previous lessons. The second parameter is set to `IloFalse` in order to speed up search; this is permissible as no constraints are posted on variables related to the `skillPenalty`. However, `skillPenalty` can still be used in the cost function.

The function `createTechnicians` adds vehicles named `technician` to the model.

Step 5 Create the vehicle/technicians

Add the following code after the comment `//Create the vehicle/technicians.`

```
    IloVehicle technician(first, last, name);
    technician.setCost(time, 1.0);
    technician.setCost(skillPenalty, 1.0);
    technician.setKey(name);
    _mdl.add(technician);
```

As there is only one technician per vehicle, it is convenient to represent the vehicle with the name `technician`. This section of `createTechnicians` sets the cost for the vehicles based on the dimensions of `time` and `skillPenalty`.

The first section of `createTechnicians` is similar to what you have worked with in the `createVehicles` function of previous lessons, with the addition of some code to ensure that a `skillPenalty` of zero is associated with the first and last visits to the depot.

```
void RoutingModel::createTechnicians(char* techFileName) {
  IloDimension2 time = IloDimension2::Find(_env, "Time");
  IloDimension1 skillPenalty = IloDimension1::Find(_env, "SkillPenalty");
  IloCsvReader csvTechReader(_env, techFileName);
  IloCsvReader::LineIterator it(csvTechReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char * namefirst = line.getStringByHeader("first");
    char * namelast = line.getStringByHeader("last");
    char * name = line.getStringByHeader("name");
    IloNum openTime = line.getFloatByHeader("open");
    IloNum closeTime = line.getFloatByHeader("close");
    IloNode node1 = IloNode::Find(_env, namefirst);
    IloNode node2 = IloNode::Find(_env, namelast);

    IloVisit first(node1, "depot");
    _mdl.add(first.getTransitVar(skillPenalty) == 0);
    _mdl.add(first.getCumulVar(time) >= openTime);

    IloVisit last(node2, "depot");
    _mdl.add(last.getCumulVar(time) <= closeTime);
    _mdl.add(last.getTransitVar(skillPenalty) == 0);
```

Step 6 Create the setVisitsSkillPenalty function

Add the following code after the comment `//Create the setVisitsSkillPenalty function`.

```
void RoutingModel::setVisitsSkillPenalty(char * skillCostsFileName,
                                         char * visitSkillsFileName,
                                         char * techSkillsFileName) {
  IloDimension1 skillPenalty = IloDimension1::Find(_env, "SkillPenalty");
  IloCsvReader csvVisitSkillsReader(_env, visitSkillsFileName);
  IloCsvReader csvTechSkillsReader(_env, techSkillsFileName);
  IloCsvReader csvSkillCostsReader(_env, skillCostsFileName);
  IloInt nbOfTech = csvTechSkillsReader.getNumberOfItems();
  IloVehicleArray techs(_env, nbOfTech);
  IloInt i = 0;
  IloCsvReader::LineIterator  it(csvTechSkillsReader);
  while(it.ok()){
    IloCsvLine line = *it;
    char * techName = line.getStringByHeader("name");
    IloVehicle technician = IloVehicle::Find(_env, techName);
    techs[i] = technician;
    i++;
    ++it;
  }
```

This section of the function creates the vehicle array `techs` with the vehicle instances `technician`. The `IloCsvReader` instance `csvVisitSkillsReader` reads the skills required at each customer visit; `csvTechSkillsReader` reads the skill level of each technician; and `csvSkillCostsReader` reads the cost for each technician skill type to perform a visit to a customer site.

The next section of `setVisitsSkillPenalty` creates an array of the cost for each visit, named `visitCost`. To create this array, the visit skill required for each visit (`visitSkillName`) is read from a csv file. The cost to make this visit with the various `techSkill` levels is read with `csvSkillCostsReader` as an `IloCsvLine` `costline`.

Then in `Iterator it3` the value in `costline` corresponding to the appropriate `techSkillName` is stored in the array `visitCost`.

```
IloCsvReader::LineIterator  it2(csvVisitSkillsReader);
while(it2.ok()){
  IloCsvLine line = *it2;
  char * visitName = line.getStringByHeader("name");
  IloVisit visit = IloVisit::Find(_env, visitName);
  char * visitSkillName = line.getStringByHeader("VisitSkillName");
  IloCsvLine costline = csvSkillCostsReader.getLineByKey(1, visitSkillName);
  IloNumArray visitCost(_env, nbOfTech);
  IloInt k = 0;
  IloCsvReader::LineIterator  it3(csvTechSkillsReader);
  while(it3.ok()){
    IloCsvLine line1 = *it3;
    char * techSkillName = line1.getStringByHeader("TechSkillName");
    visitCost[k] = costline.getFloatByHeader(techSkillName);
    k++;
    ++it3;
  }
```

Step 7 Add the skill cost

Add the following code after the comment `//Add the skill cost`.

```
  IloVehicleToNumFunction function(_env, techs, visitCost);
  IloNumVar skillCostVar (function(visit.getVehicleVar()));
  _mdl.add(visit.getTransitVar(skillPenalty) == skillCostVar);
```

In this section of `setVisitsSkillPenalty`, the `IloVehicleToNumFunction` constructor associates the values of the array `visitCost` to the vehicle/technicians of the array `techs`. Then `visit.getVehicleVar` returns the vehicle/technician for the visit, and the cost for that visit is applied to the `IloNumVar skillCostVar`. Finally, this is added to the model in the dimension `skillPenalty`.

Solve

The solution is computed, improved, and displayed by methods previously presented for the PDP problem.

Step 8 Compile and run the program

The solution improvement phase finds a solution using 5 vehicles with a cost of 1441.04 units:

```
/**
1619.09
Improving solution
Number of fails              : 0
Number of choice points      : 0
Number of variables          : 619
Number of constraints        : 232
Reversible stack (bytes)     : 76404
Solver heap (bytes)          : 402280
Solver global heap (bytes)   : 481592
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11160
Total memory used (bytes)    : 1039848
Elapsed time since creation  : 0
Number of nodes              : 21
Number of visits             : 30
Number of vehicles           : 5
Number of dimensions         : 2
Number of accepted moves     : 16
===============
Cost         : 1441.04
Number of vehicles used : 5
```

Review Exercises

For answers, see “Suggested Answers” on page 330.

1. What Dispatcher class can be used to associate `IloNumArray` values with an array of vehicles?
2. What is an *intrinsic* dimension?
3. What Dispatcher class is used to represent intrinsic dimensions?

Suggested Answers

Exercise 1

What Dispatcher class can be used to associate `IloNumArray` values with an array of vehicles?

Suggested Answer

`IloVehicleToNumFunction`.

Exercise 2

What is an *intrinsic* dimension?

Suggested Answer

An intrinsic dimension is a dimension that depends only on the object itself, not on any other factors.

Exercise 3

What Dispatcher class is used to represent intrinsic dimensions?

Suggested Answer

`IloDimension1`.

Complete Program

The complete program follows. You can also view it online in the file `YourDispatcherHome/examples/src/technic2_complete.cpp`.

```
// -------------------------------------------------------------- -*- C++ -*-
// File: examples/src/technic2.cpp
// --------------------------------------------------------------------------


#include <ildispat/ilodispatcher.h>

ILOSTLBEGIN

///////////////////////////////////////////////////////////////////////////////
// Modeling
class RoutingModel {
  IloEnv              _env;
  IloModel            _mdl;

  void createDimensions();
  void createIloNodes(char* nodeFileName);
  void createTechnicians(char* technicianFileName);
  void createVisits(char* visitsFileName);
  void setVisitsSkillPenalty(char * skillCostsFileName,
                             char * visitSkillsFileName,
                             char * techSkillsFileName);

public:
  RoutingModel(IloEnv env,
               int argc,
               char* argv[]);
  ~RoutingModel() {}
  IloEnv getEnv() const { return _env; }
  IloModel getModel() const { return _mdl; }
};

RoutingModel::RoutingModel(IloEnv env,
                                       int argc,
                                       char * argv[])
  : _env(env), _mdl(env){

  createDimensions();

  char* nodeFileName;
  if(argc < 4)
    nodeFileName = (char *) "../../../examples/data/technic2/nodes.csv";
  else
    nodeFileName = argv[3];
  createIloNodes(nodeFileName);

  char* technicianFileName;
  if (argc < 2)
    technicianFileName = (char *) "../../../examples/data/technic2/
technicians.csv";
  else
    technicianFileName = argv[1];
  createTechnicians(technicianFileName);

  char* visitsFileName;
  if (argc < 3)
    visitsFileName =
      (char*) "../../../examples/data/technic2/visits.csv";
  else
    visitsFileName = argv[2];
  createVisits(visitsFileName);

  //set the correct skillPenalty for visits
  char * visitSkillsFileName = (char *) "../../../examples/data/technic2/
visitSkills.csv";
  char * techSkillsFileName = (char *) "../../../examples/data/technic2/
techSkills.csv";
  char * skillCostsFileName = (char *) "../../../examples/data/technic2/
skillCosts.csv";

  if(argc >= 4)
    visitSkillsFileName = argv[4];
  if(argc >= 5) {
    visitSkillsFileName = argv[4];
    techSkillsFileName  = argv[5];
  }
  if(argc >= 6) {
    visitSkillsFileName = argv[4];
    techSkillsFileName  = argv[5];
    skillCostsFileName  = argv[6];
  }
  setVisitsSkillPenalty(skillCostsFileName, visitSkillsFileName,
techSkillsFileName);

}

// Add dimensions
void RoutingModel::createDimensions() {
  IloDimension1 skillPenalty(_env, IloFalse, “SkillPenalty”);
  _mdl.add(skillPenalty);
  skillPenalty.setKey(“SkillPenalty”);
  IloDimension2 time(_env, IloEuclidean, “Time”);
  time.setKey(“Time”);
  _mdl.add(time);
}

// Create IloNodes
void RoutingModel::createIloNodes(char* nodeFileName) {
  IloCsvReader csvNodeReader(_env, nodeFileName);
  IloCsvReader::LineIterator  it(csvNodeReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char* name = line.getStringByHeader(“name”);
    IloNode node(_env,
                 line.getFloatByHeader(“x”),
                 line.getFloatByHeader(“y”),
                 0,
                 name);
    node.setKey(name);
    ++it;
  }
  csvNodeReader.end();
}

// Create technicians
void RoutingModel::createTechnicians(char* techFileName) {
  IloDimension2 time = IloDimension2::Find(_env, “Time”);
  IloDimension1 skillPenalty = IloDimension1::Find(_env, “SkillPenalty”);
  IloCsvReader csvTechReader(_env, techFileName);
  IloCsvReader::LineIterator it(csvTechReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char * namefirst = line.getStringByHeader(“first”);
    char * namelast = line.getStringByHeader(“last”);
    char * name = line.getStringByHeader(“name”);
    IloNum openTime = line.getFloatByHeader(“open”);
    IloNum closeTime = line.getFloatByHeader(“close”);
    IloNode node1 = IloNode::Find(_env, namefirst);
    IloNode node2 = IloNode::Find(_env, namelast);

    IloVisit first(node1, “depot”);
    _mdl.add(first.getTransitVar(skillPenalty) == 0);
    _mdl.add(first.getCumulVar(time) >= openTime);

    IloVisit last(node2, “depot”);
    _mdl.add(last.getCumulVar(time) <= closeTime);
    _mdl.add(last.getTransitVar(skillPenalty) == 0);

    IloVehicle technician(first, last, name);
    technician.setCost(time, 1.0);
    technician.setCost(skillPenalty, 1.0);
    technician.setKey(name);
    _mdl.add(technician);

    ++it;
  }
  csvTechReader.end();
}

// Create visits
void RoutingModel::createVisits(char* visitsFileName) {
  IloDimension2 time = IloDimension2::Find(_env, “Time”);
  IloCsvReader csvVisitReader(_env, visitsFileName);
  IloCsvReader::LineIterator  it(csvVisitReader);
  while(it.ok()){
    IloCsvLine line = *it;
    //read visit data from files
    char * visitName = line.getStringByHeader(“name”);
    char * nodeName = line.getStringByHeader(“node”);
    IloNum minTime  = line.getFloatByHeader(“minTime”);
    IloNum maxTime  = line.getFloatByHeader(“maxTime”);
    IloNum dropTime = line.getFloatByHeader(“dropTime”);
    IloNode node = IloNode::Find(_env, nodeName);
    IloVisit visit(node, visitName);
    _mdl.add(visit.getDelayVar(time) == dropTime);
    _mdl.add(minTime <= visit.getCumulVar(time) <= maxTime);
    visit.setKey(visitName);
    _mdl.add(visit);
    ++it;
  }
  csvVisitReader.end();
}

//set the correct skillPenalty for visits
void RoutingModel::setVisitsSkillPenalty(char * skillCostsFileName,
                                         char * visitSkillsFileName,
                                         char * techSkillsFileName) {
  IloDimension1 skillPenalty = IloDimension1::Find(_env, “SkillPenalty”);
  IloCsvReader csvVisitSkillsReader(_env, visitSkillsFileName);
  IloCsvReader csvTechSkillsReader(_env, techSkillsFileName);
  IloCsvReader csvSkillCostsReader(_env, skillCostsFileName);
  IloInt nbOfTech = csvTechSkillsReader.getNumberOfItems();
  IloVehicleArray techs(_env, nbOfTech);
  IloInt i = 0;
  IloCsvReader::LineIterator  it(csvTechSkillsReader);
  while(it.ok()){
    IloCsvLine line = *it;
    char * techName = line.getStringByHeader(“name”);
    IloVehicle technician = IloVehicle::Find(_env, techName);
    techs[i] = technician;
    i++;
    ++it;
  }

  IloCsvReader::LineIterator  it2(csvVisitSkillsReader);
  while(it2.ok()){
    IloCsvLine line = *it2;
    char * visitName = line.getStringByHeader("name");
    IloVisit visit = IloVisit::Find(_env, visitName);
    char * visitSkillName = line.getStringByHeader("VisitSkillName");
    IloCsvLine costline = csvSkillCostsReader.getLineByKey(1, visitSkillName);
    IloNumArray visitCost(_env, nbOfTech);
    IloInt k = 0;
    IloCsvReader::LineIterator  it3(csvTechSkillsReader);
    while(it3.ok()){
      IloCsvLine line1 = *it3;
      char * techSkillName = line1.getStringByHeader("TechSkillName");
      visitCost[k] = costline.getFloatByHeader(techSkillName);
      k++;
      ++it3;
    }

    IloVehicleToNumFunction function(_env, techs, visitCost);
    IloNumVar skillCostVar (function(visit.getVehicleVar()));
    _mdl.add(visit.getTransitVar(skillPenalty) == skillCostVar);

    ++it2;
  }
  csvVisitSkillsReader.end();
  csvTechSkillsReader.end();
  csvSkillCostsReader.end();

}

///////////////////////////////////////////////////////////////////////////////
// Solving
class RoutingSolver {
  IloModel            _mdl;
  IloSolver           _solver;
  IloRoutingSolution  _solution;

  IloBool findFirstSolution(IloGoal goal);
  void greedyImprove(IloNHood nhood, IloGoal subGoal);
  void improve(IloGoal subgoal);
public:
  RoutingSolver(RoutingModel mdl);
  ~RoutingSolver() {}
  IloRoutingSolution getSolution() const { return _solution; }
  void printInformation() const;
  void solve();
};

RoutingSolver::RoutingSolver(RoutingModel mdl)
  : _mdl(mdl.getModel()), _solver(mdl.getModel()), _solution(mdl.getModel()) {}

// Solving : find first solution
IloBool RoutingSolver::findFirstSolution(IloGoal goal) {
  if (!_solver.solve(goal)) {
    _solver.error() << “Infeasible Routing Plan” << endl;
    return IloFalse;
  }
  IloDispatcher dispatcher(_solver);
  _solver.out() << dispatcher.getTotalCost() << endl;
  _solution.store(_solver);
  return IloTrue;
}

// Improve solution using nhood
void RoutingSolver::greedyImprove(IloNHood nhood, IloGoal subGoal) {
  _solver.out() << “Improving solution” << endl;
  IloEnv env = _solver.getEnv();
  nhood.reset();
  IloGoal improve = IloSingleMove(env,
                                  _solution,
                                  nhood,
                                  IloImprove(env),
                                  subGoal);
  while (_solver.solve(improve)) {}
}

// Improve solution
void RoutingSolver::improve(IloGoal subGoal) {
  IloEnv env = _solver.getEnv();
  IloNHood nhood = IloTwoOpt(env)
                 + IloOrOpt(env)
                 + IloRelocate(env)
                 + IloCross(env)
                 + IloExchange(env);
  greedyImprove(nhood, subGoal);
}

// Display Dispatcher information
void RoutingSolver::printInformation() const {
  IloDispatcher dispatcher(_solver);
  _solver.printInformation();
  dispatcher.printInformation();
  _solver.out() << “===============” << endl
                << “Cost         : “ << dispatcher.getTotalCost() << endl
                << “Number of vehicles used : “
                << dispatcher.getNumberOfVehiclesUsed() << endl
                << “Solution     : “ << endl
                << dispatcher << endl;
}

// Solving
void RoutingSolver::solve() {
  IloDispatcher dispatcher(_solver);
  IloEnv env = _solver.getEnv();
  // Subgoal
  IloGoal instantiateCost = IloDichotomize(env,
                                           dispatcher.getCostVar(),
                                           IloFalse);
  IloGoal restoreSolution = IloRestoreSolution(env, _solution);
  IloGoal goal = IloInsertionGenerate(env, instantiateCost);

  // Solving
  if (findFirstSolution(goal)) {
    improve(instantiateCost);
    _solver.solve(restoreSolution);
  }
}

////////////////////////////////////////////////////////////////////////////////

int main(int argc, char * argv[]) {
  IloEnv env;
  try {
    RoutingModel mdl(env, argc, argv);
    RoutingSolver solver(mdl);
    solver.solve();
    solver.printInformation();
  } catch(IloException& ex) {
    cerr << “Error: “ << ex << endl;
  }
  env.end();
  return 0;
}
```

Complete Output

```
/**
1619.09
Improving solution
Number of fails               : 0
Number of choice points       : 0
Number of variables           : 619
Number of constraints         : 232
Reversible stack (bytes)      : 76404
Solver heap (bytes)           : 402280
Solver global heap (bytes)    : 481592
And stack (bytes)             : 20124
Or stack (bytes)              : 44244
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 11160
Total memory used (bytes)     : 1039848
Elapsed time since creation   : 0
Number of nodes               : 21
Number of visits              : 30
Number of vehicles            : 5
Number of dimensions          : 2
Number of accepted moves      : 16
===============
Cost          : 1441.04
Number of vehicles used : 5
Solution      :
Unperformed visits : None
tech1 :
 -> depot SkillPenalty[0..Inf) Time[0..157.82] -> visit13 SkillPenalty[0..Inf)
Time[159..169] -> depot SkillPenalty[0..Inf) Time[180.18..230]
tech2 :
 -> depot SkillPenalty[0..Inf) Time[0..13.173] -> visit14 SkillPenalty[0..Inf)
Time[32.0156..45.1886] -> visit15 SkillPenalty[0..Inf) Time[61..71] -> depot
```

```
SkillPenalty[0..Inf) Time[101.414..230]
tech3 :
 -> depot SkillPenalty[0..Inf) Time[0..7.46725] -> visit6 SkillPenalty[0..Inf)
Time[11.1803..18.6476] -> visit5 SkillPenalty[0..Inf) Time[31.1803..38.6476] ->
visit16 SkillPenalty[0..Inf) Time[52.3607..59.8279] -> visit17
SkillPenalty[0..Inf) Time[73.541..81.0083] -> visit7 SkillPenalty[0..Inf)
Time[108.541..116.008] -> visit19 SkillPenalty[0..Inf) Time[129.721..137.189] -
> visit10 SkillPenalty[0..Inf) Time[154.721..162.189] -> visit20
SkillPenalty[0..Inf) Time[180.533..188] -> depot SkillPenalty[0..Inf)
Time[222.156..230]
tech4 :
 -> depot SkillPenalty[0..Inf) Time[0..32.9017] -> visit1 SkillPenalty[0..Inf)
Time[15.2315..48.1333] -> visit3 SkillPenalty[0..Inf) Time[39.7918..72.6935] ->
visit9 SkillPenalty[0..Inf) Time[97..97.6935] -> visit12 SkillPenalty[0..Inf)
Time[132.495..133.189] -> visit4 SkillPenalty[0..Inf) Time[158.306..159] ->
visit2 SkillPenalty[0..Inf) Time[188.53..202] -> depot SkillPenalty[0..Inf)
Time[216.53..230]
tech5 :
 -> depot SkillPenalty[0..Inf) Time[0..37.2929] -> visit11 SkillPenalty[0..Inf)
Time[67..70.8339] -> visit8 SkillPenalty[0..Inf) Time[101.166..105] -> visit18
SkillPenalty[0..Inf) Time[121.606..204] -> depot SkillPenalty[0..Inf)
Time[147.418..230]
*/
```

C H A P T E R

14

Dispatching Technicians II

In this lesson, you will learn how to:

- model the vehicle-visit compatibility constraint
- model various technician job skill levels
- use the Dispatcher class `IloVisitVehicleCompat`

Describe

In Lesson 13, *Dispatching Technicians*, the concept of routing skilled technicians to customer sites was introduced. In that lesson, any technician could visit any customer site to perform any job, but the cost would vary depending on the level of the skill delivered and the level of the skill needed at the customer site.

This lesson examines the case where each customer visit requires a specific set of three skills (the job or visit profile) to complete the visit. However, not all technicians possess the appropriate skills. The problem therefore consists of routing the technicians to jobs that they can be accomplish. Furthermore, most technicians can perform various jobs at customer sites, but with different levels of effectiveness.

Step 1 Describe the problem

The first step is to write a natural language description of the problem. A good way to start this process is to analyze the constraints and objectives.

What are the constraints in this problem?

- ◆ Customer visits require specific skills from a pool of technicians with differing ability levels.
- ◆ There are ten different job profiles (`VisitProfileID`) required at the customer sites, and there are five different technicians skill profiles (`TechProfileID`) with which to perform the customer jobs.
- ◆ The ability level of each technician at a particular skill (`skillnLevel`) varies from zero (not able to perform that skill) to five (“expert”). There is a constraint of a minimum ability level per skill. For example, a job may require level 3 at skill 1, level 4 at skill 2, and level 5 at skill 3.
- ◆ There are time constraints on some visits and the depot, as well as a service time.

The objective is to minimize the cost of routing the technicians to all customer sites.

Model

Once you have written a description of your problem, you can use Dispatcher classes to model it.

Step 2 Open the example file

Open the example file `YourDispatcherHome/examples/src/tutorial/technic1_partial.cpp` in your development environment.

This program starts with a class that models both skill requirements for jobs and skills provided by technicians.

Step 3 Declare the SkillProfile class

Add the following code after the comment `//Declare the SkillProfile class.`

```
class SkillProfile {
  IloEnv _env;
  IloInt _nbOfSkills;
  IloInt* _skillLevels;
public:
  SkillProfile(IloEnv env, IloInt nbOfSkills):
    _env(env),
    _nbOfSkills(nbOfSkills),
    _skillLevels(0)
  {
    _skillLevels = new (env) IloInt [ nbOfSkills ];
    for (IloInt s=0; s< _nbOfSkills; ++s) _skillLevels[s] = 0;
  }
  void setSkillLevel(IloInt skillIndex, IloInt level=1) {
    _skillLevels[ skillIndex ] = level; }

  /**
   * Returns true if <code>other</code> is comparable with this
   * and is greater.
   */
  IloBool isGreaterThanOrEqual(const SkillProfile* other);
```

The `SkillProfile` class maps a `level` for each technician skill. The function `SkillProfile::isGreaterThanOrEqual` follows, and is used to ensure that the skill level meets the constraint of the job profile.

```
IloBool SkillProfile::isGreaterThanOrEqual(const SkillProfile* other) {
  for(IloInt index = 0;
      index < _nbOfSkills && other->_nbOfSkills; ++index) {
    if ( _skillLevels[index] < other->_skillLevels[index] ) {
       return IloFalse;
    }
  }
  return IloTrue;
}
```

Step 4 Declare the RoutingModel class

Add the following code after the comment `//Declare the RoutingModel class`.

```
  void createDimensions();
  void createIloNodes(const char* nodePath);
  void createSkillProfiles(const char* profilePath);
  void setTechnicianSkillProfiles(const char* techProfilePath);
  void setVisitSkillProfiles(const char* visitProfilePath);
  void createTechnicians(const char* technicianPath);
  void createVisits(const char* visitsPath);

  void postCompatibility();
```

The only dimension defined in `createDimensions` is `time`, and `createIloNodes` is similar to previous lessons. `createTechnicians` is similar to the same function in Chapter 13, *Dispatching Technicians*. The other functions in `RoutingModel` are new, and these are associated with the technician skills, levels, and visit requirements.

The following is the first section of `RoutingModel` and is provided for you:

```
class RoutingModel {
  IloEnv              _env;
  IloModel            _mdl;
  IloArray<SkillProfile*> _profiles;
  IloInt _nbOfSkills;

  const char* _nodePath;
  const char* _technicianPath;
  const char* _visitPath;
  const char* _profilePath;
  const char* _techProfilePath;  // table binding profiles to tecnicians
  const char* _visitProfilePath; // table binding profiles to visits
```

Step 5 Create the getSkillProfile function

Add the following code after the comment `//Create the getSkillProfile function`.

```
  SkillProfile* getSkillProfile(IloInt profileId) {
    if ( _profiles.getSize() <= profileId ) {
      for (IloInt p= _profiles.getSize(); p <= profileId; ++p) {
        _profiles.add( (SkillProfile*) 0);
      }
    }
    SkillProfile* profile = _profiles[ profileId ] ;
    if ( 0 == profile ) {
      profile = new (_env) SkillProfile(_env, _nbOfSkills);
      _profiles[ profileId ] = profile;
    }
    return profile;
  }
```

Skill profiles, both for jobs and technicians, are stored in an array and identified by a unique ID. Retrieving the profile from its ID is performed by a lazy accessor technique. Missing array slots are filled with zeros; then, if a returned profile is null, a new profile is created, stored, and returned.

The final section of `RoutingModel` follows:

```
public:
  RoutingModel(IloEnv env);
  ~RoutingModel() {}

  void parse(int argc, char** argv);
  void createModel();

  IloEnv getEnv() const { return _env; }
  IloModel getModel() const { return _mdl; }
};
```

Step 6 Create the parse function

Add the following code after the comment `//Create the parse function.`

```
void RoutingModel::parse(int argc, char** argv) {
  if ( argc > 1 ) _visitPath        = argv[1];
  if ( argc > 2 ) _technicianPath   = argv[2];
  if ( argc > 3 ) _visitProfilePath = argv[3];
  if ( argc > 4 ) _techProfilePath  = argv[4];
  if ( argc > 5 ) _profilePath      = argv[5];
  if ( argc > 6 ) _nodePath         = argv[6];
}
```

`_visitPath` accesses the data for the visit: the time window during which the visit can occur, and the service time. The technicians are identified through `_technicianPath`. `_visitProfilePath` accesses the map of visits to the visit skill profile IDs. `_techProfilePath` accesses the map of technicians to the technician skill profile IDs, and `_profilePath` accesses the technician skill ability levels and the skill ability levels required at the customer visits.

```
RoutingModel::RoutingModel(IloEnv env)
  : _env(env),
  _mdl(env),
  _profiles(env),
  _nbOfSkills(3),
  _nodePath("../../../examples/data/technic1/nodes.csv"),
  _technicianPath("../../../examples/data/technic1/technicians.csv"),
  _visitPath("../../../examples/data/technic1/visits.csv"),
  _profilePath("../../../examples/data/technic1/SkillProfiles.csv"),
  _techProfilePath("../../../examples/data/technic1/techSkills.csv"),
  _visitProfilePath("../../../examples/data/technic1/visitSkills.csv")
{ }
```

Step 7 Add the vehicle/technician

Add the following code after the comment `//Add the vehicle/technician`.

```
    IloVehicle technician(first, last, name);
    technician.setCost(time, 1.0);
    technician.setKey(name);
    _mdl.add(technician);
```

This code is a section of `createTechnicians`, and adds the vehicle/technician with the sole cost factor of `time`. The first section of the function follows.

```
void RoutingModel::createTechnicians(const char* techPath) {
  IloDimension2 time = IloDimension2::Find(_env, "Time");
  IloCsvReader csvTechReader(_env, techPath);
  char namebuf[128];
  IloCsvReader::LineIterator it(csvTechReader);
  for( ; it.ok(); ++it) {
    IloCsvLine line = *it;
    char * namefirst = line.getStringByHeader("first");
    char * namelast = line.getStringByHeader("last");
    char * name = line.getStringByHeader("name");
    IloNum openTime = line.getFloatByHeader("open");
    IloNum closeTime = line.getFloatByHeader("close");
    IloNode node1 = IloNode::Find(_env, namefirst);
    IloNode node2 = IloNode::Find(_env, namelast);

    sprintf(namebuf, "start_%s", name);
    IloVisit first(node1, namebuf);
    _mdl.add( first.getCumulVar(time) >= openTime );

    sprintf(namebuf, "end_%s", name);
    IloVisit last(node2, namebuf);
    _mdl.add( last.getCumulVar(time) <= closeTime );
```

Step 8 Add the PenaltyCost

Add the following code after the comment `//Add the PenaltyCost`.

```
    _mdl.add( visit.getDelayVar(time) == serviceTime );
    _mdl.add( minTime <= visit.getCumulVar(time) <= maxTime);
    visit.setPenaltyCost(10000);
    visit.setKey(visitName);
    _mdl.add(visit);
```

`minTime` and `maxTime` are the time windows for a paticular visit. The member function `IloVisit::setPenaltyCost` is used to set the cost of not performing a visit. By default, the penalty cost is `IloInfinity`, which means that visits must be performed in a solution if a value is not set with this member function.

The following code is the first section of `createVisits`.

```
void RoutingModel::createVisits(const char* visitsPath) {
  IloDimension2 time = IloDimension2::Find(_env, "Time");
  IloCsvReader csvVisitReader(_env, visitsPath);
  IloCsvReader::LineIterator  it(csvVisitReader);
  for( ;it.ok(); ++it) {
    IloCsvLine line = *it;
    //read visit data from files
    char * visitName = line.getStringByHeader("name");
    char * nodeName = line.getStringByHeader("node");
    IloNum minTime  = line.getFloatByHeader("minTime");
    IloNum maxTime  = line.getFloatByHeader("maxTime");
    IloNum serviceTime = line.getFloatByHeader("dropTime");
    IloNode node = IloNode::Find(_env, nodeName);
    IloVisit visit(node, visitName);
```

Step 9 Define the createSkillProfiles function

Add the following code after the comment
`//Define the createSkillProfiles function`.

```
void RoutingModel::createSkillProfiles(const char* profilePath) {
  IloCsvReader skillProfileReader(_env, profilePath);
  for( IloCsvReader::LineIterator it(skillProfileReader); it.ok(); ++it) {
    IloCsvLine line = *it;
    IloInt profileId = line.getIntByHeader("id");
    IloInt skill1Level = line.getIntByHeader("Skill1Level");
    IloInt skill2Level = line.getIntByHeader("Skill2Level");
    IloInt skill3Level = line.getIntByHeader("Skill3Level");

    SkillProfile* profile = getSkillProfile(profileId);
    assert( 0 != profile );

    profile->setSkillLevel(0, skill1Level);
    profile->setSkillLevel(1, skill2Level);
    profile->setSkillLevel(2, skill3Level);
  }
  skillProfileReader.end();
}
```

This function reads the csv file and creates the skill profiles of the technicians and visits; the levels of each skill that are available from the technician or required at the customer visit.

Step 10 Define the setTechnicianSkillProfiles function

Add the following code after the comment
`//Define the setTechnicianSkillProfiles function`.

```
void RoutingModel::setTechnicianSkillProfiles(const char* techProfilePath) {
  // now read the techProfilePath table.
  IloCsvReader techProfileReader(_env, techProfilePath);
  for (IloCsvReader::LineIterator tpit(techProfileReader); tpit.ok(); ++tpit) {
    IloCsvLine line = *tpit;
    const char* techName = line.getStringByHeader("name");
    IloInt profileId = line.getIntByHeader("TechProfileId");
    IloVehicle tech = IloVehicle::Find(_env, techName);
    SkillProfile* profile = getSkillProfile(profileId);
    if ( 0 != tech.getImpl() && 0 != profile ) {
      tech.setObject( (IloAny)profile );
    }
  }
  techProfileReader.end();
}
```

This function implements the mapping between the technician and the skill profile. The profile itself is stored in the object field of the `IloVehicle` (for technicians) or `IloVisit` (for visits), as an `IloAny void*` pointer. Attaching profiles to a vehicle/visit is crucial to the implementation of the compatibility constraint, which takes one vehicle and one visit.

Step 11 Define the setVisitSkillProfiles function

Add the following code after the comment
`//Define the setVisitSkillProfiles function`.

```
void RoutingModel::setVisitSkillProfiles(const char* visitProfilePath) {
  // now read the techProfilePath table.
  IloCsvReader visitProfileReader(_env, visitProfilePath);
  for (IloCsvReader::LineIterator vpit(visitProfileReader); vpit.ok(); ++vpit)
{
    IloCsvLine line = *vpit;
    const char* visitName = line.getStringByHeader("name");
    IloInt profileId = line.getIntByHeader("VisitProfileId");
    IloVisit visit = IloVisit::Find(_env, visitName);
    SkillProfile* profile = getSkillProfile(profileId);
    if ( 0 != visit.getImpl() && 0 != profile ) {
      visit.setObject( (IloAny)profile );
    }
  }
  visitProfileReader.end();
}
```

This function identifies the `visitProfileId` required at each customer visit.

Step 12 Define the AreSkillsCompatible predicate

Add the following code after the comment `//Define the AreSkillsCompatible predicate`.

```
static IloBool AreSkillsCompatible(IloVisit visit, IloVehicle vehicle) {
  SkillProfile* visitProfile = (SkillProfile*)visit.getObject();
  SkillProfile* techProfile = (SkillProfile*)vehicle.getObject();
  if ( 0 != visitProfile ) {
    if ( 0 != techProfile ) {
      return techProfile->isGreaterThanOrEqual(visitProfile);
    } else {
      return IloFalse;
    }
  } else {
    return IloTrue;
  }
}
```

This predicate checks the compatibility relation between a visit and a technician. A visit is compatible with a technician if the visit profile is subsumed by the technician profile. A special case is also considered for a visit with no profile where all technicians are compatible.

Step 13 Define the postCompatibility function

Add the following code after the comment `//Define the postCompatibility function`.

```
void RoutingModel::postCompatibility() {
  IloVisitVehicleCompat compat(_env, AreSkillsCompatible);
  _mdl.add( IloCompatible(compat,  "skill compatibility"));
}
```

IBM ILOG Dispatcher provides a way for you to define compatibility relations between visits and vehicles. These relations can then be used to build compatibility constraints. In this example the constructor `IloVisitVehicleCompat` uses the predicate `AreSkillsCompatible` to build the compatibility relation. The function `IloCompatible` is then used to create the compatibility constraint to ensure that only compatible vehicles are used on a visit.

The compatibility constraint is used to check that only compatible visit/vehicle assignments are built, by removing incompatible vehicles. A predicate is used for convenience; you can also subclass a predefined class for more complex cases.

Solve

The solution is computed, improved, and displayed by methods previously presented for the PDP problem, with the addition of one goal.

Step 14 Add the goals

Add the following code after the comment `//Add the goals`.

```
IloGoal instantiateCost =
  IloDichotomize(env, dispatcher.getCostVar(), IloFalse);
IloGoal restoreSolution = IloRestoreSolution(env, _solution);
IloGoal goal =
  IloInsertionGenerate(env) && instantiateCost ;
```

`IloInsertionGenerate` is an *insertion* heuristic algorithm used to build a first solution for a routing plan. An insertion heuristic builds a solution to the routing model using an algorithm which consists of inserting visits at the lowest cost position at the time of insertion. This insertion point will not necessarily be the visit's lowest cost position when the entire routing plan has been constructed.

Step 15 Compile and run the program

The solution improvement phase finds a solution using 4 vehicles with a cost of 619.143 units

```
First Solution with cost: 735.716
Improving solution
***Improved Solution***
Number of fails              : 0
Number of choice points      : 1049
Number of variables          : 479
Number of constraints        : 108
Reversible stack (bytes)     : 68364
Solver heap (bytes)          : 333940
Solver global heap (bytes)   : 70960
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11160
Total memory used (bytes)    : 552836
Elapsed time since creation  : 0
Number of nodes              : 21
Number of visits             : 30
Number of vehicles           : 5
Number of dimensions         : 1
Number of accepted moves     : 13
===============
Cost         : 619.143
Number of vehicles used : 4
```

Review Exercises

For answers, see "Suggested Answers" on page 350.

1. What Dispatcher class is used to build a compatibility relation?
2. What predefined function is used to create a compatibility constraint from a compatibility relation?
3. What is an insertion heuristic algorithm used for?

Suggested Answers

Exercise 1

What Dispatcher class is used to build a compatibility relation?

Suggested Answer

`IloVisitVehicleCompat`.

Exercise 2

What predefined function is used to create a compatibility constraint from a compatibility relation?

Suggested Answer

`IloCompatible`.

Exercise 3

What is an insertion heuristic algorithm used for?

Suggested Answer

An insertion heuristic algorithm is used to build a first solution for a routing plan. An insertion heuristic builds a solution to the routing model using an algorithm which consists of inserting visits at the lowest cost position at the time of insertion.

Complete Program

The complete program follows. You can also view it online in the file `YourDispatcherHome/examples/src/technic1.cpp`.

```
// -------------------------------------------------------------- -*- C++ -*-
// File: examples/src/technic1.cpp
// --------------------------------------------------------------------------


#include <ildispat/ilodispatcher.h>

ILOSTLBEGIN

/**
 * This example shows the use of the visit-vehicle compatibility constraint.
 * The constraint is defined using skill proficiency vectors, attached
 * to both visits and vehicles.
 *
 *
 */



/**
 * The SkillProfile class is used to model vector a skill proficiencies.
 * it maps a level for each skill.
 */
```

```
class SkillProfile {
  IloEnv _env;
  IloInt _nbOfSkills;
  IloInt* _skillLevels;
public:
  SkillProfile(IloEnv env, IloInt nbOfSkills):
    _env(env),
    _nbOfSkills(nbOfSkills),
    _skillLevels(0)
  {
    _skillLevels = new (env) IloInt [ nbOfSkills ];
    for (IloInt s=0; s< _nbOfSkills; ++s) _skillLevels[s] = 0;
  }
  void setSkillLevel(IloInt skillIndex, IloInt level=1) {
    _skillLevels[ skillIndex ] = level; }

  /**
   * Returns true if <code>other</code> is comparable with this
   * and is greater.
   */
  IloBool isGreaterThanOrEqual(const SkillProfile* other);

  virtual void display(ILOSTD(ostream)& os) const {
    os << "[";
    for (IloInt s=0; s< _nbOfSkills; ++s) {
      os << _skillLevels[s];
      if ( s < _nbOfSkills-1) os << ", ";
    }
    os << "]";
  }

};

inline ostream& operator<<(ostream& os, const SkillProfile& profile) {
  profile.display(os); return os;
}


IloBool SkillProfile::isGreaterThanOrEqual(const SkillProfile* other) {
  for(IloInt index = 0;
      index < _nbOfSkills && other->_nbOfSkills; ++index) {
    if ( _skillLevels[index] < other->_skillLevels[index] ) {
       return IloFalse;
    }
  }
  return IloTrue;
}


class RoutingModel {
  IloEnv              _env;
  IloModel            _mdl;
  IloArray<SkillProfile*> _profiles;
  IloInt _nbOfSkills;

  const char* _nodePath;
  const char* _technicianPath;
  const char* _visitPath;
  const char* _profilePath;
  const char* _techProfilePath;  // table binding profiles to tecnicians
  const char* _visitProfilePath; // table binding profiles to visits

  void createDimensions();
  void createIloNodes(const char* nodePath);
  void createSkillProfiles(const char* profilePath);
  void setTechnicianSkillProfiles(const char* techProfilePath);
  void setVisitSkillProfiles(const char* visitProfilePath);
  void createTechnicians(const char* technicianPath);
  void createVisits(const char* visitsPath);

  void postCompatibility();

  /**
   * Lazy accessor to the profile table, with id as key.
   * First, missing array slots are filled with zeroes.
   * Then, if returned profile is null,
   * a new profile is created, stored and returned.
   */
  SkillProfile* getSkillProfile(IloInt profileId) {
    if ( _profiles.getSize() <= profileId ) {
      for (IloInt p= _profiles.getSize(); p <= profileId; ++p) {
        _profiles.add( (SkillProfile*) 0);
      }
    }
    SkillProfile* profile = _profiles[ profileId ] ;
    if ( 0 == profile ) {
      profile = new (_env) SkillProfile(_env, _nbOfSkills);
      _profiles[ profileId ] = profile;
    }
    return profile;
  }

public:
  RoutingModel(IloEnv env);
  ~RoutingModel() {}

  void parse(int argc, char** argv);
  void createModel();

  IloEnv getEnv() const { return _env; }
  IloModel getModel() const { return _mdl; }
};

RoutingModel::RoutingModel(IloEnv env)
  : _env(env),
  _mdl(env),
  _profiles(env),
  _nbOfSkills(3),
  _nodePath("../../../examples/data/technic1/nodes.csv"),
  _technicianPath("../../../examples/data/technic1/technicians.csv"),
  _visitPath("../../../examples/data/technic1/visits.csv"),
  _profilePath("../../../examples/data/technic1/SkillProfiles.csv"),
  _techProfilePath("../../../examples/data/technic1/techSkills.csv"),
  _visitProfilePath("../../../examples/data/technic1/visitSkills.csv")
{ }
```

```
void RoutingModel::parse(int argc, char** argv) {
  if ( argc > 1 ) _visitPath        = argv[1];
  if ( argc > 2 ) _technicianPath   = argv[2];
  if ( argc > 3 ) _visitProfilePath = argv[3];
  if ( argc > 4 ) _techProfilePath  = argv[4];
  if ( argc > 5 ) _profilePath      = argv[5];
  if ( argc > 6 ) _nodePath         = argv[6];
}

void RoutingModel::createModel() {
  createDimensions();
  createIloNodes(_nodePath);
  createTechnicians(_technicianPath);
  createVisits(_visitPath);
  createSkillProfiles(_profilePath);
  setTechnicianSkillProfiles(_techProfilePath);
  setVisitSkillProfiles(_visitProfilePath);
  postCompatibility();
}

// Create dimensions
void RoutingModel::createDimensions() {
  IloDimension2 time(_env, IloEuclidean, “Time”);
  time.setKey(“Time”);
  _mdl.add(time);
}

// Create IloNodes
void RoutingModel::createIloNodes(const char* nodePath) {
  IloCsvReader csvNodeReader(_env, nodePath);
  IloCsvReader::LineIterator it(csvNodeReader);
  while ( it.ok() ) {
    IloCsvLine line = *it;
    char* name = line.getStringByHeader(“name”);
    IloNode node(_env,
                 line.getFloatByHeader(“x”),
                 line.getFloatByHeader(“y”),
                 0,
                 name);
    node.setKey(name);
    ++it;
  }
  csvNodeReader.end();
}

// Create vehicles
void RoutingModel::createTechnicians(const char* techPath) {
  IloDimension2 time = IloDimension2::Find(_env, “Time”);
  IloCsvReader csvTechReader(_env, techPath);
  char namebuf[128];
  IloCsvReader::LineIterator it(csvTechReader);
  for( ; it.ok(); ++it) {
    IloCsvLine line = *it;
    char * namefirst = line.getStringByHeader(“first”);
    char * namelast = line.getStringByHeader(“last”);
    char * name = line.getStringByHeader(“name”);
    IloNum openTime = line.getFloatByHeader(“open”);
    IloNum closeTime = line.getFloatByHeader(“close”);
    IloNode node1 = IloNode::Find(_env, namefirst);
    IloNode node2 = IloNode::Find(_env, namelast);

    sprintf(namebuf, “start_%s”, name);
    IloVisit first(node1, namebuf);
    _mdl.add( first.getCumulVar(time) >= openTime );

    sprintf(namebuf, “end_%s”, name);
    IloVisit last(node2, namebuf);
    _mdl.add( last.getCumulVar(time) <= closeTime );

    IloVehicle technician(first, last, name);
    technician.setCost(time, 1.0);
    technician.setKey(name);
    _mdl.add(technician);

  }
  csvTechReader.end();
}

// Create visits
void RoutingModel::createVisits(const char* visitsPath) {
  IloDimension2 time = IloDimension2::Find(_env, “Time”);
  IloCsvReader csvVisitReader(_env, visitsPath);
  IloCsvReader::LineIterator  it(csvVisitReader);
  for( ;it.ok(); ++it) {
    IloCsvLine line = *it;
    //read visit data from files
    char * visitName = line.getStringByHeader(“name”);
    char * nodeName = line.getStringByHeader(“node”);
    IloNum minTime  = line.getFloatByHeader(“minTime”);
    IloNum maxTime  = line.getFloatByHeader(“maxTime”);
    IloNum serviceTime = line.getFloatByHeader(“dropTime”);
    IloNode node = IloNode::Find(_env, nodeName);
    IloVisit visit(node, visitName);

    _mdl.add( visit.getDelayVar(time) == serviceTime );
    _mdl.add( minTime <= visit.getCumulVar(time) <= maxTime);
    visit.setPenaltyCost(10000);
    visit.setKey(visitName);
    _mdl.add(visit);

  }
  csvVisitReader.end();
}

void RoutingModel::createSkillProfiles(const char* profilePath) {
  IloCsvReader skillProfileReader(_env, profilePath);
  for( IloCsvReader::LineIterator it(skillProfileReader); it.ok(); ++it) {
    IloCsvLine line = *it;
    IloInt profileId = line.getIntByHeader(“id”);
    IloInt skill1Level = line.getIntByHeader(“Skill1Level”);
    IloInt skill2Level = line.getIntByHeader(“Skill2Level”);
    IloInt skill3Level = line.getIntByHeader(“Skill3Level”);

    SkillProfile* profile = getSkillProfile(profileId);
    assert( 0 != profile );
    profile->setSkillLevel(0, skill1Level);
    profile->setSkillLevel(1, skill2Level);
    profile->setSkillLevel(2, skill3Level);
  }
  skillProfileReader.end();
}

void RoutingModel::setTechnicianSkillProfiles(const char* techProfilePath) {
  // now read the techProfilePath table.
  IloCsvReader techProfileReader(_env, techProfilePath);
  for (IloCsvReader::LineIterator tpit(techProfileReader); tpit.ok(); ++tpit) {
    IloCsvLine line = *tpit;
    const char* techName = line.getStringByHeader(“name”);
    IloInt profileId = line.getIntByHeader(“TechProfileId”);
    IloVehicle tech = IloVehicle::Find(_env, techName);
    SkillProfile* profile = getSkillProfile(profileId);
    if ( 0 != tech.getImpl() && 0 != profile ) {
      tech.setObject( (IloAny)profile );
    }
  }
  techProfileReader.end();
}

void RoutingModel::setVisitSkillProfiles(const char* visitProfilePath) {
  // now read the techProfilePath table.
  IloCsvReader visitProfileReader(_env, visitProfilePath);
  for (IloCsvReader::LineIterator vpit(visitProfileReader); vpit.ok(); ++vpit)
{
    IloCsvLine line = *vpit;
    const char* visitName = line.getStringByHeader(“name”);
    IloInt profileId = line.getIntByHeader(“VisitProfileId”);
    IloVisit visit = IloVisit::Find(_env, visitName);
    SkillProfile* profile = getSkillProfile(profileId);
    if ( 0 != visit.getImpl() && 0 != profile ) {
      visit.setObject( (IloAny)profile );
    }
  }
  visitProfileReader.end();
}

/**
 * This predicate codes the compatibility relation between a visit and a
 * technician. A visit is compatible with a technician if the
 * visit profile is subsumed by the technician profile.
 * To be defensive, a special case is also considered, when
 * the visit has no profile, then all technicians are compatible.
 */
static IloBool AreSkillsCompatible(IloVisit visit, IloVehicle vehicle) {
  SkillProfile* visitProfile = (SkillProfile*)visit.getObject();
  SkillProfile* techProfile = (SkillProfile*)vehicle.getObject();
  if ( 0 != visitProfile ) {
    if ( 0 != techProfile ) {
      return techProfile->isGreaterThanOrEqual(visitProfile);
    } else {
      return IloFalse;
    }
  } else {
    return IloTrue;
  }
}

void RoutingModel::postCompatibility() {
  IloVisitVehicleCompat compat(_env, AreSkillsCompatible);
  _mdl.add( IloCompatible(compat,  “skill compatibility”));
}

// Solving
class RoutingSolver {
  IloModel            _mdl;
  IloSolver           _solver;
  IloRoutingSolution  _solution;

  IloBool findFirstSolution(IloGoal goal);
  void greedyImprove(IloNHood nhood, IloGoal subGoal);
  void improve(IloGoal subgoal);
public:
  RoutingSolver(RoutingModel mdl);
  ~RoutingSolver() {}
  IloRoutingSolution getSolution() const { return _solution; }
  void printInformation(const char* heading = 0) const;
  void solve();
};

RoutingSolver::RoutingSolver(RoutingModel mdl)
  : _mdl(mdl.getModel()), _solver(mdl.getModel()), _solution(mdl.getModel()) {}

// Solving : find first solution
IloBool RoutingSolver::findFirstSolution(IloGoal goal) {
  if (!_solver.solve(goal)) {
    _solver.error() << “Infeasible Routing Plan” << endl;
    return IloFalse;
  }
  _solution.store(_solver);
  return IloTrue;
}

// Improve solution using nhood
void RoutingSolver::greedyImprove(IloNHood nhood, IloGoal subGoal) {
  _solver.out() << “Improving solution” << endl;
  IloEnv env = _solver.getEnv();
  nhood.reset();
  IloGoal improve = IloSingleMove(env,
                                  _solution,
                                  nhood,
                                  IloImprove(env),
                                  subGoal);
  while (_solver.solve(improve)) {}
}

// Improve solution
void RoutingSolver::improve(IloGoal subGoal) {
  IloEnv env = _solver.getEnv();
  IloNHood nhood =
    IloMakePerformed(env)
    +
    IloTwoOpt(env)
    + IloOrOpt(env)
    + IloRelocate(env)
    + IloCross(env)
    + IloExchange(env);
  greedyImprove(nhood, subGoal);
}

// Display Dispatcher information
void RoutingSolver::printInformation(const char* heading) const {
  if ( 0 != heading ) {
    _solver.out() << heading << endl;
  }
  IloDispatcher dispatcher(_solver);
  _solver.printInformation();
  dispatcher.printInformation();
  _solver.out() << "===============" << endl
                << "Cost         : " << dispatcher.getTotalCost() << endl
                << "Number of vehicles used : "
                << dispatcher.getNumberOfVehiclesUsed() << endl
                << "Solution     : " << endl
                << dispatcher << endl;
}

// Solving
void RoutingSolver::solve() {
  IloDispatcher dispatcher(_solver);
  IloEnv env = _solver.getEnv();

  IloGoal instantiateCost =
    IloDichotomize(env, dispatcher.getCostVar(), IloFalse);
  IloGoal restoreSolution = IloRestoreSolution(env, _solution);
  IloGoal goal =
    IloInsertionGenerate(env) && instantiateCost ;

  // Solving
  if (findFirstSolution(goal)) {
    _solver.out() << "First Solution with cost: "
                  << _solution.getObjectiveValue() << endl;
    improve(instantiateCost);
    _solver.solve(restoreSolution && instantiateCost);
    printInformation("***Improved Solution***");
  }
}

///////////////////////////////////////////////////////////////////////////////

int main(int argc, char * argv[]) {
  IloEnv env;
  try {
    RoutingModel mdl(env);
    mdl.parse(argc, argv);
    mdl.createModel();

    RoutingSolver rsolver(mdl);
    rsolver.solve();
  } catch(IloException& ex) {
    cerr << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Complete Output

```
/**
First Solution with cost: 735.716
Improving solution
***Improved Solution***
Number of fails              : 0
Number of choice points      : 1049
Number of variables          : 479
Number of constraints        : 108
Reversible stack (bytes)     : 68364
Solver heap (bytes)          : 333940
Solver global heap (bytes)   : 70960
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11160
Total memory used (bytes)    : 552836
Elapsed time since creation  : 0
Number of nodes              : 21
Number of visits             : 30
Number of vehicles           : 5
Number of dimensions         : 1
Number of accepted moves     : 13
===============
Cost         : 619.143
Number of vehicles used : 4
Solution     :
Unperformed visits : None
tech1 :
 -> start_tech1 Time[0..59.6393] -> visit3 Time[22.3607..82] -> visit9
Time[97..107] -> visit20 Time[118.18..188] -> end_tech1 Time[159.803..230]
tech2 :
 -> start_tech2 Time[0..16.6953] -> visit14 Time[32.0156..48.7109] -> visit17
Time[64.3763..81.0716] -> visit8 Time[95..105] -> visit19 Time[122.72..162.82]
-> visit7 Time[143.9..184] -> visit18 Time[163.9..204] -> end_tech2
Time[189.712..230]
tech3 :
 -> start_tech3 Time[0..30] -> visit2 Time[18..48] -> visit15 Time[61..71] ->
visit16 Time[96..110.749] -> visit5 Time[117.18..131.929] -> visit6
Time[137.18..151.929] -> visit13 Time[159..169] -> end_tech3 Time[180.18..230]
tech4 :
 -> start_tech4 Time[0..23.4691] -> visit10 Time[25.4951..48.9642] -> visit11
Time[67..70.1445] -> visit1 Time[103.401..106.545] -> visit12
Time[130.044..133.189] -> visit4 Time[155.855..159] -> end_tech4
Time[190.855..230]
tech5 : Unused

*/
```

C H A P T E R

15

CARP: Visiting Arcs Using Multiple Vehicles

In this lesson, you will learn how to:

- model a routing problem to visit arcs
- use the class `IloDispatcherGraph` to store and compute the costs of the arcs
- use a metaheuristic in search with `IloDispatcherGLS`

Describe

In the vehicle routing problems presented in the previous lessons the goal was to perform visits located at given nodes. There is another type of problem where the goal is to visit *arcs* instead, arcs being a link between two nodes (arcs typically represent streets or road segments). This type of problem can involve activities such as winter gritting (dispensing salt or grit on snow-covered roads), door-to-door mail delivery, street cleaning, or garbage collecting. This chapter focuses on a particular type of problem called a *Capacitated Arc Routing Problem (CARP)* in which vehicles have a limited capacity.

The main difference in considering an arc problem is that a visit is not defined as a trip to a node, but is rather the path (arc) between two nodes. A quantity of a good is delivered to each arc and vehicles can carry a limited amount of goods (capacity). Time windows are not considered here although they could easily be modeled. All vehicles are identical and perform a single tour leaving and arriving at a unique depot.

Distances between nodes are computed from a graph representing the network of roads. For a given dimension *d*, the distance between two nodes *n1* and *n2* using a vehicle *v* is the sum of the value according to *d* of all the arcs on the cheapest path going from *n1* to *n2* using *v*. The cheapest path is the path of minimum cost using *v*. Furthermore, the distance between two visits located at arc (*n1*, *n2*) and arc (*n3*, *n4*) is defined to be the distance between nodes *n2* and *n3*.

In this example, two extrinsic dimensions are used, one representing time and the other length. Both dimensions have their distance function based on the graph; the shortest path between two nodes depends on the arcs that are available in the graph. The cost function is a linear combination of the duration and of the length of the route of the vehicles.

The following figure shows a sample road network for an arc problem with eight nodes and 12 arcs (the example in this lesson has 52 nodes).

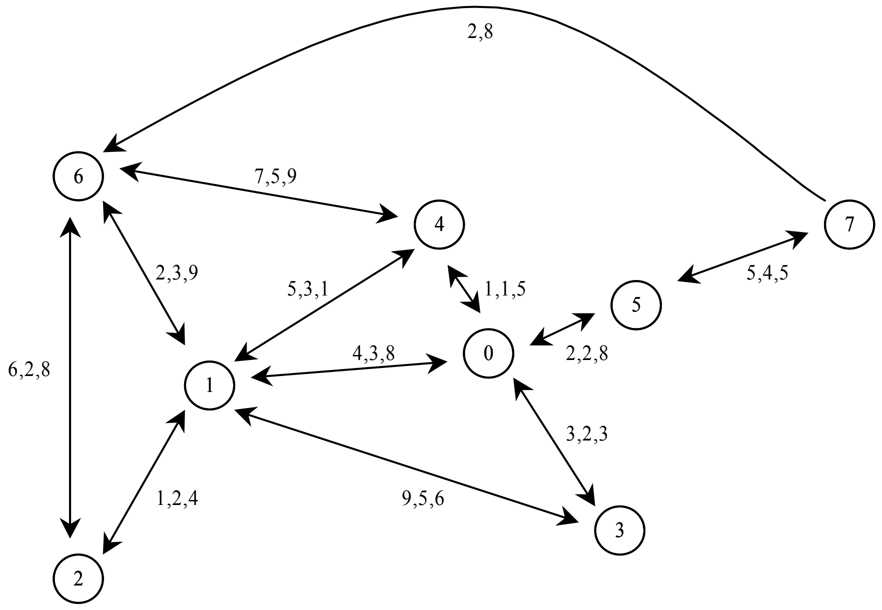

Figure 15.1 *An Example Arc Problem*

The arrows show in which direction an arc can be taken (a double arrow means it can be taken in both directions). The values on the arcs indicate respectively the duration in time and length of the arc. The third number represents the quantity of goods to be delivered to this arc; when a third number is not present the arc does not need to be visited.

Step 1 Describe the problem

The first step is to write a natural language description of the problem. A good way to start this process is to analyze the constraints and objectives. The primary constraint in this problem is that vehicles must deliver a quantity of goods to selected arcs on the road network. Not all arcs need to be visited (that is, have goods delivered along the arc), but those arcs may be travelled through anyway in order to minimize the travel cost.

The objective is to minimize the cost of the delivery of the quantity of goods along the arcs.

Model

Once you have written a description of your problem, you can use Dispatcher classes to model it.

Step 2 Open the example file

Open the example file `YourDispatcherHome/examples/src/tutorial/carp_partial.cpp` in your development environment.

As in the previous lessons, you will use a `RoutingModel` class to call the functions that create the dimensions, nodes, vehicles, and visits. For this example, `RoutingModel` also calls a function to create the graph that represents the road network.

Step 3 Declare the RoutingModel class

Add the following code after the comment `//Declare the RoutingModel class.`

```
class RoutingModel {
  IloEnv             _env;
  IloModel           _mdl;
  IloDispatcherGraph  _graph;
  IloDimension2      _time;
  IloDimension2      _distance;
  IloDimension1      _weight;

  void addDimensions();
  void loadGraphInformation (char* arcFileName);
  void CreateIloNodes(char* nodeFileName, char* coordFileName);
  void CreateVehicles(char* vehicleFileName);
  void CreateVisits(char* visitsFileName);
```

An instance of `IloDispatcherGraph` is declared (`_graph`); this instance is used to create a road network of nodes. It can then be used to compute and store the cheapest paths between nodes based on the cost functions for any given vehicle.

The function `loadGraphInformation` is used to load and create the arcs and the arc costs that are stored in `_graph`.

The remaining code of `RoutingModel` is provided for you, and it calls a function (`modifyGraphArcCost`) that allows you to modify the cost of an arc within the code itself, rather than in a csv data file.

```
public:
  RoutingModel(IloEnv env, int argc, char* argv[]);
  ~RoutingModel() {}
  IloEnv    getEnv() const { return _env; }
  IloModel  getModel() const { return _mdl; }
  void modifyGraphArcCost();
  IloDispatcherGraph getGraph() const { return _graph; }
};
```

The function `RoutingModel:addDimensions` uses a shortest path calculation to represent the dimensions of time and distance in the graph network.

Step 4 Create the distance functions

Add the following code after the comment `//Create the distance functions`.

```
  IloDistance SP_time =   IloGraphDistance (_graph);
  _time  =IloDimension2 (_env, SP_time, "time");
  _mdl.add(_time);

  IloDistance SP_distance = IloGraphDistance (_graph);
  _distance =IloDimension2 (_env, SP_distance, "distance");
  _mdl.add(_distance);
}
```

The shortest path time and distance functions (`SP_time` and `SP_distance`) are defined and added to the model (instead of the predefined functions `IloEuclidian` and `IloManhatten` that you have used before).

The next function creates the graph containing the arcs.

Step 5 Load the graph information

Add the following code after the comment `//Load the graph information`.

```
void RoutingModel::loadGraphInformation (char* arcFileName)    {
  _graph.createArcsFromFile (arcFileName);
  _graph.loadArcDimensionDataFromFile (arcFileName, _time);
  _graph.loadArcDimensionDataFromFile (arcFileName, _distance);
}
```

The member function `IloDispatcherGraph::createArcsFromFile` loads the road network data from a csv file, and creates the necessary nodes and arcs. The member function

`IloDispatcherGraph::loadArcDimensionDataFromFile` loads the arc cost data from a csv file.

The function `CreateIloNodes` includes a command to associate the `IloNodes` to the graph nodes.

Step 6 Create the nodes

Add the following code after the comment `//Create the nodes`.

```
    _graph.associateByCoordsInFile (node, coordFileName);
```

Instances of `IloNode` must be positioned within the graph. This can be done node-by-node with the method `IloDispatcherGraph::setLocation`, but this becomes impractical with even modestly sized networks. Instead, in this lesson you use the method `associateByCoordsInFile` to look up the coordinates of a given `IloNode` in a csv file, and then automatically associate the node to the graph node with matching coordinates.

Next, you create the vehicles with a cost function that uses a cost ratio.

Step 7 Set the vehicle costs

Add the following code after the comment `//Set the vehicle costs`.

```
    vehicle.setCost(_time, 100 * costRatio);
    vehicle.setCost(_distance, 100 * (1- costRatio));
    vehicle.setCapacity(_weight, capacity);
    _mdl.add(vehicle);
```

The following line of code appears earlier in the `CreateVehicles` function:

```
    IloNum costRatio = line.getFloatByHeader("costRatio");
```

The `costRatio` is data from the vehicle csv file, and can be used, for example, to model vehicles that have different operating costs. Using different cost ratios for the different vehicles will change the solution paths; paths depend not only on the distance between nodes, but also on the way vehicle costs are calculated.

Next, the `CreateVisits` function adds the actual arc visits to the model.

Step 8 Create the visits

Add the following code after the comment `//Create the visits`.

```
IloNode node1 = IloNode::Find(_env, nodeName1);
IloNode node2 = IloNode::Find(_env, nodeName2);

IloVisit visit1(node1, node2, visitName);
_mdl.add(visit1.getTransitVar(_weight) == quantity);
_mdl.add(visit1.getDelayVar(_time) == timeValue );
_mdl.add(visit1.getDelayVar(_distance) == distanceValue );
_mdl.add(visit1);
if(symmetric) {
  visit1.setPenaltyCost(0);
  char visitName2[16];
  sprintf(visitName2, "%s%c", visitName, c);
  IloVisit visit2(node2, node1, visitName2);
  _mdl.add(visit2.getTransitVar(_weight) == quantity);
  _mdl.add(visit2.getDelayVar(_time) == timeValue );
  _mdl.add(visit2.getDelayVar(_distance) == distanceValue );
  visit2.setPenaltyCost(0);
  _mdl.add(visit2);
  _mdl.add(visit1.performed() + visit2.performed() == 1);
```

This code is the second part of the `createVisits` function. The first part of the iteration reads in the nodes, time and distance values, and `quantity` to be delivered per visit.

To create an arc visit rather than a visit to a single node, you provide two node names to the `IloVisit` constructor. If the visit to the arc is `symmetric`—that is, the cost to visit the arc is the same regardless of which node is visited first—then both possible paths for the arc visit are added to the model with zero penalty cost.

The constraint `_mdl.add(visit1.performed() + visit2.performed() == 1)` states that just one of those two visits must be performed.

In problems of this type it may sometimes be desirable to be able to quickly modify the costs of certain paths, due to traffic or road conditions, experimental modeling, or other reasons. You may not want to edit your data files to model these factors; the function `modifyGraphArcCost` is used for this purpose.

Step 9 Modify graph arc costs

Add the following code after the comment `//Modify graph arc costs.`

```
void RoutingModel::modifyGraphArcCost() {
  IloDispatcherGraph::Arc arc1 = _graph.getArc(3916); //Arc from 1043 to 987
  IloDispatcherGraph::Arc arc2 = _graph.getArc(4134); //Arc from 1043 to 1042
  IloDispatcherGraph::Arc arc3 = _graph.getArc(4136); //Arc from 1043 to 1044
  IloDispatcherGraph::Arc arc4 = _graph.getArc(4137); //Arc from 1043 to 1099

  _graph.setArcCost(arc1, _time, 10);
  _graph.setArcCost(arc2, _time, 10);
  _graph.setArcCost(arc3, _time, 10);
  _graph.setArcCost(arc4, _time, 10);
}
```

This function is used to quickly modify the cost of a few particular arcs. This may be appropriate if, for example, conditions merit a temporary increase in the cost of visiting an arc. The impact on the solution cost can be significant.

Solve

The solving section of this example introduces *metaheuristics*. In previous lessons, heuristics terminated when no cost-reducing move could be found at a local minimum. Metaheuristics allow the search to continue by allowing neighborhood moves that degrade the current solution and allow the search to move from the current position. This degradation must be carefully controlled, however, to prevent the search from moving to very poor solutions. This control is provided by various types of metaheuristics. The type used in this lesson is based on guided local search (GLS).

Guided local search works by making a series of greedy searches, each to a local minimum. However, it reduces a different cost function from the original. If the original cost function is represented by c, then guided local search attempts to reduce the cost $c+wp$, where p is a penalty term that is adjusted every time a local minimum is reached, and w is a constant. So, in essence, guided local search tries to minimize a combination of the true cost and a penalty term. The weighting constant w is an important search parameter that determines how important the penalty term is with respect to the true cost.

Guided local search is constructed so that the penalty term is higher when the search moves to solutions that resemble previous ones. This is done by recording certain arcs as “bad” each time a local minimum is reached. When these arcs appear in a solution, the penalty term is increased. Guided local search determines which arcs are bad based upon the cost of the arc in the solution and how often that arc has previously appeared at local minima.

More complete details of the use of metaheuristics can be found in Appendix C, *Predefined Search Heuristics and Metaheuristics* and in the *IBM ILOG Dispatcher Reference Manual*.

The first step in the solving section is to create the `RoutingSolver` function.

Step 10 Create the RoutingSolver function

Add the following code after the comment `//Create the RoutingSolver function.`

```
  void greedyImprove(IloNHood nhood, IloGoal subGoal);
  void improve(IloGoal subgoal);
  void improveWithFastGLS(IloNHood nhood, IloInt nbOfIter, IloGoal subgoal);
public:
  RoutingSolver(RoutingModel mdl);
  ~RoutingSolver() {}
  IloRoutingSolution getSolution() const { return _solution; }
  void printInformation() const;
  IloBool findFirstSolution();
  void improveWithNHood(IloInt nbIter);
  void modifyFilterLevelAndRestore();
  void restoreSolution();
  void displayPaths(IloDispatcherGraph graph);
};
```

The function `improveWithFastGLS` uses the metaheuristic `IloDispatcherGLS` to get out of the local minimum solution to find a better one.

Two new functions appear in the class `RoutingSolver` for this lesson: `modifyFilterLevelAndRestore` and `restoreSolution`.

Step 11 Modify filter level and restore solution

Add the following code after the comment `//Modify filter level and restore solution`.

```
void RoutingSolver::modifyFilterLevelAndRestore() {
  _dispatcher.setFilterLevel(IlcMedium);
  IloNum cost = _solution.getObjectiveValue();
  _solver.solve(_restoreSolution
      && IloSetMax(_env, _dispatcher.getCostVar(), cost));
  _dispatcher.setFilterLevel(IlcLow);
}
```

This function changes the filter level of the propagation before restoring the solution. `IlcMedium` is the highest level of propagation, and takes considerably longer than `IlcLow`. This function is called to have better propagation of the cumul variables and to have a better display of the solution.

Step 12 Add the restoreSolution function

Add the following code after the comment `//Add the restoreSolution function.`

```
void RoutingSolver::restoreSolution() {
  _solver.solve(_restoreSolution && _instantiateCost);
  _solution.store(_solver);
  _solver.out() << "Modified cost: " << _dispatcher.getTotalCost() << endl;
}
```

This function is called by `main` to restore the solution after `modifyGraphArcCost` has been called to modify the original arc cost data.

The next function, `improveWithFastGLS`, uses the metaheuristic `IloDispatcherGLS` to get out of the local minimum solution to find a better one.

Step 13 Add the metaheuristic function

Add the following code after the comment `//Add the metaheuristic function.`

```
void RoutingSolver::improveWithFastGLS(IloNHood nhood,
                                       IloInt nbOfIter,
                                       IloGoal instantiateCost) {
  nhood.reset();
  _solver.out() << "Improving with first-accept GLS" << endl;
  IloRoutingSolution rsol = _solution.makeClone(_env);
  IloRoutingSolution best = _solution.makeClone(_env);
  IloDispatcherGLS dgls(_env, 0.2);
  IloGoal move = IloSingleMove(_env, rsol, nhood, dgls, instantiateCost);
  move = move && IloStoreBestSolution(_env, best);
  IloNumVar costVar = _dispatcher.getCostVar();
  IloCouple(nhood, dgls);
  for (IloInt i = 0; i < nbOfIter; i++) {
    if (_solver.solve(move)) {
      _solver.out() << "Cost = " << _solver.getMax(costVar) << endl;
    } else {
      _solver.out() << "---" << endl;
      if (dgls.complete()) break;
    }
  }
  _solver.out() << endl;
  IloDecouple(nhood, dgls);
  IloGoal restoreSolution = IloRestoreSolution(_env, best) && instantiateCost;
  _solver.solve(restoreSolution);
  _solution.store(_solver);
  rsol.end();
  best.end();
  dgls.end();
}
```

IloDispatcherGLS tries to penalize “bad” arcs—arcs with a high cost. However, if an arc has been penalized a number of times, the importance of cost reduces. This is due to the fact that if an arc has been penalized a large number of times and is still in the solution, there may be no better arc with which to replace it and it is probably best to start looking elsewhere to place penalties.

The functions IloCouple and IloDecouple connect and disconnect a neighborhood to a metaheuristic. The neighborhood must be coupled to an instance of IloDispatcherGLS, otherwise an exception is thrown.

Step 14 The display paths function

Locate the following code after the comment //The display paths function.

```
void RoutingSolver::displayPaths(IloDispatcherGraph graph) {
  _env.out() << "Paths" << endl;
  for (IloIterator<IloVehicle> iter1(_env); iter1.ok(); ++iter1) {
    IloVehicle vehicle = *iter1;
    IloVisit visit1 = vehicle.getFirstVisit();
    for (IloDispatcher::RouteIterator iter2(_dispatcher, vehicle);
         iter2.ok();
         ++iter2) {
      IloVisit visit2 = *iter2;
      if (visit1 != visit2) {
        _env.out() << visit1.getName() << " {"
                  <<  graph.getLocation(visit1.getStartNode())
                                  .getIndex();
        for (IloDispatcherGraph::PathIterator iter3(graph,
                                                    visit1.getEndNode(),
                                                    visit2.getStartNode(),
                                                    vehicle);
             iter3.ok();
             ++iter3) {
          _env.out() << " -> " << (*iter3).getIndex();
        }
        _env.out() << "} -> ";
        visit1 = visit2;
      }
    }
    _env.out() << vehicle.getLastVisit().getName() << endl;
  }
  _env.out() << endl;
}
```

The function displayPaths is used to display the route of each vehicle following the actual paths on the road network. An instance of IloIterator<IloVehicle> is used to iterate over the vehicles, and an instance of IloDispatcher::RouteIterator is used to iterate over the visits of each vehicle. IloDispatcherGraph::PathIterator is used to iterate over the nodes present on the shortest path between visits.

Finally, `main` calls these functions.

Step 15 Add the main function

Add the following code after the comment `//Add the main function.`

```
int main(int argc, char * argv[]) {
  IloEnv env;
  try {
    RoutingModel mdl(env, argc, argv);
    RoutingSolver solver(mdl);
    if (solver.findFirstSolution()) {
      solver.printInformation();
      solver.improveWithNHood(250);
      solver.modifyFilterLevelAndRestore();
      solver.printInformation();
      solver.displayPaths(mdl.getGraph());
      mdl.modifyGraphArcCost();
      solver.restoreSolution();
      solver.improveWithNHood(50);
      solver.modifyFilterLevelAndRestore();
      solver.printInformation();
      solver.displayPaths(mdl.getGraph());
    }
  } catch(IloException& ex) {
    cerr << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Note that three sets of results are displayed: a set before and a set after the first `modifyFilterLevelAndRestore` function is called to change the propagation to `IlcMedium`, and one set after `modifyGraphArcCost` is called, increasing the cost of certain arcs.

Step 16 Compile and run the program

The complete program and output are presented in “Complete Program” on page 375.

First Solution Information

The first solution phase, before `improveWithNHood(250)` and `modifyFilterLevelAndRestore` have been called, finds a solution using 3 vehicles with a cost of 8100.14 units:

```
8100.14
Number of fails              : 218
Number of choice points      : 1048
Number of variables          : 1550
Number of constraints        : 197
Reversible stack (bytes)     : 132684
Solver heap (bytes)          : 715840
Solver global heap (bytes)   : 83476
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11160
Total memory used (bytes)    : 1011572
Elapsed time since creation  : 1.602
Number of nodes              : 52
Number of visits             : 55
Number of vehicles           : 5
Number of dimensions         : 3
Number of accepted moves     : 0
===============
Cost         : 8100.14
Number of vehicles used : 3
```

Improved Solution Information

The first solution is improved with `improveWithNHood(250)` and `modifyFilterLevelAndRestore`:

```
Improving solution
Improving with first-accept GLS
---
Cost = 7781.68
Cost = 7712.67
Cost = 7616.17
Cost = 7592.99
---
```

The solution improvement phase continues lowering the cost until it finds a solution using 3 vehicles with a cost of 7413.03 units:

```
---
Cost = 7546.96
Cost = 7546.96
---

Number of fails                : 0
Number of choice points        : 0
Number of variables            : 2086
Number of constraints          : 601
Reversible stack (bytes)       : 221124
Solver heap (bytes)            : 1636420
Solver global heap (bytes)     : 99556
And stack (bytes)              : 20124
Or stack (bytes)               : 44244
Search Stack (bytes)           : 4044
Constraint queue (bytes)       : 18172
Total memory used (bytes)      : 2043684
Elapsed time since creation    : 0.13
Number of nodes                : 52
Number of visits               : 55
Number of vehicles             : 5
Number of dimensions           : 3
Number of accepted moves       : 193
===============
Cost         : 7413.03
Number of vehicles used : 3
```

Solution Information with Modified Arc Costs

Now the problem is solved a second time. The cost is modified with `modifyGraphArcCost` and then reoptimized with `improveWithNHood(50)`. Note that the modified cost is large until the solution is improved.

```
Modified cost: 8311.26
Improving solution
Improving with first-accept GLS
---
Cost = 7506.89
Cost = 7482.83
---
```

This solution improvement phase continues lowering the cost until it finds a solution using 3 vehicles with a cost of 7459.55 units:

```
Cost = 7626.4

Number of fails              : 0
Number of choice points      : 0
Number of variables          : 2086
Number of constraints        : 601
Reversible stack (bytes)     : 221124
Solver heap (bytes)          : 1636420
Solver global heap (bytes)   : 99556
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 18172
Total memory used (bytes)    : 2043684
Elapsed time since creation  : 0.12
Number of nodes              : 52
Number of visits             : 55
Number of vehicles           : 5
Number of dimensions         : 3
Number of accepted moves     : 230
===============
Cost          : 7459.55
Number of vehicles used : 3
```

Review Exercises

For answers, see “Suggested Answers” on page 374.

1. How do you create an arc visit, instead of a visit to a single node?
2. What is the class name of the guided local search metaheuristic in Dispatcher?
3. What do the functions `IloCouple` and `IloDecouple` do?

Suggested Answers

Exercise 1

How do you create an arc visit, instead of a visit to a single node?

Suggested Answer

You pass two node names to the `IloVisit` constructor instead of just one node name.

Exercise 2

What is the class name of the guided local search metaheuristic in Dispatcher?

Suggested Answer

`IloDispatcherGLS`.

Exercise 3

What do the functions `IloCouple` and `IloDecouple` do?

Suggested Answer

The functions `IloCouple` and `IloDecouple` are used to connect and disconnect a neighborhood to a metaheuristic, in this example, to `IloDispatcherGLS`. The neighborhood must be coupled to an instance of `IloDispatcherGLS`, otherwise an exception is thrown when you try to use the instance in the search.

Complete Program

The complete program follows. You can also view it online in the file `YourDispatcherHome/examples/src/carp.cpp`.

```
// -------------------------------------------------------------- -*- C++ -*-
// File: examples/src/carp.cpp
// --------------------------------------------------------------------------


#include <ildispat/ilodispatcher.h>

ILOSTLBEGIN

////////////////////////////////////////////////////////////////////////////////
// Modeling
class RoutingModel {
  IloEnv              _env;
  IloModel            _mdl;
  IloDispatcherGraph  _graph;
  IloDimension2       _time;
  IloDimension2       _distance;
  IloDimension1       _weight;

  void addDimensions();
  void loadGraphInformation (char* arcFileName);
  void CreateIloNodes(char* nodeFileName, char* coordFileName);
  void CreateVehicles(char* vehicleFileName);
  void CreateVisits(char* visitsFileName);

public:
  RoutingModel(IloEnv env, int argc, char* argv[]);
  ~RoutingModel() {}
  IloEnv    getEnv() const { return _env; }
  IloModel  getModel() const { return _mdl; }
  void modifyGraphArcCost();
  IloDispatcherGraph getGraph() const { return _graph; }
};

RoutingModel::RoutingModel( IloEnv env,
                            int argc,
                            char* argv[]):
  _env(env), _mdl(env), _graph(env) {
  addDimensions();

 // load dispatcher graph information from file and add instance-specific
features
    char * arcFileName;
    if(argc < 5)  arcFileName = (char *) "../../../examples/data/
dispatcherGraphData/gridNetwork.csv";
    else          arcFileName = argv[4];

        loadGraphInformation (arcFileName);

//create IloNodes
    char * nodeFileName;
    if(argc < 4)  nodeFileName = (char *) "../../../examples/data/carp/
nodes.csv";
    else          nodeFileName = argv[3];
    char * nodeCoordsFile;
    if(argc < 6)  nodeCoordsFile = (char *) "../../../examples/data/carp/
coordTable.csv";
    else          nodeCoordsFile = argv[5];
    CreateIloNodes(nodeFileName, nodeCoordsFile);

  //create vehicles
    char * vehiclesFileName;
    if(argc < 2)  vehiclesFileName = (char *) "../../../examples/data/carp/
vehicles.csv";
    else          vehiclesFileName = argv[1];
    CreateVehicles(vehiclesFileName);

  //create visits
    char * visitsFileName;
    if(argc < 3)  visitsFileName = (char *) "../../../examples/data/carp/
visits.csv";
    else          visitsFileName = argv[2];
    CreateVisits(visitsFileName);
}

// Create distance functions for dimensions, add dimensions to model
void RoutingModel::addDimensions() {
  _weight =IloDimension1 (_env, "weight");
  _mdl.add(_weight);

  IloDistance SP_time =   IloGraphDistance (_graph);
  _time  =IloDimension2 (_env, SP_time, "time");
  _mdl.add(_time);

  IloDistance SP_distance = IloGraphDistance (_graph);
  _distance =IloDimension2 (_env, SP_distance, “distance”);
  _mdl.add(_distance);
}

// load network topology and travel costs from files.
void RoutingModel::loadGraphInformation (char* arcFileName)    {
  _graph.createArcsFromFile (arcFileName);
  _graph.loadArcDimensionDataFromFile (arcFileName, _time);
  _graph.loadArcDimensionDataFromFile (arcFileName, _distance);
}

//create IloNodes
void RoutingModel::CreateIloNodes(char* nodeFileName, char* coordFileName) {
  IloCsvReader csvNodeReader(_env, nodeFileName);
  IloCsvReader::LineIterator  it(csvNodeReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char * name = line.getStringByHeader(“name”);
    IloNode node(_env, line.getFloatByHeader(“x”),
        line.getFloatByHeader(“y”), 0, name);
    node.setKey(name);

    _graph.associateByCoordsInFile (node, coordFileName);

    ++it;
  }
  csvNodeReader.end();
}

//create vehicles
void RoutingModel::CreateVehicles(char* vehicleFileName) {
  IloCsvReader csvVehicleReader(_env, vehicleFileName);
  IloCsvReader::LineIterator  it(csvVehicleReader);
  while(it.ok()) {
    IloCsvLine line = *it;
    char * namefirst = line.getStringByHeader(“first”);
    char * namelast = line.getStringByHeader(“last”);
    char * name = line.getStringByHeader(“name”);
    IloNum capacity = line.getFloatByHeader(“capacity”);

    IloNum costRatio = line.getFloatByHeader(“costRatio”);

    IloNode node1 = IloNode::Find(_env, namefirst);
    IloNode node2 = IloNode::Find(_env, namelast);
    IloVisit first(node1, “depot”);
    _mdl.add(first.getCumulVar(_distance) == 0);
    _mdl.add(first.getCumulVar(_time) == 0);
    IloVisit last(node2, “depot”);
    IloVehicle vehicle(first, last, name);

    vehicle.setCost(_time, 100 * costRatio);
    vehicle.setCost(_distance, 100 * (1- costRatio));
    vehicle.setCapacity(_weight, capacity);
    _mdl.add(vehicle);

    ++it;
  }
  csvVehicleReader.end();
```

```
}

//create visits
void RoutingModel::CreateVisits(char* visitsFileName) {
  IloCsvReader csvVisitReader(_env, visitsFileName);
  IloCsvReader::LineIterator  it(csvVisitReader);
  char c = ‘s’;
  while(it.ok()){
    IloCsvLine line = *it;
    //read visit data from files
    char * visitName =  line.getStringByHeader(“name”);
    char * nodeName1 = line.getStringByHeader(“nodeName1”);
    char * nodeName2 = line.getStringByHeader(“nodeName2”);
    IloInt symmetric = line.getIntByHeader(“symmetric”);
    IloNum quantity = line.getFloatByHeader(“quantity”);
    IloNum timeValue = line.getFloatByHeader(“time”);
    IloNum distanceValue = line.getFloatByHeader(“distance”);

    IloNode node1 = IloNode::Find(_env, nodeName1);
    IloNode node2 = IloNode::Find(_env, nodeName2);

    IloVisit visit1(node1, node2, visitName);
    _mdl.add(visit1.getTransitVar(_weight) == quantity);
    _mdl.add(visit1.getDelayVar(_time) == timeValue );
    _mdl.add(visit1.getDelayVar(_distance) == distanceValue );
    _mdl.add(visit1);
    if(symmetric) {
      visit1.setPenaltyCost(0);
      char visitName2[16];
      sprintf(visitName2, “%s%c”, visitName, c);
      IloVisit visit2(node2, node1, visitName2);
      _mdl.add(visit2.getTransitVar(_weight) == quantity);
      _mdl.add(visit2.getDelayVar(_time) == timeValue );
      _mdl.add(visit2.getDelayVar(_distance) == distanceValue );
      visit2.setPenaltyCost(0);
      _mdl.add(visit2);
      _mdl.add(visit1.performed() + visit2.performed() == 1);

    }
    ++it;
  }
  csvVisitReader.end();
}

 // Modify problem set-up by modifying the cost of un arc in the graph

void RoutingModel::modifyGraphArcCost() {
  IloDispatcherGraph::Arc arc1 = _graph.getArc(3916); //Arc from 1043 to 987
  IloDispatcherGraph::Arc arc2 = _graph.getArc(4134); //Arc from 1043 to 1042
  IloDispatcherGraph::Arc arc3 = _graph.getArc(4136); //Arc from 1043 to 1044
  IloDispatcherGraph::Arc arc4 = _graph.getArc(4137); //Arc from 1043 to 1099

  _graph.setArcCost(arc1, _time, 10);
  _graph.setArcCost(arc2, _time, 10);
  _graph.setArcCost(arc3, _time, 10);
  _graph.setArcCost(arc4, _time, 10);
}
```

```
////////////////////////////////////////////////////////////////////////////////
// Solving
class RoutingSolver {
  IloModel            _mdl;
  IloSolver           _solver;
  IloRoutingSolution  _solution;
  IloEnv              _env;
  IloDispatcher       _dispatcher;
  IloGoal             _instantiateCost;
  IloGoal             _restoreSolution;
  IloGoal             _goal;

  void greedyImprove(IloNHood nhood, IloGoal subGoal);
  void improve(IloGoal subgoal);
  void improveWithFastGLS(IloNHood nhood, IloInt nbOfIter, IloGoal subgoal);
public:
  RoutingSolver(RoutingModel mdl);
  ~RoutingSolver() {}
  IloRoutingSolution getSolution() const { return _solution; }
  void printInformation() const;
  IloBool findFirstSolution();
  void improveWithNHood(IloInt nbIter);
  void modifyFilterLevelAndRestore();
  void restoreSolution();
  void displayPaths(IloDispatcherGraph graph);
};

RoutingSolver::RoutingSolver(RoutingModel mdl):
  _mdl(mdl.getModel()),
  _solver(mdl.getModel()),
  _solution(mdl.getModel()),
  _env(mdl.getEnv()),
  _dispatcher(_solver) {
    _instantiateCost = IloGoal (IloDichotomize(_env, _dispatcher.getCostVar(),
IloFalse));
    _restoreSolution = IloGoal(IloRestoreSolution(_env, _solution));
    _goal = IloGoal(IloSavingsGenerate(_env) && _instantiateCost);
  }

// Solving : find first solution
IloBool RoutingSolver::findFirstSolution() {
  if (!_solver.solve(_goal)) {
    _solver.error() << “Infeasible Routing Plan” << endl;
    return IloFalse;
  }
  _solver.out() << _dispatcher.getTotalCost() << endl;
  _solution.store(_solver);
  return IloTrue;
}

// Improve solution using nhood
void RoutingSolver::greedyImprove(IloNHood nhood, IloGoal subGoal) {
  _solver.out() << “Improving solution” << endl;
  nhood.reset();
  IloGoal improve = IloSingleMove(_env,
                                  _solution,
                                  nhood,
                                  IloImprove(_env),
                                  subGoal);
  while (_solver.solve(improve)) {}
}

// Modify dispatcher filter level and resolve
void RoutingSolver::modifyFilterLevelAndRestore() {
  _dispatcher.setFilterLevel(IlcMedium);
  IloNum cost = _solution.getObjectiveValue();
  _solver.solve(_restoreSolution
      && IloSetMax(_env, _dispatcher.getCostVar(), cost));
  _dispatcher.setFilterLevel(IlcLow);
}

// Modify graph arc cost and resolve
void RoutingSolver::restoreSolution() {
  _solver.solve(_restoreSolution && _instantiateCost);
  _solution.store(_solver);
  _solver.out() << "Modified cost: " << _dispatcher.getTotalCost() << endl;
}

void RoutingSolver::improveWithFastGLS(IloNHood nhood,
                                       IloInt nbOfIter,
                                       IloGoal instantiateCost) {
  nhood.reset();
  _solver.out() << "Improving with first-accept GLS" << endl;
  IloRoutingSolution rsol = _solution.makeClone(_env);
  IloRoutingSolution best = _solution.makeClone(_env);
  IloDispatcherGLS dgls(_env, 0.2);
  IloGoal move = IloSingleMove(_env, rsol, nhood, dgls, instantiateCost);
  move = move && IloStoreBestSolution(_env, best);
  IloNumVar costVar = _dispatcher.getCostVar();
  IloCouple(nhood, dgls);
  for (IloInt i = 0; i < nbOfIter; i++) {
    if (_solver.solve(move)) {
      _solver.out() << "Cost = " << _solver.getMax(costVar) << endl;
    } else {
      _solver.out() << "---" << endl;
      if (dgls.complete()) break;
    }
  }
  _solver.out() << endl;
  IloDecouple(nhood, dgls);
  IloGoal restoreSolution = IloRestoreSolution(_env, best) && instantiateCost;
  _solver.solve(restoreSolution);
  _solution.store(_solver);
  rsol.end();
  best.end();
  dgls.end();
}

// Display Dispatcher information
void RoutingSolver::printInformation() const {
  _solver.printInformation();
  _dispatcher.printInformation();
  _solver.out() << "===============" << endl
                << "Cost         : " << _dispatcher.getTotalCost() << endl
                << "Number of vehicles used : "
                << _dispatcher.getNumberOfVehiclesUsed() << endl
                << “Solution     : “ << endl
                << _dispatcher << endl;
}

// Solving
void RoutingSolver::improveWithNHood(IloInt nbIter) {
  IloNHood nhood = IloTwoOpt(_env)
                 + IloOrOpt(_env)
                 + IloRelocate(_env)
                 + IloExchange(_env)
                 + IloCross(_env)
                 + IloSwapPerform(_env);

   greedyImprove(nhood, _instantiateCost);
   improveWithFastGLS(nhood, nbIter, _instantiateCost);
}

void RoutingSolver::displayPaths(IloDispatcherGraph graph) {
  _env.out() << “Paths” << endl;
  for (IloIterator<IloVehicle> iter1(_env); iter1.ok(); ++iter1) {
    IloVehicle vehicle = *iter1;
    IloVisit visit1 = vehicle.getFirstVisit();
    for (IloDispatcher::RouteIterator iter2(_dispatcher, vehicle);
         iter2.ok();
         ++iter2) {
      IloVisit visit2 = *iter2;
      if (visit1 != visit2) {
        _env.out() << visit1.getName() << “ {“
                  <<  graph.getLocation(visit1.getStartNode())
                                .getIndex();
        for (IloDispatcherGraph::PathIterator iter3(graph,
                                                  visit1.getEndNode(),
                                                  visit2.getStartNode(),
                                                  vehicle);
             iter3.ok();
             ++iter3) {
          _env.out() << “ -> “ << (*iter3).getIndex();
        }
        _env.out() << “} -> “;
        visit1 = visit2;
      }
    }
    _env.out() << vehicle.getLastVisit().getName() << endl;
  }
  _env.out() << endl;
}

///////////////////////////////////////////////////////////////////////////////
int main(int argc, char * argv[]) {
  IloEnv env;
  try {
    RoutingModel mdl(env, argc, argv);
    RoutingSolver solver(mdl);
    if (solver.findFirstSolution()) {
      solver.printInformation();
      solver.improveWithNHood(250);
      solver.modifyFilterLevelAndRestore();
      solver.printInformation();
      solver.displayPaths(mdl.getGraph());
      mdl.modifyGraphArcCost();
      solver.restoreSolution();
      solver.improveWithNHood(50);
      solver.modifyFilterLevelAndRestore();
      solver.printInformation();
      solver.displayPaths(mdl.getGraph());
    }
  } catch(IloException& ex) {
    cerr << “Error: ” << ex << endl;
  }
  env.end();
  return 0;
}
```

Complete Output

```
//output

/**
8100.14
Number of fails               : 218
Number of choice points       : 1048
Number of variables           : 1550
Number of constraints         : 197
Reversible stack (bytes)      : 132684
Solver heap (bytes)           : 715840
Solver global heap (bytes)    : 83476
And stack (bytes)             : 20124
Or stack (bytes)              : 44244
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 11160
Total memory used (bytes)     : 1011572
Elapsed time since creation   : 1.602
Number of nodes               : 52
Number of visits              : 55
Number of vehicles            : 5
Number of dimensions          : 3
Number of accepted moves      : 0
===============
Cost         : 8100.14
Number of vehicles used : 3
Solution     :
Unperformed visits : visit2 visit3s visit4s visit6 visit7s visit9s visit10
visit11s visit12 visit14s visit15 visit17s visit19 visit20s visit22s visit23s
visit25 visit26
vehicle1 :
 -> depot weight[0..150] time[0] distance[0] -> visit16 weight[0..4]
time[0.308232..Inf) distance[15..Inf) -> visit19s weight[2..6]
time[3.00777..Inf) distance[61..Inf) -> visit25s weight[11..15]
time[4.14869..Inf) distance[71..Inf) -> visit17 weight[27..31]
time[5.00869..Inf) distance[72..Inf) -> visit13 weight[39..43]
time[5.96301..Inf) distance[82..Inf) -> visit7 weight[62..66]
time[7.40416..Inf) distance[97..Inf) -> visit27 weight[67..71]
time[7.90416..Inf) distance[98..Inf) -> visit6s weight[87..91]
time[8.80905..Inf) distance[105..Inf) -> visit24 weight[90..94]
```

```
time[9.05905..Inf) distance[106..Inf) -> visit4 weight[110..114]
time[9.70781..Inf) distance[111..Inf) -> visit2s weight[129..133]
time[11.265..Inf) distance[127..Inf) -> visit1 weight[136..140]
time[13.1035..Inf) distance[148..Inf) -> depot weight[146..150]
time[14.8287..Inf) distance[185..Inf)
vehicle2 :
 -> depot weight[0..150] time[0] distance[0] -> visit18 weight[0..6]
time[0.559723..Inf) distance[30..Inf) -> visit22 weight[17..23]
time[1.62321..Inf) distance[45..Inf) -> visit21 weight[42..48]
time[2.73992..Inf) distance[72..Inf) -> visit23 weight[65..71]
time[3.70439..Inf) distance[87..Inf) -> visit20 weight[80..86]
time[5.1382..Inf) distance[108..Inf) -> visit26s weight[100..106]
time[6.10098..Inf) distance[122..Inf) -> visit9 weight[119..125]
time[7.15098..Inf) distance[123..Inf) -> visit8 weight[135..141]
time[8.01789..Inf) distance[133..Inf) -> depot weight[144..150]
time[9.35878..Inf) distance[158..Inf)
vehicle3 :
 -> depot weight[0..150] time[0] distance[0] -> visit15s weight[0..36]
time[0.380972..Inf) distance[14..Inf) -> visit14 weight[8..44]
time[1.97156..Inf) distance[25..Inf) -> visit12s weight[28..64]
time[3.86793..Inf) distance[33..Inf) -> visit11 weight[47..83]
time[5.03332..Inf) distance[45..Inf) -> visit3 weight[59..95]
time[6.29017..Inf) distance[60..Inf) -> visit5 weight[72..108]
time[7.87694..Inf) distance[83..Inf) -> visit10s weight[98..134]
time[9.99977..Inf) distance[127..Inf) -> depot weight[114..150]
time[11.2586..Inf) distance[148..Inf)
vehicle4 : Unused
vehicle5 : Unused
Improving solution
Improving with first-accept GLS
---
Cost = 7781.68
Cost = 7712.67
Cost = 7616.17
Cost = 7592.99
---
---
Cost = 7616.02
Cost = 7549.36
Cost = 7573.36
Cost = 7502.99
Cost = 7479.5
---
Cost = 7456.5
---
Cost = 7552.36
Cost = 7529.36
Cost = 7505.91
Cost = 7482.26
Cost = 7458.55
Cost = 7433.43
---
Cost = 7456.43
---
Cost = 7433.43
Cost = 7433.43
---
Cost = 7456.43
```

```
---
Cost = 7433.43
---
Cost = 7575.43
Cost = 7551.78
Cost = 7528.07
Cost = 7502.95
---
Cost = 7479.95
---
Cost = 7526.15
---
Cost = 7549.15
Cost = 7618.33
Cost = 7505.86
Cost = 7482.68
Cost = 7482.68
---
Cost = 7505.73
---
Cost = 7482.68
---
Cost = 7436.48
Cost = 7413.03
---
Cost = 7459.74
Cost = 7459.72
Cost = 7459.66
---
Cost = 7530.93
Cost = 7530.68
Cost = 7513.94
Cost = 7490.93
---
Cost = 7514.11
---
Cost = 7632.87
Cost = 7609.22
Cost = 7585.51
Cost = 7560.39
Cost = 7560.39
---
Cost = 7652.79
Cost = 7652.4
Cost = 7536.94
Cost = 7629.74
Cost = 7629.59
Cost = 7627.35
Cost = 7649.45
Cost = 7625.07
Cost = 7581.32
Cost = 7558.23
Cost = 7558.16
---
Cost = 7558.99
Cost = 7535.9
Cost = 7512.85
Cost = 7489.67
Cost = 7489.32
Cost = 7489.24
---
Cost = 7512.29
---
Cost = 7489.24
---
---
Cost = 7627.91
Cost = 7696
Cost = 7672.91
Cost = 7649.86
Cost = 7626.83
Cost = 7626.48
Cost = 7626.4
---
Cost = 7558.16
---
Cost = 7581.21
---
Cost = 7558.16
---
Cost = 7558.99
Cost = 7535.9
Cost = 7581.25
---
Cost = 7535.9
Cost = 7512.85
Cost = 7512.5
Cost = 7512.42
---
Cost = 7604.73
Cost = 7581.64
Cost = 7558.59
Cost = 7558.23
---
Cost = 7626.56
Cost = 7602.18
Cost = 7602.11
---
Cost = 7738.97
Cost = 7715.52
---
Cost = 7738.99
Cost = 7625.16
Cost = 7697
Cost = 7696.8
Cost = 7652.02
Cost = 7605.16
Cost = 7582.11
Cost = 7581.99
Cost = 7558.28
---
Cost = 7581.46
Cost = 7649.54
---
Cost = 7581.46
---
```

```
Cost = 7558.28
Cost = 7558.28
---
Cost = 7581.46
---
Cost = 7650.46
Cost = 7650.29
Cost = 7558.28
---
Cost = 7627.13
Cost = 7603.84
---
Cost = 7626.9
Cost = 7581.39
Cost = 7558.11
---
Cost = 7561.15
Cost = 7538.1
Cost = 7514.92
Cost = 7491.21
Cost = 7466.09
Cost = 7466.1
---
Cost = 7536.55
Cost = 7515.17
Cost = 7514.81
Cost = 7514.74
Cost = 7491.1
Cost = 7465.98
---
---
Cost = 7604.65
Cost = 7581.6
Cost = 7558.42
Cost = 7558.06
Cost = 7558.06
---
Cost = 7626.18
Cost = 7601.8
Cost = 7601.73
---
Cost = 7579.64
---
Cost = 7603.51
Cost = 7580.46
Cost = 7557.28
Cost = 7533.22
Cost = 7533.23
---
Cost = 7509.93
---
Cost = 7670.08
Cost = 7646.64
Cost = 7556.28
---
Cost = 7646.64
---
Cost = 7623.28
Cost = 7556.28
Cost = 7533.54
Cost = 7510.49
---
Cost = 7532.81
Cost = 7512.71
Cost = 7489.07
---
Cost = 7557.27
Cost = 7532.88
Cost = 7532.81
---
Cost = 7649.13
Cost = 7672.02
Cost = 7648.97
Cost = 7625.79
Cost = 7625.79
---
Cost = 7697.6
Cost = 7652.79
Cost = 7629.5
Cost = 7582.3
Cost = 7581.94
Cost = 7581.82
---
Cost = 7650.54
Cost = 7626.9
---
Cost = 7603.84
---
Cost = 7620.59
---
Cost = 7572.57
Cost = 7572.36
Cost = 7525.99
Cost = 7525.94
---
Cost = 7572.65
Cost = 7572.64
Cost = 7731.5
Cost = 7708.5
Cost = 7593.25
Cost = 7570.03
Cost = 7569.96
---
Cost = 7546.96
Cost = 7546.96
---

Number of fails              : 0
Number of choice points      : 0
Number of variables          : 2086
Number of constraints        : 601
Reversible stack (bytes)     : 221124
Solver heap (bytes)          : 1636420
Solver global heap (bytes)   : 99556
And stack (bytes)            : 20124
Or stack (bytes)             : 44244
```

```
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 18172
Total memory used (bytes)     : 2043684
Elapsed time since creation   : 0.13
Number of nodes               : 52
Number of visits              : 55
Number of vehicles            : 5
Number of dimensions          : 3
Number of accepted moves      : 193
===============
Cost         : 7413.03
Number of vehicles used : 3
Solution     :
Unperformed visits : visit2 visit3s visit4s visit6 visit7s visit9s visit10
visit11 visit12 visit14s visit15 visit17s visit19 visit20 visit22s visit23s
visit25 visit26
vehicle1 :
 -> depot weight[0..150] time[0] distance[0] -> visit25s weight[0..2]
time[0.503581..0.503581] distance[21..21] -> visit17 weight[16..18]
time[1.36358..1.36358] distance[22..22] -> visit13 weight[28..30]
time[2.3179..2.3179] distance[32..32] -> visit6s weight[51..53]
time[3.68681..3.68681] distance[43..43] -> visit24 weight[54..56]
time[3.93681..3.93681] distance[44..44] -> visit4 weight[74..76]
time[4.58557..4.58557] distance[49..49] -> visit7 weight[93..95]
time[5.97477..5.97477] distance[55..55] -> visit27 weight[98..100]
time[6.47477..6.47477] distance[56..56] -> visit2s weight[118..120]
time[7.51843..7.51843] distance[71..71] -> visit1 weight[125..127]
time[9.35692..9.35692] distance[92..92] -> visit3 weight[135..137]
time[10.6721..10.6721] distance[111..111] -> depot weight[148..150]
time[12.2627..12.2627] distance[129..129]
vehicle2 :
 -> depot weight[0..150] time[0] distance[0] -> visit15s weight[0..5]
time[0.380972..0.380972] distance[14..14] -> visit14 weight[8..13]
time[1.97156..1.97156] distance[25..25] -> visit12s weight[28..33]
time[3.86793..3.86793] distance[33..33] -> visit11s weight[47..52]
time[5.01769..5.01769] distance[44..44] -> visit5 weight[59..64]
time[6.54135..6.54135] distance[75..75] -> visit16 weight[85..90]
time[8.25959..8.25959] distance[95..95] -> visit18 weight[87..92]
time[10.3843..10.3843] distance[110..110] -> visit22 weight[104..109]
time[11.4478..11.4478] distance[125..125] -> visit10s weight[129..134]
time[12.5504..12.5504] distance[151..151] -> depot weight[145..150]
time[13.8091..13.8091] distance[172..172]
vehicle3 :
 -> depot weight[0..150] time[0] distance[0] -> visit21 weight[0..39]
time[0.589454..0.589454] distance[30..30] -> visit23 weight[23..62]
time[1.55392..1.55392] distance[45..45] -> visit19s weight[38..77]
time[2.91295..2.91295] distance[61..61] -> visit20s weight[47..86]
time[4.16945..4.16945] distance[79..79] -> visit26s weight[67..106]
time[5.1485..5.1485] distance[94..94] -> visit9 weight[86..125]
time[6.1985..6.1985] distance[95..95] -> visit8 weight[102..141]
time[7.0654..7.0654] distance[105..105] -> depot weight[111..150]
time[8.4063..8.4063] distance[130..130]
vehicle4 : Unused
vehicle5 : Unused
Paths
depot {1995 -> 1995 -> 2051 -> 2107 -> 2163 -> 2219 -> 2275 -> 2331 -> 2330 ->
2329 -> 2328 -> 2327 -> 2326 -> 2325 -> 2324 -> 2323 -> 2322 -> 2321 -> 2320 ->
2319 -> 2318 -> 2317 -> 2316} -> visit25s {2316 -> 2260} -> visit17 {2260 ->
2134 -> 2133 -> 2077 -> 2021 -> 1965 -> 1909 -> 1853 -> 1797 -> 1741 -> 1685} -
> visit13 {1685 -> 1684 -> 1628 -> 1572 -> 1516 -> 1460 -> 1404 -> 1348 -> 1292
-> 1236 -> 1237 -> 1238} -> visit6s {1238 -> 1182} -> visit24 {1182 -> 1126 ->
1127 -> 1128 -> 1129 -> 1130} -> visit4 {1130 -> 1186 -> 1242 -> 1298 -> 1354 -
> 1355 -> 1356} -> visit7 {1356 -> 1300} -> visit27 {1300 -> 1244 -> 1188 ->
1132 -> 1076 -> 1020 -> 964 -> 908 -> 852 -> 796 -> 740 -> 684 -> 628 -> 629 ->
630 -> 631} -> visit2s {631 -> 575 -> 519 -> 463 -> 407 -> 351 -> 295 -> 296 ->
297 -> 298 -> 299 -> 300 -> 301 -> 302 -> 303 -> 304 -> 305 -> 306 -> 307 ->
308 -> 309 -> 310} -> visit1 {310 -> 254 -> 255 -> 256 -> 257 -> 258 -> 259 ->
315 -> 371 -> 427 -> 483 -> 539 -> 595 -> 651 -> 707 -> 763 -> 819 -> 875 ->
931 -> 987} -> visit3 {987 -> 1043 -> 1099 -> 1155 -> 1211 -> 1267 -> 1323 ->
1379 -> 1435 -> 1491 -> 1547 -> 1603 -> 1659 -> 1715 -> 1771 -> 1827 -> 1883 ->
1939 -> 1995} -> depot
depot {1995 -> 1995 -> 1939 -> 1883 -> 1827 -> 1771 -> 1715 -> 1714 -> 1713 ->
1712 -> 1711 -> 1710 -> 1709 -> 1708 -> 1707 -> 1706} -> visit15s {1706 -> 1705
-> 1704 -> 1703 -> 1702 -> 1701 -> 1700 -> 1699 -> 1698 -> 1697 -> 1696 ->
1695} -> visit14 {1695 -> 1639 -> 1583 -> 1527 -> 1528 -> 1529 -> 1530 -> 1531
-> 1532} -> visit12s {1532 -> 1476 -> 1477 -> 1478 -> 1479 -> 1480 -> 1481 ->
1482 -> 1483 -> 1484 -> 1485 -> 1486} -> visit11s {1486 -> 1430 -> 1431 -> 1432
-> 1433 -> 1434 -> 1435 -> 1436 -> 1437 -> 1438 -> 1439 -> 1440 -> 1441 -> 1442
-> 1443 -> 1444 -> 1445 -> 1446 -> 1447 -> 1448 -> 1449 -> 1450 -> 1451 -> 1452
-> 1453 -> 1454 -> 1455 -> 1399 -> 1343 -> 1287 -> 1231 -> 1175} -> visit5
{1175 -> 1231 -> 1287 -> 1343 -> 1399 -> 1455 -> 1511 -> 1567 -> 1623 -> 1679 -
> 1735 -> 1791 -> 1847 -> 1903 -> 1959 -> 2015 -> 2014 -> 2013 -> 2012 -> 2011
-> 2010} -> visit16 {2010 -> 2066 -> 2067 -> 2068 -> 2069 -> 2070 -> 2071 ->
2127 -> 2183 -> 2239 -> 2295 -> 2351 -> 2407 -> 2463 -> 2519 -> 2575} ->
visit18 {2575 -> 2631 -> 2687 -> 2743 -> 2799 -> 2855 -> 2911 -> 2967 -> 3023 -
> 3079 -> 3135 -> 3191 -> 3247 -> 3303 -> 3359 -> 3415} -> visit22 {3415 ->
3471 -> 3470 -> 3414 -> 3358 -> 3302 -> 3246 -> 3190 -> 3134 -> 3078 -> 3022 ->
2966 -> 2910 -> 2854 -> 2853 -> 2852 -> 2851 -> 2850 -> 2849 -> 2848 -> 2847 ->
2846 -> 2845 -> 2844 -> 2843 -> 2842 -> 2841} -> visit10s {2841 -> 2785 -> 2729
-> 2673 -> 2617 -> 2561 -> 2505 -> 2449 -> 2393 -> 2337 -> 2281 -> 2225 -> 2169
-> 2113 -> 2057 -> 2001 -> 2000 -> 1999 -> 1998 -> 1997 -> 1996 -> 1995} ->
depot
depot {1995 -> 1995 -> 2051 -> 2107 -> 2163 -> 2219 -> 2275 -> 2331 -> 2387 ->
2443 -> 2499 -> 2555 -> 2611 -> 2667 -> 2723 -> 2779 -> 2835 -> 2891 -> 2947 ->
3003 -> 3059 -> 3115 -> 3171 -> 3227 -> 3283 -> 3339 -> 3395 -> 3394 -> 3393 ->
3392 -> 3391 -> 3390} -> visit21 {3390 -> 3389 -> 3445 -> 3501 -> 3557 -> 3613
-> 3669 -> 3668 -> 3667 -> 3666 -> 3665 -> 3664 -> 3663 -> 3662 -> 3661 ->
3660} -> visit23 {3660 -> 3659 -> 3660 -> 3604 -> 3548 -> 3492 -> 3436 -> 3380
-> 3324 -> 3268 -> 3212 -> 3156 -> 3100 -> 3044 -> 2988 -> 2932 -> 2876} ->
visit19s {2876 -> 2820 -> 2876 -> 2932 -> 2988 -> 3044 -> 3100 -> 3156 -> 3155
-> 3154 -> 3153 -> 3152 -> 3151 -> 3150 -> 3149 -> 3148 -> 3147 -> 3146 ->
3145} -> visit20s {3145 -> 3144 -> 3088 -> 3032 -> 2976 -> 2920 -> 2864 -> 2808
-> 2752 -> 2696 -> 2640 -> 2584 -> 2528 -> 2472 -> 2473 -> 2474} -> visit26s
{2474 -> 2418} -> visit9 {2418 -> 2362 -> 2306 -> 2250 -> 2194 -> 2138 -> 2082
-> 2083 -> 2084 -> 2085 -> 2086} -> visit8 {2086 -> 2142 -> 2143 -> 2144 ->
2145 -> 2146 -> 2147 -> 2148 -> 2149 -> 2150 -> 2151 -> 2152 -> 2153 -> 2154 ->
2155 -> 2156 -> 2157 -> 2158 -> 2159 -> 2160 -> 2161 -> 2162 -> 2163 -> 2107 ->
2051 -> 1995} -> depot
depot {1995 -> 1995} -> depot
depot {1995 -> 1995} -> depot
Modified cost: 8311.26
Improving solution
Improving with first-accept GLS
---
Cost = 7506.89
Cost = 7482.83
```

```
---
---
Cost = 7483
---
Cost = 7482.83
---
Cost = 7505.88
---
Cost = 7482.83
---
Cost = 7529.53
Cost = 7529.52
Cost = 7529.45
---
Cost = 7600.73
Cost = 7600.48
Cost = 7583.73
Cost = 7655.14
Cost = 7632.13
---
Cost = 7724.91
---
Cost = 7779.65
Cost = 7751.15
Cost = 7632.13
Cost = 7560.73
---
Cost = 7560.9
Cost = 7537.45
---
Cost = 7560.63
---
Cost = 7679.39
Cost = 7655.94
Cost = 7632.29
Cost = 7608.58
Cost = 7583.46
Cost = 7583.46
---
Cost = 7629.59
Cost = 7627.35
Cost = 7649.45
Cost = 7625.07
Cost = 7581.32
Cost = 7558.23
Cost = 7558.16
Cost = 7626.4

Number of fails               : 0
Number of choice points       : 0
Number of variables           : 2086
Number of constraints         : 601
Reversible stack (bytes)      : 221124
Solver heap (bytes)           : 1636420
Solver global heap (bytes)    : 99556
And stack (bytes)             : 20124
Or stack (bytes)              : 44244
Search Stack (bytes)          : 4044
```

```
Constraint queue (bytes)      : 18172
Total memory used (bytes)     : 2043684
Elapsed time since creation   : 0.12
Number of nodes               : 52
Number of visits              : 55
Number of vehicles            : 5
Number of dimensions          : 3
Number of accepted moves      : 230
===============
Cost          : 7459.55
Number of vehicles used : 3
Solution      :
Unperformed visits : visit2 visit3 visit4s visit6 visit7s visit9s visit10
visit11 visit12 visit14s visit15 visit17s visit19 visit20 visit22s visit23s
visit25 visit26
vehicle1 :
 -> depot weight[0..150] time[0] distance[0] -> visit25s weight[0..2]
time[0.503581..0.503581] distance[21..21] -> visit17 weight[16..18]
time[1.36358..1.36358] distance[22..22] -> visit13 weight[28..30]
time[2.3179..2.3179] distance[32..32] -> visit6s weight[51..53]
time[3.68681..3.68681] distance[43..43] -> visit24 weight[54..56]
time[3.93681..3.93681] distance[44..44] -> visit4 weight[74..76]
time[4.58557..4.58557] distance[49..49] -> visit7 weight[93..95]
time[5.97477..5.97477] distance[55..55] -> visit27 weight[98..100]
time[6.47477..6.47477] distance[56..56] -> visit2s weight[118..120]
time[7.51843..7.51843] distance[71..71] -> visit1 weight[125..127]
time[9.35692..9.35692] distance[92..92] -> visit3s weight[135..137]
time[10.6915..10.6915] distance[112..112] -> depot weight[148..150]
time[12.3351..12.3351] distance[133..133]
vehicle2 :
 -> depot weight[0..150] time[0] distance[0] -> visit15s weight[0..5]
time[0.380972..0.380972] distance[14..14] -> visit14 weight[8..13]
time[1.97156..1.97156] distance[25..25] -> visit12s weight[28..33]
time[3.86793..3.86793] distance[33..33] -> visit11s weight[47..52]
time[5.01769..5.01769] distance[44..44] -> visit5 weight[59..64]
time[6.54135..6.54135] distance[75..75] -> visit16 weight[85..90]
time[8.25959..8.25959] distance[95..95] -> visit18 weight[87..92]
time[10.3843..10.3843] distance[110..110] -> visit22 weight[104..109]
time[11.4478..11.4478] distance[125..125] -> visit10s weight[129..134]
time[12.5504..12.5504] distance[151..151] -> depot weight[145..150]
time[13.8091..13.8091] distance[172..172]
vehicle3 :
 -> depot weight[0..150] time[0] distance[0] -> visit21 weight[0..39]
time[0.589454..0.589454] distance[30..30] -> visit23 weight[23..62]
time[1.55392..1.55392] distance[45..45] -> visit19s weight[38..77]
time[2.91295..2.91295] distance[61..61] -> visit20s weight[47..86]
time[4.16945..4.16945] distance[79..79] -> visit26s weight[67..106]
time[5.1485..5.1485] distance[94..94] -> visit9 weight[86..125]
time[6.1985..6.1985] distance[95..95] -> visit8 weight[102..141]
time[7.0654..7.0654] distance[105..105] -> depot weight[111..150]
time[8.4063..8.4063] distance[130..130]
vehicle4 : Unused
vehicle5 : Unused
Paths
depot {1995 -> 1995 -> 2051 -> 2107 -> 2163 -> 2219 -> 2275 -> 2331 -> 2330 ->
2329 -> 2328 -> 2327 -> 2326 -> 2325 -> 2324 -> 2323 -> 2322 -> 2321 -> 2320 ->
2319 -> 2318 -> 2317 -> 2316} -> visit25s {2316 -> 2260} -> visit17 {2260 ->
2134 -> 2133 -> 2077 -> 2021 -> 1965 -> 1909 -> 1853 -> 1797 -> 1741 -> 1685} -
```

```
> visit13 {1685 -> 1684 -> 1628 -> 1572 -> 1516 -> 1460 -> 1404 -> 1348 -> 1292
-> 1236 -> 1237 -> 1238} -> visit6s {1238 -> 1182} -> visit24 {1182 -> 1126 ->
1127 -> 1128 -> 1129 -> 1130} -> visit4 {1130 -> 1186 -> 1242 -> 1298 -> 1354 -
> 1355 -> 1356} -> visit7 {1356 -> 1300} -> visit27 {1300 -> 1244 -> 1188 ->
1132 -> 1076 -> 1020 -> 964 -> 908 -> 852 -> 796 -> 740 -> 684 -> 628 -> 629 ->
630 -> 631} -> visit2s {631 -> 575 -> 519 -> 463 -> 407 -> 351 -> 295 -> 296 ->
297 -> 298 -> 299 -> 300 -> 301 -> 302 -> 303 -> 304 -> 305 -> 306 -> 307 ->
308 -> 309 -> 310} -> visit1 {310 -> 254 -> 255 -> 256 -> 257 -> 258 -> 259 ->
315 -> 371 -> 427 -> 483 -> 539 -> 595 -> 651 -> 707 -> 763 -> 819 -> 875 ->
931 -> 987 -> 1043} -> visit3s {1043 -> 987 -> 988 -> 1044 -> 1100 -> 1156 ->
1212 -> 1268 -> 1324 -> 1380 -> 1436 -> 1492 -> 1548 -> 1604 -> 1660 -> 1716 ->
1772 -> 1828 -> 1884 -> 1940 -> 1996 -> 1995} -> depot
depot {1995 -> 1995 -> 1939 -> 1883 -> 1827 -> 1771 -> 1715 -> 1714 -> 1713 ->
1712 -> 1711 -> 1710 -> 1709 -> 1708 -> 1707 -> 1706} -> visit15s {1706 -> 1705
-> 1704 -> 1703 -> 1702 -> 1701 -> 1700 -> 1699 -> 1698 -> 1697 -> 1696 ->
1695} -> visit14 {1695 -> 1639 -> 1583 -> 1527 -> 1528 -> 1529 -> 1530 -> 1531
-> 1532} -> visit12s {1532 -> 1476 -> 1477 -> 1478 -> 1479 -> 1480 -> 1481 ->
1482 -> 1483 -> 1484 -> 1485 -> 1486} -> visit11s {1486 -> 1430 -> 1431 -> 1432
-> 1433 -> 1434 -> 1435 -> 1436 -> 1437 -> 1438 -> 1439 -> 1440 -> 1441 -> 1442
-> 1443 -> 1444 -> 1445 -> 1446 -> 1447 -> 1448 -> 1449 -> 1450 -> 1451 -> 1452
-> 1453 -> 1454 -> 1455 -> 1399 -> 1343 -> 1287 -> 1231 -> 1175} -> visit5
{1175 -> 1231 -> 1287 -> 1343 -> 1399 -> 1455 -> 1511 -> 1567 -> 1623 -> 1679 -
> 1735 -> 1791 -> 1847 -> 1903 -> 1959 -> 2015 -> 2014 -> 2013 -> 2012 -> 2011
-> 2010} -> visit16 {2010 -> 2066 -> 2067 -> 2068 -> 2069 -> 2070 -> 2071 ->
2127 -> 2183 -> 2239 -> 2295 -> 2351 -> 2407 -> 2463 -> 2519 -> 2575} ->
visit18 {2575 -> 2631 -> 2687 -> 2743 -> 2799 -> 2855 -> 2911 -> 2967 -> 3023 -
> 3079 -> 3135 -> 3191 -> 3247 -> 3303 -> 3359 -> 3415} -> visit22 {3415 ->
3471 -> 3470 -> 3414 -> 3358 -> 3302 -> 3246 -> 3190 -> 3134 -> 3078 -> 3022 ->
2966 -> 2910 -> 2854 -> 2853 -> 2852 -> 2851 -> 2850 -> 2849 -> 2848 -> 2847 ->
2846 -> 2845 -> 2844 -> 2843 -> 2842 -> 2841} -> visit10s {2841 -> 2785 -> 2729
-> 2673 -> 2617 -> 2561 -> 2505 -> 2449 -> 2393 -> 2337 -> 2281 -> 2225 -> 2169
-> 2113 -> 2057 -> 2001 -> 2000 -> 1999 -> 1998 -> 1997 -> 1996 -> 1995} ->
depot
depot {1995 -> 1995 -> 2051 -> 2107 -> 2163 -> 2219 -> 2275 -> 2331 -> 2387 ->
2443 -> 2499 -> 2555 -> 2611 -> 2667 -> 2723 -> 2779 -> 2835 -> 2891 -> 2947 ->
3003 -> 3059 -> 3115 -> 3171 -> 3227 -> 3283 -> 3339 -> 3395 -> 3394 -> 3393 ->
3392 -> 3391 -> 3390} -> visit21 {3390 -> 3389 -> 3445 -> 3501 -> 3557 -> 3613
-> 3669 -> 3668 -> 3667 -> 3666 -> 3665 -> 3664 -> 3663 -> 3662 -> 3661 ->
3660} -> visit23 {3660 -> 3659 -> 3660 -> 3604 -> 3548 -> 3492 -> 3436 -> 3380
-> 3324 -> 3268 -> 3212 -> 3156 -> 3100 -> 3044 -> 2988 -> 2932 -> 2876} ->
visit19s {2876 -> 2820 -> 2876 -> 2932 -> 2988 -> 3044 -> 3100 -> 3156 -> 3155
-> 3154 -> 3153 -> 3152 -> 3151 -> 3150 -> 3149 -> 3148 -> 3147 -> 3146 ->
3145} -> visit20s {3145 -> 3144 -> 3088 -> 3032 -> 2976 -> 2920 -> 2864 -> 2808
-> 2752 -> 2696 -> 2640 -> 2584 -> 2528 -> 2472 -> 2473 -> 2474} -> visit26s
{2474 -> 2418} -> visit9 {2418 -> 2362 -> 2306 -> 2250 -> 2194 -> 2138 -> 2082
-> 2083 -> 2084 -> 2085 -> 2086} -> visit8 {2086 -> 2142 -> 2143 -> 2144 ->
2145 -> 2146 -> 2147 -> 2148 -> 2149 -> 2150 -> 2151 -> 2152 -> 2153 -> 2154 ->
2155 -> 2156 -> 2157 -> 2158 -> 2159 -> 2160 -> 2161 -> 2162 -> 2163 -> 2107 ->
2051 -> 1995} -> depot
depot {1995 -> 1995} -> depot
depot {1995 -> 1995} -> depot

*/
```

Part IV

Developing Dispatcher Applications

This part consists of the following lessons:

- Chapter 16, *Designing Dispatcher Models*
- Chapter 17, *Developing Your Own First Solution Heuristics*\
- Chapter 18, *Developing Your Own Neighborhoods*

C H A P T E R

16

Designing Dispatcher Models

In this lesson, you will learn how to:

- simplify a model
- decompose a problem
- use other modeling hints

This chapter offers a few design principles gleaned from experienced users of Dispatcher. These principles are meant to help you to avoid the errors often made by new users of Dispatcher when they design a model for a problem. Of course, not every problem benefits from a mechanical application of every principle mentioned in this lesson, but in general these principles should help you develop robust and efficient Dispatcher examples.

The most important modeling hints are to begin by simplifying your problem and to decompose a large problem. See “Simplify the model” on page 395 and “Decompose the problem” on page 396. For other assorted modeling hints, see “Other modeling hints” on page 406.

Simplify the model

When you begin developing your Dispatcher model, the most important thing you can do is to simplify the model. If you start by trying to design a model for a complex problem with many constraints and a complex cost function, you will be frustrated. You need to understand where the complexity in your problem is coming from before you can add it to the model. If you do not, the solution may have a poor quality. After you have developed your simplified model and found a first solution, you can progressively add the complexity back to your model.

Try to get your problem to resemble, as closely as possible, one of the distributed examples. For a vehicle routing problem, try to simplify your problem until it resembles `vrp.cpp`. For a pickup-and-delivery problem, do the same with `pdp.cpp` and likewise for an arc routing problem and `carp.cpp`.

How to simplify a problem? Try the following:

- Remove as many constraints as possible, including breaks, variable delays, extra dimensions, and so on. You can keep capacity, time windows, and pickup-and-delivery constraints in your simplified model.
- Remove alternative depots, and so on.
- Remove Dispatcher graph functionality.
- Simplify the cost function. One useful idea is to linearize the cost function and overestimate the cost.

Suggested procedure

The following is a suggested procedure for designing a Dispatcher application:

1. Prepare a data set for which you already have a solution and fit it into the csv data format.
2. Simplify the model following the suggestions in this section.
3. Use a simplified cost function to find a solution and then compute the real cost afterwards. Compare this real cost to the cost of existing solutions.
4. Add the removed constraints one-by-one back to the model, moving from the simplest to the most complex. In parallel, restore the complexity to your cost function.

Decompose the problem

There are two main ways to decompose a problem: geographical and temporal. An example of a geographical decomposition is to preallocate to depots and to create subproblems for each depot. An example of a temporal decomposition is a problem where technicians return to their base every night. You create subproblems for each day.

Why decompose a problem? The complexity of Dispatcher is $O(n^2)$. This empirical complexity varies, at the very least, with the square of the number of visits. If you have n visits and k subproblems, you have n/k visits per problem. Therefore, if you decompose the problem the complexity is reduced to

$$k(n/k)^2 = n^2/k$$

You do pay a price by decomposing the problem. The solution will not be as good as one found for the whole problem. However, you can overcome this limitation by using an iterative solution process. Each subproblem is improved and then the whole problem is improved. This gives you the performance advantages of problem decomposition, while still allowing you to find a good overall solution.

Into how many subproblems should you decompose your problem? It is most useful to decompose your problem into logical subproblems based on your problem: number of depots, number of days, number of weeks, and so on. Even if you decompose into only 2-5 subproblems, you will already see substantial performance advantages.

In Lesson 10, *Pickup and Delivery by Multiple Vehicles from Multiple Depots*, you performed a geographical decomposition. You solved subproblems for each depot and then solved the whole problem. You will use this same technique in this section to solve a temporal decomposition problem. In this problem, you will solve a technician dispatching problem by decomposing the problem into subproblems for each day.

Note: *It is important to be careful of the connections between subproblems in temporal decompositions. You must be aware of how the days or weeks connect when the whole problem is solved.*

Temporal decomposition

To show how to use temporal decomposition, you will solve a technician dispatching problem. As in Chapter 13, *Dispatching Technicians*, you will model and solve this problem using `RoutingModel` and `RoutingSolver` classes. There are 5 technicians who must perform 50 visits over 3 days. There are 5 different skill levels, depending on technician. Certain visits require technicians of a certain skill level. A `SkillCosts` class is used to model costs and maximize the quality of service. If a technician type is well suited to performing a certain visit, then the cost for that technician/visit pair should be low. If a technician type is not well suited to performing a visit, the cost for that technician/visit pair should be high.

You will also create a `Day` class to create submodels of the problem for each of the 3 days. One model, an instance of `IloModel`, per day is maintained. Each of these models contains the dimensions of the problem and the technicians assigned to work that day. A model of the whole problem, containing the day submodels, is also maintained.

The solution process is iterative. Each day submodel is improved and then the whole problem is improved. This latter phase is one that can cause visits to migrate from one day to another. In other words, moves can be performed that cause a technician to perform a visit on a different day. When the next iteration to improve days begins, a synchronization needs to be performed. During this synchronization, the submodels are updated with the changes made during the improvement stage of the whole problem.

Only the parts of this example that are changed from Chapter 13, *Dispatching Technicians* are shown here. You can view the complete program and output online in the `YourDispatcherHome/examples/src/technicianDispatching.cpp` file.

Declare the Day class

The code for the declaration of the `Day` class follows:

```
class Day {
private:
  IloEnv            _env;
  IloNode           _node;
  IloInt            _nbOfTechnicians;
  IloInt            _capacity;
  IloNum            _openTime;
  IloNum            _closeTime;
  IloVehicleArray   _technicians;
  IloVisitArray     _visits;
  IloModel          _model;
  IloNHood          _nhood;
  IloMetaHeuristic  _mh;
  IloInt            _index;
  char *            _name;

public:
  Day(IloEnv env,
      IloInt index,
      const char *name,
      IloNode node,
      IloInt nbOfTechnicians,
      IloInt capacity,
      IloNum openTime,
      IloNum closeTime);
  ~Day();

  const char* getName() const { return _name;}
  void add(IloExtractable ex) { _model.add(ex); }
  void add(IloVehicle technician) {
    _technicians.add(technician);
    _model.add(technician);
  }
  IloInt getIndex() const { return _index; }
  IloModel getModel() const { return _model; }
  IloVehicleArray getTechnicians() const { return _technicians; }
  IloBool improve(IloSolver solver, IloRoutingSolution rs, IloGoal g);
  void fillModel(IloRoutingSolution rs);

  void createTechnicians(IloDimension2 time, IloDimension1 skillPenalty);
  IloVehicle createOneTechnician (IloInt technicianIndex,
                                  IloDimension2 time,
                                  IloDimension1 skillPenalty);
};
```

Define the Day constructor

A day is built from an instance of `IloEnv` and is given a name `name`:

```
Day::Day (IloEnv env,
          IloInt index,
          const char *name,
          IloNode node,
          IloInt nbOfTechnicians,
          IloInt capacity,
          IloNum openTime,
          IloNum closeTime)
  : _env(env),
    _node(node),
    _nbOfTechnicians(nbOfTechnicians),
    _capacity(capacity),
    _openTime(openTime),
    _closeTime(closeTime),
    _technicians(env),
    _visits(env),
    _model(env),
    _index(index) {
```

In order to improve solutions for this day, a neighborhood `_nhood` and a greedy metaheuristic `_mh` are created. As the subproblem of improving the solution for a single day is likely to be relatively small, five move operators are used in the neighborhood.

```
  _nhood = IloTwoOpt(env)
         + IloOrOpt(env)
         + IloRelocate(env)
         + IloCross(env)
         + IloExchange(env)
       //+ IloRelocateTours(env)
         ;
  _mh = IloImprove(env);
```

A day destructor is also defined:

```
Day::~Day() {
  _nhood.end();
  _mh.end();
  _technicians.end();
  _visits.end();
  _env.getImpl()->free(_name, sizeof(char) * (strlen(_name) + 1));
}
```

Define the Day::Improve function

You define a method `improve` that is used for improving the solution `rs` passed as an argument. A solver `solver` and a goal `g` to be executed at each move are also passed as parameters. The number of moves made is returned. First, the neighborhood and greedy metaheuristic are reset. Then, the improvement goal is created using `IloSingleMove`. An improvement loop is then entered, which stops when no move can improve the cost of the current solution. Finally, the number of moves made is returned.

```
IloBool Day::improve(IloSolver solver, IloRoutingSolution rs, IloGoal g) {
  _nhood.reset();
  _mh.reset();
  IloGoal improve = IloSingleMove(_env, rs, _nhood, _mh, g);
  solver.out() << "  Optimizing day " << getName() << " "
               << rs.getSolution().getObjectiveValue() << flush;
  IloBool moves = 0;
  while (solver.solve(improve)) ++moves;
  solver.out() << " ---> " << rs.getSolution().getObjectiveValue() << endl;
  return (moves>0);
}
```

Define the Day::fillModel function

The class `Day` also requires the ability to synchronize its internal model with a routing solution. To perform this you first empty the model of the visit models which previously comprised the model. You then scan the solution for the technicians who are working on this day, and for each one, find out which visits are made by these technicians. These visits (or more accurately, the models pertaining to these visits) must then be placed in the day model. You keep track of which visits are in the model in the array `_visits`.

```
void Day::fillModel(IloRoutingSolution rs) {
  IloInt i;
  for (i = 0; i < _visits.getSize(); i++)
    _model.remove((IloModelI *)_visits[i].getObject());
  _visits.end();
  _visits = IloVisitArray(_env);
  for (i = 0; i < _technicians.getSize(); i++) {
   for (IloRoutingSolution::RouteIterator r(rs, _technicians[i]); r.ok(); ++r)
{
      IloVisit visit = *r;
      if ( !visit.isFirstVisit() && !visit.isLastVisit() ) {
        _visits.add(visit);
        _model.add( (IloModelI*)visit.getObject() );
      }
    }
  }
}
```

Define the Day::createOneTechnician function

You add a function to create a technician associated to the day. The technicians have first and last visits each day. You add side constraints that the technicians must leave the depot after it opens and return to the depot before it closes. You set the cost of the technician. The cost of each technician is directly proportional to the dimension `time` and the `skillPenalty`.

```
IloVehicle Day::createOneTechnician(IloInt technicianIndex,
                                    IloDimension2 time,
                                    IloDimension1 skillPenalty) {
  char namebuf[128];
  const char* dayName = getName();

  IloVisit first(_node, "depot");
  IloVisit last (_node, "depot");
  sprintf(namebuf, "Technician %ld working on %s", technicianIndex, dayName);
  IloVehicle technician(first, last, namebuf);
  add(technician);
  add(first.getCumulVar(time) >= _openTime);
  add(last.getCumulVar(time)  <= _closeTime);
  add(first.getTransitVar(skillPenalty) == 0);
  add(last.getTransitVar(skillPenalty) == 0);
  technician.setCost(time, 1.0);
  technician.setCost(skillPenalty, 1.0);
  technician.setKey(namebuf);

  technician.setObject(this);
  return technician;
}
```

Define the Day::createTechnicians function

You create a function that will be called by `RoutingModel::createDays`. This loop will be used to create the technicians assigned to work each day.

```
void Day::createTechnicians(IloDimension2 time,
                            IloDimension1 skillPenalty) {
  for (IloInt v=0; v < _nbOfTechnicians; v++) {
    createOneTechnician(v, time, skillPenalty);
  }
}
```

Define the RoutingModel::createDays function

You use Concert Technology’s csv reader functionality to input day data from csv files. For each of the days, you get the number of technicians (`nbOfTechnicians`), the depot (`nodeName`), and its opening and closing times (`openTime` and `closeTime`). An array of pointers to class type `Day` are created. This array is then populated by a loop that creates the days. For each day, the function `Day::createTechnicians` is called to create the technicians associated with each day and the model containing all dimensions `_dimModel` is added to the day model. Finally, each day model is added to the model of the whole problem `_mdl`.

```
void RoutingModel::createDays(const char* daysFileName) {
  IloCsvReader dayReader(_env, daysFileName);
  _nbOfDays = dayReader.getNumberOfItems();
  _days = new (_env) Day* [_nbOfDays];

  IloInt dayIndex = 0;
  for (IloCsvReader::LineIterator it(dayReader); it.ok(); ++it, ++dayIndex) {
    IloCsvLine line = *it;
    const char* name = line.getStringByHeader("name");
    char * nodeName  = line.getStringByHeader("nodeName");
    IloNum openTime   = line.getFloatByHeader("openTime");
    IloNum closeTime  = line.getFloatByHeader("closeTime");
    IloInt nbOfTechnicians = line.getIntByHeader("nbOfTechnicians");
    IloInt capacity   = line.getIntByHeader("capacity");

    IloNode depot = IloNode::Find(_env, nodeName);

    Day* day = new (_env) Day(_env,
                              dayIndex,
                              name,
                              depot,
                              nbOfTechnicians,
                              capacity,
                              openTime,
                              closeTime);
    _days[dayIndex] = day;
  }
  dayReader.end();

  IloInt d;
  for (d=0; d < _nbOfDays; ++d) {
    Day* day = _days[d];
    day->createTechnicians(_time, _skillPenalty);
    day->add(_dimModel);
    _mdl.add(day->getModel());
  }

}
```

Define the createSkillCostFunction

The function `createSkillCostFunction` creates the vehicle array `technicians`. The `IloCsvReader` instance `csvVisitSkillsReader` reads the skills required at each customer visit; `csvTechSkillsReader` reads the skill level of each technician; and `csvSkillCostsReader` reads the cost for each technician skill type to perform a visit to a customer site.

The next section of `createSkillCostFunction` creates an array of the cost for each visit, named `skillCosts`. To create this array, the visit skill required for each visit (`VisitSkillName`) is read from a csv file. The cost to make this visit with the various `techSkill` levels is read with `csvSkillCostsReader` as an `IloCsvLine` `costline`.

Then the value in `costline` corresponding to the appropriate `TechSkillName` is stored in the array `skillCosts`.

Finally, the `IloVehicleToNumFunction` constructor associates the values of the array `skillCosts` to the vehicle/technicians of the array `technicians`.

```
IloVehicleToNumFunction
SkillCosts::createSkillCostFunction(Day* day,
                                    IloVisit visit,
                                    IloCsvReader csvVisitSkillsReader,
                                    IloCsvReader csvTechSkillsReader,
                                    IloCsvReader csvSkillCostsReader) {
  IloVehicleArray technicians = day->getTechnicians();
  IloInt size = technicians.getSize();
  IloNumArray skillCosts(_env, size);
  IloCsvLine line1 = csvVisitSkillsReader.getLineByKey(1, visit.getName());
  char* skillName = line1.getStringByHeader("VisitSkillName");
  IloCsvLine costline = csvSkillCostsReader.getLineByKey(1, skillName);
  for (IloInt  i = 0; i < size; i++) {
    IloVehicle technician = technicians[i];
    IloCsvLine line2 = csvTechSkillsReader.getLineByKey(1,
technician.getName());
    char * techSkillName = line2.getStringByHeader("TechSkillName");
    skillCosts[i] = costline.getFloatByHeader(techSkillName);
  }
  return IloVehicleToNumFunction(_env, technicians, skillCosts, 0);
}
```

Define the RoutingSolver::improveDays function

If a first solution is found, the solution is improved using local search. You iterate over all days, using `Day::fillModel` to populate each model. You then update the cost of the global solution according to the model of this day. You use `Day::improve` to attempt to reduce the cost of the routing for this day. Finally, the solution is synchronized with the global model.

```
IloBool RoutingSolver::improveDays() {
  IloEnv env = getEnv();
  _move =
    IloSingleMove(env, _solution, _nhood, _greedy, _instantiateCost);

  _solver.out() << endl << "Improvement loop" << endl;
  _solver.out() << "================" << endl;
  IloBool improved = IloTrue;
  IloInt nbOfDays= _routing.getNumberOfDays();
  improved = IloFalse;
  for (IloInt d = 0; d < nbOfDays; ++d) {
    Day* day = _routing.getDay(d);
    day->fillModel(_solution);
    syncSolution(day->getModel(), _solution, _instantiateCost);
    if ( day->improve(_solver, _solution, _instantiateCost) ) {
      improved = IloTrue;
    }
  }
  // sync back to full model.

  syncSolution(_mdl, _solution, _instantiateCost);
  return improved;
}
```

Define the RoutingSolver::improvePlan function

After the routing for all days has been improved independently, the routing for the problem as a whole is improved. After resetting the neighborhood and greedy metaheuristic, an improvement loop is entered to improve the cost of all days using the goal `_move`. If there was some improvement, then you go back to an independent improvement of each day. Otherwise, you leave the main improvement loop.

```
IloBool RoutingSolver::improvePlan() {
  _solver.out() << "Optimizing plan "
                << _solution.getSolution().getObjectiveValue()
                << flush;
  _nhood.reset();
  _greedy.reset();
  IloInt nbImproved = 0;
  while ( _solver.solve(_move) ) { ++nbImproved;}
  _solver.out() << " ---> "
               << _solution.getSolution().getObjectiveValue() << endl;
  return ( nbImproved > 0 );
}


void RoutingSolver::syncSolution(IloModel mdl, IloSolution s, IloGoal g) {
  _solver.extract(mdl);
  if ( !_solver.solve( IloRestoreSolution( _solver.getEnv(), s) && g))
    _solver.out() << "Synchronization failed" << endl;
  else {
    s.store(_solver);
  }
}

void RoutingSolver::syncSolution() {
  syncSolution(_mdl, _solution, _instantiateCost);
}
```

Other modeling hints

This section lists some other modeling hints that may be useful to you as you design your Dispatcher application.

In problems where the same truck makes many returns to the depot, you may want to use shorter routes. For example, if you have the same truck make 10 visits to the depot, you may end up with a route with 50-100 visits. Since local search complexity is $O(n^2)$ of the length of the route, you may want to split one physical truck into several symbolic trucks. You can constrain the last visit of truck 1 to be before first visit of truck 2, and so on. This will improve local search performance.

In problems, such as taxi dispatching and concrete delivery, where vehicles go from one location to another and are either empty or full, you may want to model these as arc routing problems. However, if both visits have time windows or other constraints, then they should be modeled as two nodes. The first solution and local search will see them as a single visit since the next variable is bound.

C H A P T E R

17

Developing Your Own First Solution Heuristics

In this lesson, you will learn how to:

- decide which predefined first solution heuristics to use
- write your own first solution heuristic
- use the Dispatcher first solution framework

Dispatcher offers a variety of predefined heuristics for use in generating a first solution. To better understand which predefined first solution heuristics to use with your problem, see “Using the predefined first solution heuristics” on page 407. The process of writing a simple first solution heuristic gives you a better understanding of the concepts. See “Creating your own first solution heuristic” on page 409. Dispatcher also provides an open framework to define custom first solution heuristics. See “Using the Dispatcher First Solution Framework” on page 413

Using the predefined first solution heuristics

This section offers some insights into the relative complexity of the various predefined first solution heuristics. These hints are meant to help you decide which heuristics to use with your problem. Of course, every problem is different and no set of guidelines can account for all this diversity. A best approach is to start by following these guidelines, but to be flexible in trying different first solution heuristics. It can also be interesting to try several first solution methods and to apply local search to these different methods. You may also want to apply local search to an existing first solution.

If you are not able to create a first solution using the predefined heuristics, you may want to build your own. In other situations, you may want to build your own first solution heuristic because your problem has a specific structure that is not reflected in the predefined heuristics. For example, you may want to load your own trucks first and then load trucks belonging to contractors. For these situations, you can build your own first solution directly or using the Dispatcher first solution framework.

Deciding which heuristic to use

The Dispatcher predefined first solution heuristics can be placed in a rough hierarchy based on performance (with the fastest heuristic listed first):

Sweep > nearest-to-depot > nearest addition > savings > insertion > generate

However, certain problems respond more effectively to certain heuristics. Here are some general guidelines:

- If your vehicles have fixed starting and ending points, use a savings heuristic. Otherwise, use nearest addition.
- If your vehicles have fixed starting and ending points, the problem is not too constrained, and you have coordinates, use sweep.
- When the problem is really hard to solve and not too large, use insertion directly or insert visits one-by-one using `IloInsertVisit`. This approach can also be interesting if you want to update an already existing solution or if you already know the order of some visits.
- Only use generate on small problems.

More hints

The `size` parameter in the nearest addition and savings heuristics limits the number of possible next visits to the *n*-closest one, according to cost. This parameter can lower memory consumption and also speed up search. The parameter `size` should be fixed and not proportional to the size of the problem. A good number to try is something between 50 and 100 visits. As always, testing to find the best fit for your problem and environment works best.

The parameter `mode` indicates the behavioral mode of the nearest addition heuristic during execution. The mode `IloNearestAdditionForward` extends the route forward from the first visit. The mode `IloNearestAdditionBackward` extends the route backward from the last visit. The mode `IloNearestAdditionBoth` extends the route simultaneously in both directions. In general, it is best to use the backwards mode with outbound problems (few pickups, many delivery locations). The forwards mode works best with inbound problems (few deliveries, many pickup locations).

Creating your own first solution heuristic

Although Dispatcher offers a variety of predefined heuristics for use in generating a first solution, the process of writing a simple one gives you a better understanding of the concepts. Knowing how to write your own first solution heuristic can also be useful when facing certain specific problems that the predefined heuristics do not cover. This section describes how to write your own efficient search goal based on a greedy addition heuristic.

To show how to create your own first solution heuristic, you will solve a PDP problem. Except for the custom first solution heuristic, you will model and solve this problem in the same way as shown in Lesson 7, *Pickup and Delivery Problems*. The only difference is that you create your own first solution goal and that you call this goal in the `RoutingSolver::solve` function instead of a predefined first solution goal. Only the changed parts of the example are shown in this section. You can view the complete program and output online in the `YourDispatcherHome/examples/src/firstsol.cpp` file.

Heuristic Description

For the sake of simplicity, the heuristic chosen supposes that the cost of the vehicles only depends on one dimension `dim`. It sequentially builds the routes for the vehicles which have the largest capacity, extending these routes with the cheapest visits after each step of visit addition.

Here is how the algorithm works:

1. Select an open vehicle `w` with the largest capacity. If there is none, the algorithm ends.
2. Start a partial route consisting of the first visit of `w`. Let `vl` be the last visit of this partial route.
3. Find a visit `v` which minimizes the cost of going from `vl` to `v` using vehicle `w`. If it is not possible to find such a visit without violating constraints, close the current partial route of `w` and go to step 1.
4. Add `v` to the end of the partial route and let `vl = v`. Go to step 3.

Building Routes

The goal that builds the route for a given vehicle is the central goal of the search. Given the addition heuristic approach described in the preceding section, the goal iteratively chooses visits and tests to see if the selected visit can extend the current partial route.

Visits are chosen in order to minimize the cost increase of the current partial route. This is done by iterating on the possible successors of the last visit of the current partial route. The heuristic selects the cheapest visit that is not a first or a last visit and the vehicle variable that contains the current vehicle.

```
IloVisit ChooseCheapestVisit(IloDispatcher dispatcher,
                             IloVehicle vehicle,
                             IloVisit visit) {
  IloInt vehIndex = dispatcher.getIndex(vehicle);
  IlcIntVar nextVar = dispatcher.getNextVar(visit);
  IloVisit bestVisit;
  IloNum bestCost = IloInfinity;
  for (IlcIntExpIterator iter(nextVar); iter.ok(); ++iter) {
    IloVisit v = dispatcher.getVisit(*iter);
    IloNum cost = dispatcher.getCost(visit, v, vehicle);
    IlcIntVar vehicleVar = dispatcher.getVehicleVar(v);
    if (cost < bestCost
        && !v.isFirstVisit()
        && !v.isLastVisit()
        && vehicleVar.isInDomain(vehIndex)) {
      bestVisit = v;
      bestCost = cost;
    }
  }
  return bestVisit;
}
```

An internal search tests if the visit can extend the current partial route by checking if the route can be closed with the addition of the newly selected visit. The goal used for this search constrains the selected visit to directly follow the last visit of the current route, using `IloDispatcher::setNext(IloVisit, IloVisit)`, and closes the route using `IloGenerateRoute(IloSolver, IloVehicle)`. Note that it is better to limit the search when closing the route to avoid spending too much time proving that the route cannot be closed.

```
ILCGOAL3(AddArcTestGoal,
         IloVisit&, visit1,
         IloVisit&, visit2,
         IloVehicle&, vehicle) {
  IloSolver solver =  getSolver();
  IloDispatcher dispatcher(solver);
  dispatcher.setNext(visit1, visit2);
  return IloLimitSearch(solver,
                        IloGenerateRoute(solver, vehicle),
                        IloFailLimit(solver, 100));
}
```

If the selected visit can extend the route, it is constrained to do so, using `IloDispatcher::setNext(IloVisit, IloVisit)`, and becomes the new last visit of the current partial route. The iteration ends when no more visits can be found to extend the partial route.

Here is the code for generating a route:

```
ILCGOAL1(Ilc_VehicleGenerateSolution, IloVehicle, vehicle) {
  IloSolver solver = getSolver();
  IloDispatcher dispatcher(solver);
  IloVisit visit1 = vehicle.getFirstVisit();
  IloVisit visit2 = ChooseCheapestVisit(dispatcher, vehicle, visit1);
  IlcGoal testGoal = AddArcTestGoal(solver, visit1, visit2, vehicle);
  while (visit2.getImpl() != 0) {
    if (solver.solve(testGoal, IloTrue)) {
      dispatcher.setNext(visit1, visit2);
      visit1 = visit2;
    } else {
      solver.add(dispatcher.getNextVar(visit1) != dispatcher.getIndex(visit2));
    }
    visit2 = ChooseCheapestVisit(dispatcher, vehicle, visit1);
  }
  solver.solve(IloGenerateRoute(solver, vehicle));
  return 0;
}
```

Building a Complete Solution

The last goal to be written recursively chooses a vehicle and builds its route. It considers the visits which have not yet been assigned to any vehicle by the end of the search and marks those visits as “unperformed.” Choosing a vehicle is done by iterating over all vehicles and finding one which has an incomplete route and which has the largest capacity for dimension `dim`.

```
IloVehicle ChooseLargestVehicle(IloDispatcher dispatcher, IloDimension1 dim) {
  IloEnv env = dispatcher.getEnv();
  IloVehicle bestVehicle;
  IloNum bestCapacity = - IloInfinity;
  for (IloIterator<IloVehicle> vehIter(env); vehIter.ok(); ++vehIter) {
    IloVehicle vehicle = *vehIter;
    IloNum capa = vehicle.getCapacity(dim);
    if (capa > bestCapacity && !dispatcher.isRouteComplete(vehicle)) {
      bestVehicle = vehicle;
      bestCapacity = capa;
    }
  }
  return bestVehicle;
}
```

Building a route for the selected vehicle is done using the goals described in the preceding sections. The search ends by making unassigned visits unperformed when all open vehicles have been used.

```
ILCGOAL0(UnperformVisits) {
  IloSolver solver = getSolver();
  IloDispatcher dispatcher(solver);
  IloEnv env = dispatcher.getEnv();
  for (IloIterator<IloVisit> visIter(env); visIter.ok(); ++visIter) {
    IloVisit visit = *visIter;
    if (!dispatcher.getVehicleVar(visit).isBound())
      solver.add(dispatcher.unperformed(visit));
  }
  return 0;
}
```

In order to have more propagation from the path constraints during search the filter level is set to `IlcBasic`.

```
ILCGOAL1(Ilc_GenerateFirstSolution, IloDimension1, dim) {
  IloSolver solver = getSolver();
  IloDispatcher dispatcher(solver);
  dispatcher.setFilterLevel(IlcBasic);
  IloVehicle vehicle = ChooseLargestVehicle(dispatcher, dim);
  if (vehicle.getImpl() == 0) return UnperformVisits(solver);
  return IlcAnd(Ilc_VehicleGenerateSolution(solver, vehicle), this);
}
```

To use this goal, a subclass of `IlcGoal`, in your Concert Technology modeling environment you wrap it using the Solver `ILOCPGOALWRAPPER` macro.

```
ILOCPGOALWRAPPER1(GenerateFirstSolution, solver, IloDimension1, dim) {
  return Ilc_GenerateFirstSolution(solver, dim);
}
```

Note: *This approach can be adapted to the user's needs by customizing the functions selecting the vehicles and the visits.*

Using the custom first solution goal

The custom first solution goal `GenerateFirstSolution` is called in the `RoutingSolver::solve` function.

```
void RoutingSolver::solve() {
  IloDispatcher dispatcher(_solver);
  IloEnv env = _solver.getEnv();
  IloDimension1 weight = IloDimension1::Find(env, "Weight");
  // Subgoal
  IloGoal instantiateCost = IloDichotomize(env,
             dispatcher.getCostVar(),
             IloFalse);
  IloGoal restoreSolution = IloRestoreSolution(env, _solution);
  IloGoal goal = GenerateFirstSolution(env, weight) && instantiateCost;

  // Solving
  if (findFirstSolution(goal)) {
    improve(instantiateCost);
    _solver.solve(restoreSolution);
  }
}
```

Using the Dispatcher First Solution Framework

Dispatcher provides an open framework to define custom first solution heuristics. A first solution heuristic builds a routing plan by assigning visits (or groups of visits) to vehicles. It does so by considering elementary decisions, and either rejecting them as infeasible, or committing them into the solution that is being built.

To show how to use the Dispatcher First Solution Framework to create your own first solution heuristic, you will solve a PDP problem. Except for creating and using the first solution framework goal, you will model and solve this problem in the same way as shown in Lesson 7, *Pickup and Delivery Problems*. Only the changed parts of the example are shown in this section. You can view the complete program online in the `YourDispatcherHome/examples/src/fsframepdp.cpp` file.

Note: *This example is designed to show how to use the Dispatcher First Solution Framework. It is not intended to scale for use with problems that have large numbers of vehicles or orders.*

You will perform the following steps to create a first solution framework goal:

◆ Create a decision class `PDPDecision` with four member functions: `make`, `evaluate`, `calcFeasibility`, and `display`.

- Create a decision maker class `PDPDecisionMakerI` with three member functions: `init`, `getBestDecision`, and `getBestSerialDecision`. This class will use the decision class `PDPDecision`.
- Create a decision tracer class `PDPDecisionTracerI`.
- Create the first solution goal `IloPDPFSGoal`.

Creating the decision class

First, you create your decision class `PDPDecision`. This is a subclass of `IloPairDecisionI`, an abstract class that is a subclass of `IloSingleVehicleFSDecisionI`, involving one vehicle, and a pair of pickup and delivery visits. This class is dedicated to PDP heuristics.

The abstract class `IloSingleVehicleFSDecisionI` handles decisions concerning one vehicle. To simplify the decision comparison process, a cost is associated to decisions, computed by a new virtual member function called `evaluate`. Subclassing from this class requires you to define an `evaluate` member function. In addition, you will also define the following member functions: `make`, `calcFeasibility`, and `display`.

```
class PDPDecision : public IloPairDecisionI {

public:
  PDPDecision(IloVehicle vehicle,
              IloVisit pickup, IloVisit delivery);

  virtual void make(IloFSDecisionMakerI * dm);
  virtual IloNum evaluate(IloFSDecisionMakerI* dm) const;
  virtual IloBool calcFeasibility(IloFSDecisionMakerI* dm) const ;
  virtual void display(ostream& os) const;
};
```

The constructor for `PDPDecision` takes three parameters: a vehicle, a pickup visit, and a delivery visit.

```
PDPDecision
::PDPDecision(IloVehicle vehicle, IloVisit pickup, IloVisit delivery)
  : IloPairDecisionI(vehicle, pickup, delivery) {}
```

The function `make` performs the decision. It is called after a decision has been selected and checked for feasibility.

```
void PDPDecision::make(IloFSDecisionMakerI* dm) {
  IloDispatcher dispatcher = dm->getDispatcher();
  IloVisit up = getOutChainEnd(dispatcher);
  IloVisit down = getInChainStart(dispatcher);

  dispatcher.setNext( up, getPickup());
  dispatcher.setNext( getDelivery(), down);
}
```

The `evaluate` member function computes a cost that is used to select a best decision to execute. Decisions with the lowest cost are preferred.

```
IloNum PDPDecision::evaluate(IloFSDecisionMakerI* dm) const {
  IloDispatcher dispatcher = dm->getDispatcher();
  IloVisit up = getOutChainEnd(dispatcher);
  IloVisit down = getInChainStart(dispatcher);
  IloVehicle vehicle(getVehicle());

  IloNum cost = 0;

  cost += dispatcher.getCost(up, getPickup(), vehicle);
  cost += dispatcher.getCost(getDelivery(), down, vehicle);
  return cost;
}
```

The `calcFeasibility` member function is used to filter infeasible decisions before calling `solver.solve` to save CPU time.

```
IloBool PDPDecision::calcFeasibility(IloFSDecisionMakerI * dm) const {
  IloBool ok = IloPairDecisionI::calcFeasibility(dm);
  IloDispatcher dispatcher = dm->getDispatcher();
  IloVisit up = getOutChainEnd(dispatcher);
  IloVisit down = getInChainStart(dispatcher);
  if ( ok ) {
    ok = IsArcLinkable(dispatcher, up, getPickup())
      && IsArcLinkable(dispatcher, getPickup(), getDelivery())
      && IsArcLinkable(dispatcher, getDelivery(), down)
      ;
  }
  if ( ok ) {
    IloDimension2 time = IloDimension2::Find(dispatcher.getEnv(), "Time");
    ok = canInsertPairOnDimension
      (dispatcher, time, getVehicle(), up, down, getPickup(), getDelivery());
  }
  return ok;
}
```

The `display` member function displays the decision object.

```
void PDPDecision::display(ostream& os) const {
  os << "[<-" << getPickup().getName()
     << ".(" << getVehicle().getName() << ")."
     << getDelivery().getName() << "->]"
    ;
}
```

Creating the decision maker class

Now, you create `IloPDPDecisionMakerI`, a specialized decision maker class for PDP problems. It will use the decision class `PDPDecision` that you just created. You will define three member functions: `init`, `getBestDecision`, and `getBestSerialDecision`.

```
class PDPDecisionMakerI : public IloFSDecisionMakerI {
private:
  IloVehicle _current;

  IloFSDecisionI* getBestSerialDecision();
public:
  PDPDecisionMakerI(IloDispatcher dispatcher);

  virtual void init();
  virtual IloFSDecisionI* getBestDecision();
};
```

The member function `init` iterates on all visit pairs that are extracted in the dispatcher and, for each visit, looks for all vehicles that can be assigned to these visits. For each compatible set (vehicle, pickup, delivery) it then does two things. First, it decides whether or not it is worth creating a `PDPDecision` decision for the set. If the set (vehicle, pickup, delivery) is not “interesting” then no decision is created and this possibility will not be considered by the first solution framework. If the set is “interesting,” then this decision is stored.

```
PDPDecisionMakerI
::PDPDecisionMakerI(IloDispatcher dispatcher)
  : IloFSDecisionMakerI(dispatcher),
    _current(0) {}

void PDPDecisionMakerI::init() {
  _current = 0;
//  cout << "starting PDP decision maker init()"<< endl;
  IloDispatcher dispatcher(getDispatcher());
  IloEnv env = dispatcher.getEnv();

  // iterate on all posted ordered sequences.
  for (IloDispatcher::OrderedVisitPairIterator vpiter(dispatcher);
       vpiter.ok(); ++vpiter) {
    IloVisit pickup = vpiter.getPickup();
    IloVisit delivery = vpiter.getDelivery();
    IlcIntVar pickupVclVar = dispatcher.getVehicleVar(pickup);
    IlcIntVar deliveryVclVar = dispatcher.getVehicleVar(delivery);
    for (IloDispatcher::VehicleIterator
           vehIter(dispatcher); vehIter.ok(); ++vehIter) {
      IloVehicle vehicle = *vehIter;
      IlcInt vehIndex = dispatcher.getIndex(vehicle);
      if ( pickupVclVar.isInDomain(vehIndex) &&
           deliveryVclVar.isInDomain(vehIndex) ) {
        IloFSDecisionI* decision =
          new (env) PDPDecision(vehicle, pickup, delivery);
        storeDecision(decision);
      }
    }
  }
  //cout << "end PDP decision maker init()"<< endl;
}
```

The member function `getBestDecision` is the implementation of the virtual member function defined in the abstract `IloNADecisionMakerI` class. In this example, the extension mode is serial. Therefore, the decision maker fills vehicles one at a time, and this member function returns the best decision concerning the currently open vehicle. At the beginning, no vehicle is open, and the member function returns the best decision among all possible decisions. Afterwards, the decision's vehicle becomes the current vehicle, and only decisions registered with this vehicle are considered. When the vehicle is full, the decision maker looks again for the best global decision and chooses a new current decision, or terminates.

```
IloFSDecisionI* PDPDecisionMakerI::getBestDecision() {
  //return getBestGlobalDecision();
  return getBestSerialDecision();
}


IloFSDecisionI* PDPDecisionMakerI::getBestSerialDecision() {
  IloFSDecisionI * d = 0;
  if ( 0 != _current.getImpl() ) {
    d = getBestVehicleDecision(_current);
  }
  if ( 0 == d ) {
    IloFSDecisionI * bestVehicleDecision = getBestVehicleDecision();
    if ( 0 != bestVehicleDecision ) {
      IloSingleVehicleFSDecisionI * vehDecision =
        (IloSingleVehicleFSDecisionI*)bestVehicleDecision;
      _current = vehDecision->getVehicle();
      d = bestVehicleDecision;
    } else {
      _current = 0;
    }
  }
  return d;
}
```

Creating the decision tracer class

You can monitor events happening while building the first solution using decisions. A decision tracer can be attached to a decision maker object. The decision maker object will call the virtual member functions of the tracer objects to notify certain events. For example, each time a decision is chosen, the tracer object is notified. Dispatcher also provides a default tracer that does nothing. Subclassing from the default tracer `IloDefaultDecisionTracerI` only requires the definition of the member functions you want to trace, not all of them.

In this example, you use the default tracer to define only the `notifyChosen` trace member function.

```
class PDPDecisionTracerI : public IloDefaultDecisionTracerI {
  IloInt _nbChosen;
public:
  PDPDecisionTracerI(IloDispatcher dsp)
    : IloDefaultDecisionTracerI(dsp),
      _nbChosen(0) {}

  virtual void notifyChosen(const IloFSDecisionI * d) {
    ++_nbChosen;
    cout << _nbChosen << "> chosen: " << (*d) << endl;
  }
};
```

Creating the first solution framework goal

When building a Dispatcher routing plan, the resulting plan may not be complete. This means that some visits are left unassigned, and that some routes are not complete. By complete, it is meant that route visits must have their next variables instantiated from first to last. The routing plan may not be complete if, for example, only a subset of visits had been passed to the decision maker, or if a time-out occurred. An incomplete plan cannot be stored in an `IloRoutingSolution`, as some decision variables are left unbound.

The `IloFinalizePlan` function returns a goal that fixes an incomplete plan, if possible.

The goal performs the two functions:

- sets all visits that have no route assigned to unperformed
- closes all incomplete routes

You should run this goal after a decision maker attempts to finalize the plan and reach a state where the plan can be saved into an `IloRoutingSolution`.

```
ILCGOAL0(PDPFinalizeGoal) {
  cout << "start to finalize plan" << endl;
  return IloFinalizePlan(getSolver());
}
```

Finally, you write the goal `IlcPDPFSGoal`, which creates the decision maker and the decision tracer.

```
ILCGOAL0(IlcPDPFSGoal) {
  IloSolver solver(getSolver());
  IloDispatcher dispatcher(getSolver());

  PDPDecisionMakerI* dm =
    new (solver.getHeap()) PDPDecisionMakerI(dispatcher);

  // put a tracer  for debug
  PDPDecisionTracerI* tracer =
    new (solver.getHeap()) PDPDecisionTracerI(dispatcher);
  dm->setTracer(tracer);

  return IlcAnd( dm, PDPFinalizeGoal(solver) );
}
```

To use this goal, a subclass of `IlcGoal`, in your Concert Technology modeling environment you wrap it using the Solver `ILOCPGOALWRAPPER` macro.

```
ILOCPGOALWRAPPER0(IloPDPFSGoal, solver) {
  return IlcPDPFSGoal(solver);
}
```

Using the first solution framework goal

The first solution framework goal `IloPDPFSGoal` is called in the `RoutingSolver::solve` function.

```
void RoutingSolver::solve() {
  IloDispatcher dispatcher(_solver);
  IloEnv env = _solver.getEnv();
  // Subgoal
  IloGoal instantiateCost = IloDichotomize(env,
                                           dispatcher.getCostVar(),
                                           IloFalse);
  IloGoal restoreSolution = IloRestoreSolution(env, _solution);
  IloGoal goal = IloSavingsGenerate(env);
  goal = IloPDPFSGoal(env);

  // Solving
  if (findFirstSolution(goal && instantiateCost )) {
    improve(instantiateCost);
    _solver.solve(restoreSolution);
  }
}
```

C H A P T E R

18

Developing Your Own Neighborhoods

In this lesson, you will learn how to:

- define your own neighborhood
- create a solution delta
- implement your neighborhood

The predefined neighborhoods that Dispatcher offers are sufficient for most first solution improvements. Occasionally, you may want to write your own neighborhood. This section provides an example of how to create your own neighborhood solely for the user who is comfortable with using the predefined neighborhoods and who encounters the need to create a neighborhood.

The predefined neighborhoods provided with Dispatcher may not perform well on certain routing problems. For example, in a routing problem where there are constraints stating that after a visit A, visits B, C, and D must follow directly, neighborhoods like `IloRelocate` will not be able to move A, B, C, or D on its own to a new vehicle; the constraints forbid it. For problems with such tight constraints, you can write your own neighborhood to perform the movements you require. In the above example, you could create a neighborhood which was capable of moving A, B, C, and D as a single block to another location.

The following example uses a type of "restricted exchange." This is a neighborhood exactly like `IloExchange`, except that only visits within a certain distance of each other are considered for exchange. The other pairs of visits are ignored. To write your own neighborhood in Dispatcher, you subclass the class `IloNHoodI`. This is the exact same way you would create a neighborhood in Solver. A neighborhood is created through redefining virtual methods such as `start`, `getSize`, and `define`. The only difference between the methods used in Solver and those used in Dispatcher is that in Dispatcher you deal with routing objects (visits) and their associated attributes (next visit, previous visit, and vehicle). However, the method is essentially the same.

The following example relies heavily on your knowledge of Solver's local search methods. So, if you have not already done so, you should read about local search in the *IBM ILOG Solver User's Manual* before creating your own neighborhood.

Neighborhood Description

It is often a good idea to break down a neighborhood into its simplest form. This technique is used here. This new neighborhood exchanges all pairs of visits within a certain distance of each other. You can then break down this neighborhood such that there is a single neighborhood per visit which exchanges just this visit with all others in its vicinity. Then, by using `IloConcatenate` on an array of such subneighborhoods (one for each visit), you can achieve the desired result. Unfortunately, with this idea, an unwanted symmetry is introduced; each pair of visits within the distance limit will be considered twice. For example, consider the swap of visit pair A/B. The neighborhood which swaps visit A with its close neighbors and the neighborhood which swaps visit B with its close neighbors will both consider the same swap. Therefore, you add an additional rule. Given a total order for the visits, a subneighborhood which exchanges visit A with the visits in its vicinity can only exchange visit A with visits ranked after visit A in the total order.

To simplify the design of the subneighborhood, you specify the “focus” visit and the set of visits that can be used in an exchange. The code relating to proximity and the symmetry rule exists outside of this neighborhood. It is invoked only to construct the set of visits which will be exchanged with the “focus” visit.

Defining the Neighborhood

You begin by defining a neighborhood which exchanges the visit `here` with each one of a set of predefined visits `there`. You also declare other fields which are used internally: `rsolution` holds the current solution and `size` holds the size of the neighborhood (the number of neighbors). The fields `prevHere` and `nextHere` are used to hold the previous and next visits of `here` in the current solution. The field `vehHere` holds the vehicle performing the visit `here` and the field `perfHere` returns whether the visit `here` is performed or not.

```
class LocalizedExchangeI : public IloNHoodI {
private:
  IloVisit           _here;
  IloVisitArray      _there;
  IloRoutingSolution _rsolution;
  IloInt             _size;

  IloVisit           _prevHere;
  IloVisit           _nextHere;
  IloVehicle         _vehHere;
  IloBool            _perfHere;
```

You then declare the methods for the new neighborhood class. You use a constructor which takes a visit to swap and an array of visits with which this "focus" visit can be swapped. Three virtual member functions of `IloNHoodI` are also redefined: `start`, `define`, and `getSize`. The latter is implemented inline.

```
public:
  LocalizedExchangeI(IloEnv env, IloVisit visit, IloVisitArray arr);
  void start(IloSolver, IloSolution);
  IloSolution define(IloSolver, IloInt);
  IloInt getSize(IloSolver) const { return _size; }
};
```

The constructor of the subneighborhood builds the base class and installs the passed parameters.

```
LocalizedExchangeI::LocalizedExchangeI(IloEnv env,
                                       IloVisit visit,
                                       IloVisitArray arr)
  : IloNHoodI(env), _here(visit), _there(arr) { }
```

The `start` method is called by local search goals such as `IloSingleMove` to indicate the current solution. You keep a reference to the current solution, which is cast into an `IloRoutingSolution` for later use. (Casting the solution to an `IloRoutingSolution` allows you to use API suitable for routing problems on the solution. Note that the solution itself is not altered in any way by this cast; it simply allows the use of a routing-based API on the solution.) The `start` method also performs operations that need only to be done once. For example, you retrieve the following from the solution: the preceding and following visits of `here`, the vehicle performing the visit `here`, and whether or not the visit is performed. (If `here` is not performed, the values of the other three pieces of information are later ignored.) You also calculate the size of the neighborhood, which is the size of the `there` set. However, in order to be robust, you must set the size of the neighborhood to 0 if `here` is not in the current solution (in which case it does not make sense to involve it in a move), or if the visit is not extracted (in which case attempting to move it will cause Solver to throw an exception later.)

```
void LocalizedExchangeI::start(IloSolver solver,
                               IloSolution solution) {
  _rsolution = IloRoutingSolution(solution);
  if (solver.isExtracted(_here) && solution.contains(_here)) {
    _size = _there.getSize();
    _prevHere = _rsolution.getPrev(_here);
    _nextHere = _rsolution.getNext(_here);
    _vehHere = _rsolution.getVehicle(_here);
    _perfHere = _rsolution.isPerformed(_here);
  }
  else
    _size = 0;
}
```

The `define` method is usually the main workhorse of any neighborhood definition. It takes an index and converts it to a solution delta. (The solution delta is a solution holding only the part of the current solution which needs to be changed, together with its new value.) This delta needs to be of type `IloSolution`, **not** of type `IloRoutingSolution`, although it can be cast to an `IloRoutingSolution` to perform routing operations on it. The first thing you do in `define` is to determine what the index signifies. (In other words, you determine to which swap the index corresponds.) This is easily done as `here` is swapped with `there[index]`. Then you perform various checks which allow you to ignore this neighbor in certain circumstances. Specifically, when the solution does not contain the visit `there[index]`, or when this visit is not extracted, you can ignore the neighbor. Likewise, when both visits are unperformed (a swap is useless), or when both visits are served by the same vehicle (the move you define is only inter-route, but could be generalized), you ignore the neighbor. The neighbor is ignored by returning an empty handle (here you use `return 0` to perform this action).

```
IloSolution LocalizedExchangeI::define(IloSolver solver, IloInt index) {
  IloVisit v = _there[index];
  if (!_rsolution.getSolution().contains(v)) return 0;
  if (!solver.isExtracted(v)) return 0;
  IloBool perfV = _rsolution.isPerformed(v);
  if (!_perfHere && !perfV) return 0;
  if (_perfHere && perfV && _vehHere == _rsolution.getVehicle(v)) return 0;
```

Creating the Solution Delta

You now create the structure representing the neighbor: the solution delta. This is done by creating an empty solution delta (an instance of `IloSolution`). You also create a routing solution `rdelta` from this delta, so that routing operations can be performed on `delta`. Note that the delta should not be *created* as an `IloRoutingSolution` as this also performs other operations, such as adding an objective to the routing solution, which is not appropriate here.

```
IloSolution delta(getEnv());
IloRoutingSolution rdelta(delta);
rdelta.add(_here);
rdelta.add(v);
```

You now perform the “swap.” This is done in two stages. In the first stage, v is moved into the place of here, and in the second, here is moved into the former place of v. When you “insert” v into the place where here formerly was, you involve three visits: the visit preceding here in the current solution, the visit following here in the current solution, and v itself. The visit preceding here in the current solution will keep all its attributes in the delta, except that its following visit will be set to v, rather than here. A similar situation holds for the visit following here in the current solution. Most of the state of these visits is preserved in the delta. For this reason, their state is copied from the current solution. Then, the vehicle, previous visit, and next visit of v are set. The action of setting the next and previous visits of v also correctly sets the next and previous visits of the connected visits, so that consistency of the solution is maintained, and no explicit action is required for those visits. In the case where here is not performed, the “insertion” simply makes v unperformed.

```
if (_perfHere) {
  rdelta.setVehicle(v, _vehHere);
  rdelta.add(_nextHere); delta.copy(_nextHere, _rsolution);
  rdelta.add(_prevHere); delta.copy(_prevHere, _rsolution);
  rdelta.setNext(v, _nextHere);
  rdelta.setPrev(v, _prevHere);
}
else
  rdelta.setUnperformed(v);
```

The symmetric “insertion” is almost identical. The only difference is that the current preceding and following visits, as well as the vehicle serving v, need to be retrieved from the current solution first.

```
if (perfV) {
  rdelta.setVehicle(_here, _rsolution.getVehicle(v));
  IloVisit nextV = _rsolution.getNext(v);
  IloVisit prevV = _rsolution.getPrev(v);
  rdelta.add(nextV); delta.copy(nextV, _rsolution);
  rdelta.add(prevV); delta.copy(prevV, _rsolution);
  rdelta.setNext(_here, nextV);
  rdelta.setPrev(_here, prevV);
}
else
  rdelta.setUnperformed(_here);
```

Finally, the delta is returned, which concludes the definition of the `LocalizedExchangeI` class.

```
  return delta;
}
```

Implementing the Neighborhood

Recall that the above function only swaps a single visit with a predefined set of visits. Our initial goal was to create a neighborhood which would swap all visits within a certain range of each other. This more complete neighborhood is constructed by the following function. The essential technique is quite simple. You construct an array of neighborhoods, one corresponding to each visit (an instance of `LocalizedExchange`). The set of visits for each `LocalizedExchange` is calculated according to a proximity rule and to a symmetry rule. The array of neighborhoods is then concatenated to produce one larger neighborhood. The function is passed with the following parameters:

- an array of visits to consider for swaps (This should correspond to all real visits of the problem. In other words, it should not include visits which are first or last visits of a vehicle.)
- an instance of `IloDimension2` used for measuring distances
- a vehicle which is used as a “measuring vehicle” (A distance in Dispatcher depends on a dimension **and** a vehicle. Therefore, a vehicle is needed to compute the distances.)
- a limit on the round trip between the two visits (If the actual distance of the round trip is less than this value, the two visits will be considered for swapping. A round trip is used to avoid ambiguity problems which may occur with asymmetric distance functions.)

Given this information, the implementation of the function follows:

```
IloNHood LocalizedExchange(IloEnv env,
                           IloVisitArray visits,
                           IloDimension2 dim,
                           IloVehicle modelVehicle,
                           IloNum roundTrip) {
  IloNHoodArray nhoods(env, visits.getSize() - 1);
  for (IloInt i = 0; i < visits.getSize() - 1; i++) {
    IloVisit v = visits[i];
    IloVisitArray arr(env);
    for (IloInt j = i + 1; j < visits.getSize(); j++) {
      IloVisit w = visits[j];
      IloNum dist2 = v.getDistanceTo(w, dim, modelVehicle) +
                     w.getDistanceTo(v, dim, modelVehicle);
      if (dist2 <= roundTrip) arr.add(w);
    }
    nhoods[i] = new (env) LocalizedExchangeI(env, v, arr);
  }
  return IloConcatenate(env, nhoods);
}
```

You can view the entire program and output online in the file `YourDispatcherHome/examples/src/newnhood.cpp`.

A

Predefined First Solution Heuristics

Dispatcher offers a variety of predefined first solution heuristics for use in generating a first solution. Each predefined first solution heuristic is explained in this appendix. To show how the different predefined first solution heuristics work, their results on the same example problem are shown in each section of this appendix. The following predefined first solution heuristics are provided by Dispatcher: enumeration, savings, sweep, nearest-to-depot, nearest addition, and insertion.

Example data

The following example data is used in this appendix to demonstrate how each predefined first solution heuristic works. The goal of the example is to compute a route that starts and ends at the depot and visits each node exactly once. The length of the route is computed according to Euclidean distance.

In the following figure, the depot is represented as a black circle. The other circles represent visits to make at various locations. The coordinates of the depot are (30,40). The capacity of the truck is 50 and there are 11 visits to perform. The tuples (x,y,quantity) of the visits are (17,63,19), (31,62,23), (52,64,16), (21,47,15), (37,52,7), (49,49,30), (42,41,19), (20,26,9), (40,30,21), (52,33,11) and (51,21,5).

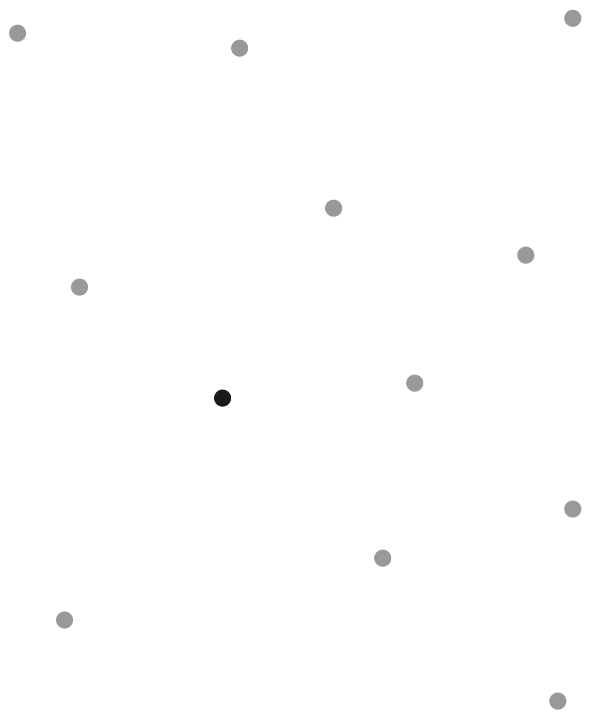

Figure A.1 *Example data: depot and visits*

Enumeration Heuristic

Dispatcher provides a basic, complete enumeration method for generating a first solution. The enumeration heuristic builds a solution to the problem using an algorithm that completely explores the search space using backtracking. This method should only be used for small problems.

This heuristic is implemented in Dispatcher as the goal `IloDispatcherGenerate`.

The following figure provides an example of the use of `IloDispatcherGenerate` on the example VRP with capacity constraints. The cost of this solution is 352.57.

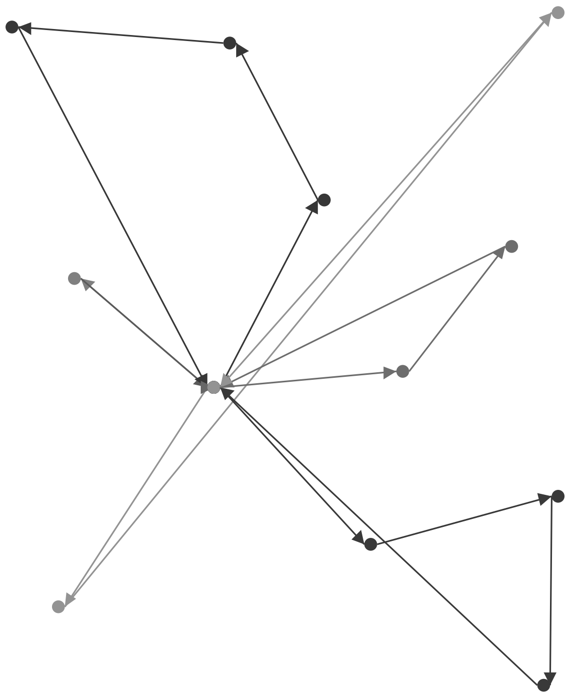

Figure A.2 *Sample solution using enumeration heuristic*

Savings Heuristic

For problems with multiple vehicles, it is very important to consider the trade-off between more vehicles with shorter routes and fewer vehicles with longer routes. For example, the following figure shows two ways of making visits *a* and *b.*

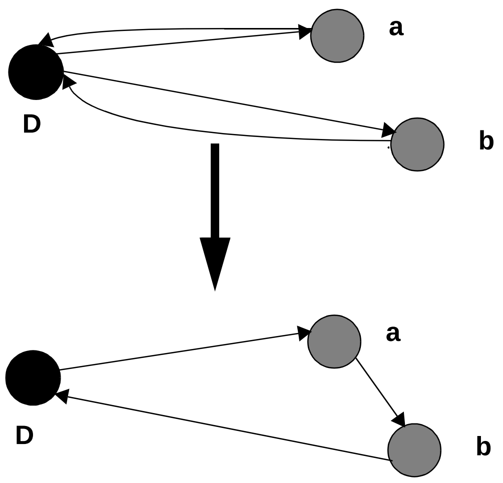

Figure A.3 *Principle of the savings heuristic*

The *savings* of serving a and b in the same route, denoted *savings (a, b)*, is defined as *savings (a, b) = cost (a, Depot) + cost (Depot, b) – cost (a, b)*. The savings heuristic generates a solution based on this equation. It works like this:

1. Choose an arbitrary visit *O* (usually the depot), and for all pairs of visits *(i, j)*, compute the savings function: *savings(i, j) = cost (i, O) + cost (O, j) – cost(i, j)*.
2. Sort the arcs *(i, j)* according to *savings(i, j)* in descending order, and put them in a list *L*.
3. If all visits are scheduled, the goal succeeds.
4. If there are unscheduled visits, choose an untried vehicle *v* from the plan. If there are no untried vehicles, the unscheduled visits are constrained to be unperformed. If one of these visits must be performed, the goal fails.
5. Scan *L* to find an arc that can be used to create an initial partial route for *v*. If no such legal arc can be found, go to step 4, otherwise remove the chosen arc from *L*.
6. Scan *L* to find an arc that can be added to the start or end of the route. If no such arc can be found, go to step 3, otherwise remove the arc from *L*, and repeat step 6.

This heuristic is implemented in Dispatcher as the goal `IloSavingsGenerate`.

The following figure provides an example of the use of `IloSavingsGenerate` on the example VRP with capacity constraints. The cost of this solution is 280.947.

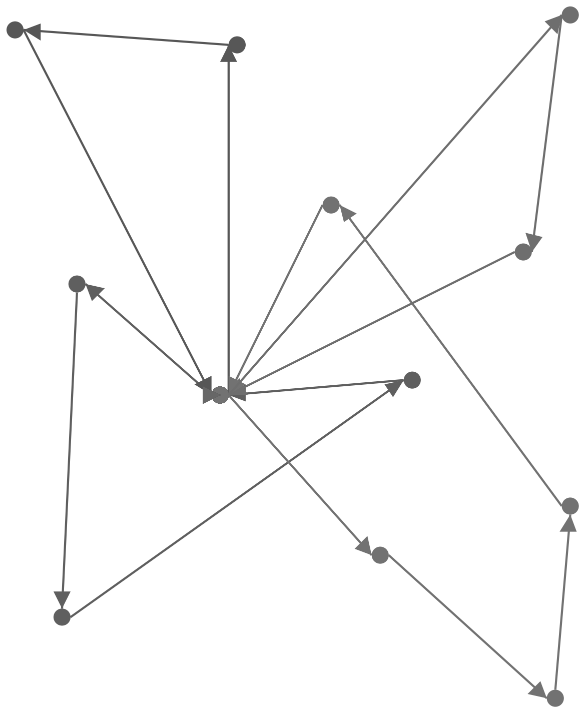

Figure A.4 *Sample solution using savings heuristic*

Sweep Heuristic

The sweep heuristic builds routes by sweeping around the depot:

1. Let *O* be a site from which vehicles leave (usually a depot), and let *A* (different from *O*) be another site which serves as a reference.
2. Sort all the sites *S* in the routing plan by increasing angle *AOS*. Put the result in a list *L*.
3. The visits corresponding to the sites in *L* will be allocated to the vehicles in that order as long as constraints are respected.
4. If all vehicles have been used, the remaining visits are constrained to be unperformed. If one of these visits must be performed, the goal fails.

This heuristic is implemented in Dispatcher as the goal `IloSweepGenerate`.

The following figure provides an example of the use of `IloSweepGenerate` on the example VRP with capacity constraints. The cost of this solution is 270.403.

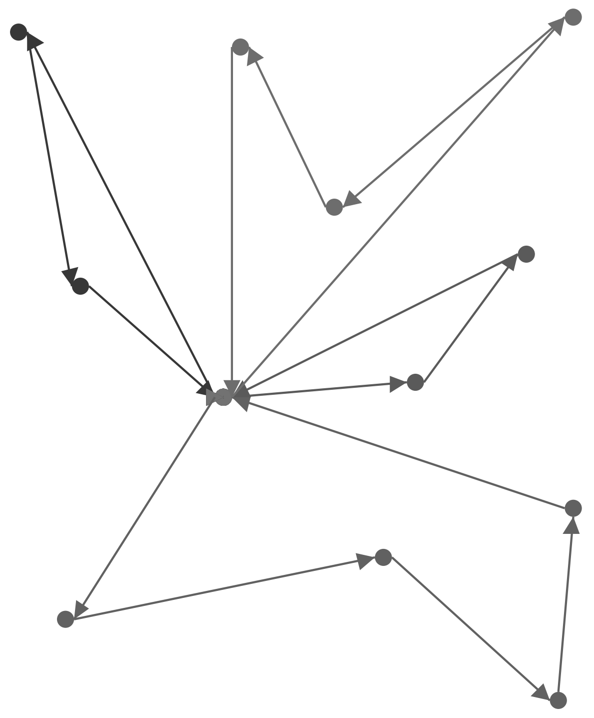

Figure A.5 *Sample solution using sweep heuristic*

Nearest-to-Depot Heuristic

The nearest-to-depot heuristic builds routes by adding the visits close to the depot first.

For all vehicles:

1. Denote the vehicle to be considered by *w*.
2. Start with a partial route consisting of the departure from the depot.
3. Find the visit *v* which is closest to the starting point of the current partial route of *w*. If it is not possible to find such a visit without violating constraints, close the current partial route of *w*, choose another empty vehicle and go to step 2. If no empty vehicles remain, the goal fails.

4. Add *v* to the end of the partial route.
5. Go to step 3.
6. If all vehicles have been used, the remaining visits are constrained to be unperformed. If one of these visits must be performed, the goal fails.

This heuristic is implemented as the goal `IloNearestDepotGenerate`.

The following figure provides an example of the use of `IloNearestDepotGenerate` on the example VRP with capacity constraints. The cost of this solution is 369.297.

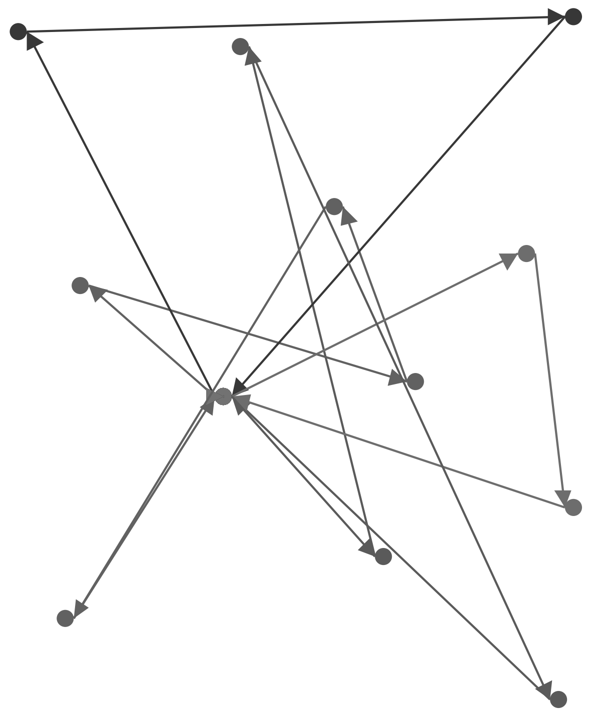

Figure A.6 *Sample solution using nearest-to-depot heuristic*

Nearest Addition Heuristic

The nearest addition heuristic is very similar to the nearest-to-depot heuristic just described. It often gives better results because the visit added to the route is the one closer to the end of the route, not the one closer to the depot.

For all vehicles:

1. Denote the vehicle to be considered by *w*.
2. Start with a partial route consisting of the departure from the depot.
3. Find the visit *v* which is closest (that is, least costly to get to) to the end of the current partial route of *w*. If it is not possible to find such a visit without violating constraints, close the current partial route *w*, choose another empty vehicle and go to step 2. If no empty vehicles exist, the goal fails.
4. Add *v* to the end of the partial route.
5. Go to step 3.
6. If all vehicles have been used, the remaining visits are constrained to be unperformed. If one of these visits must be performed, the goal fails.

This heuristic is implemented in Dispatcher as the goal `IloNearestAdditionGenerate`.

The following figure provides an example of the use of `IloNearestAdditionGenerate` on the example VRP with capacity constraints.The cost of this solution is 285.232.

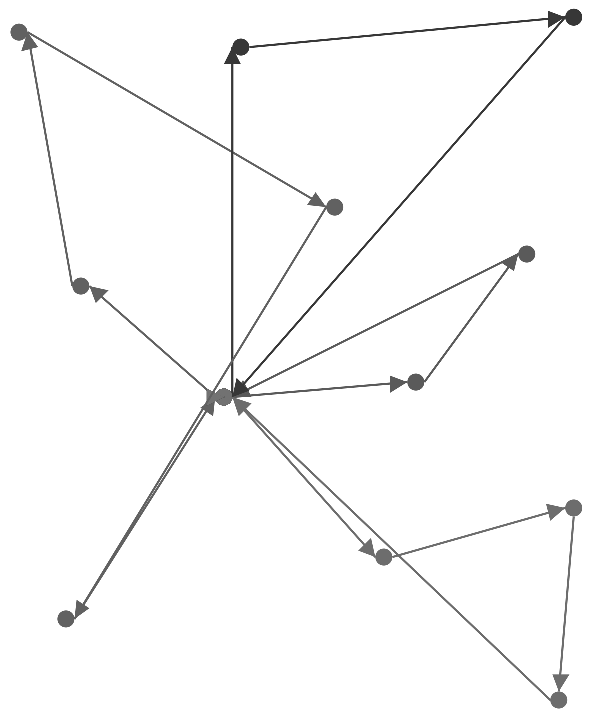

Figure A.7 Sample solution using nearest addition heuristic

Insertion Heuristic

The insertion heuristic works by inserting each visit (in the order they were created) at the best possible place, in terms of cost.

1. Let all vehicles have empty routes.
2. Let *L* be the list of unassigned visits.
3. Take a visit *v* in *L*.
4. Insert *v* in a route at a feasible position where there will be the least increase in cost. If there is no feasible position, then the goal fails.
5. Remove *v* from *L*.
6. If *L* is not empty, go to 3.

This heuristic is implemented in Dispatcher as the goal `IloInsertionGenerate`.

The following figure provides an example of the use of `IloInsertionGenerate` on the example VRP with capacity constraints.The cost of this solution is 300.681.

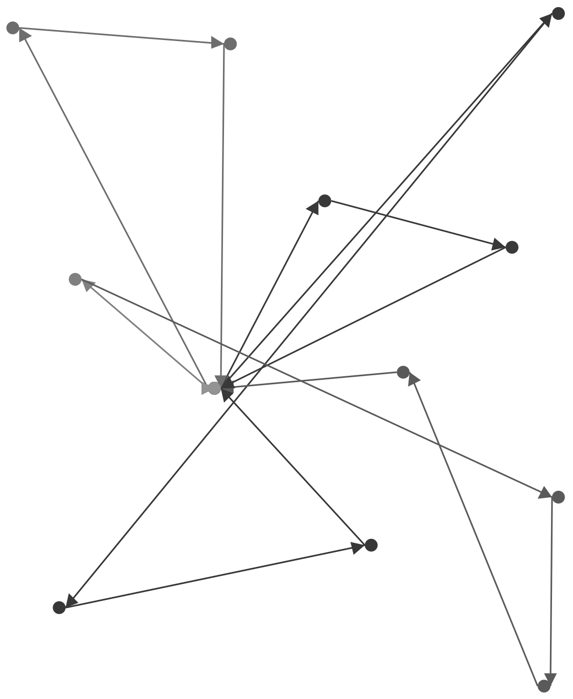

Figure A.8 *Sample solution using insertion heuristic*

APPENDIX

B

Predefined Neighborhoods

Dispatcher offers a variety of predefined neighborhoods for use in improving a solution. Each predefined neighborhood is explained in this appendix.

The heuristics that generate a first solution to a routing problem find a “good” feasible solution very quickly. These first solutions can be further improved by using neighborhoods to reduce the costs of the routes they find. The central idea of the neighborhood is to define a set of solution changes, or deltas, that represent alternative moves that can be taken. Neighborhoods are classified in three groups: those that modify only one route are known as intra-route neighborhoods; those that make changes between routes are known as inter-route neighborhoods; and those that change whether visits are performed or not. Interestingly enough, the inter-route neighborhoods can sometimes be used to improve a single route and thus become intra-route neighborhoods themselves.

The following predefined neighborhoods are provided by Dispatcher: Two-Opt, Or-Opt, Relocate, Cross, Exchange, and several neighborhoods that change whether visits are performed or not.

***Note:** If visit v2 is forced to follow immediately after visit v1 because of a constraint, the predefined neighborhoods will move the two visits v1 and v2 as a unit, or a chain.*

Intra-Route Improvement: IloTwoOpt neighborhood

In a Two-Opt neighborhood two arcs in a single route are cut and reconnected to improve the total cost of the route, as follows:

1. Take an initial route.
2. Remove two arcs from the route, and try the other possible reconnecting of the remaining parts of the route.
3. If the cost has been reduced and if all constraints are satisfied, go back to Step 2.
4. End.

With this neighborhood, directional flows between visits may be reversed. The presence of tight time constraints can therefore decrease its effectiveness. Two-Opt is implemented by the function `IloTwoOpt`.

The following figure illustrates this process. Here, we assume that the cost is proportional to the length of the route. The move eliminates the crossing by destroying two arcs and creating two new arcs. The resulting route is shorter, and thus less costly.

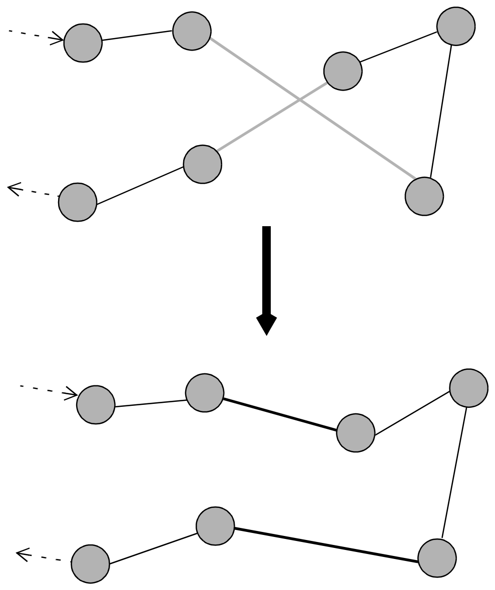

Figure B.1 *Using a Two-Opt neighborhood*

Intra-Route Improvement: IloOrOpt neighborhood

In an Or-Opt neighborhood segments of visits in the same route are relocated.

1. Start with an initial route.
2. Move parts composed of one visit elsewhere in the route.
3. If the cost has been reduced and if all constraints are satisfied, go back to Step 2.
4. When all such moves have been tested, try moving parts of the route composed of two consecutive visits.
5. After testing all moves of parts composed of two consecutive visits, try moving parts of the route composed of three consecutive visits.

Or-Opt is implemented by the function `IloOrOpt`.

The following figure illustrates the process. Here, the cost is assumed to be proportional to the length of the route. The move eliminates the crossing by destroying three arcs and creating three new arcs. The resulting route is shorter, and thus less costly.

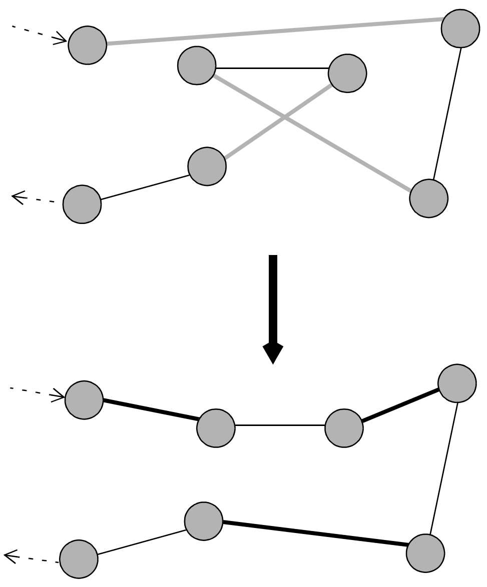

Figure B.2 *Using an Or-Opt neighborhood*

Inter-Route Improvement: IloRelocate neighborhood

In a relocate neighborhood a visit is inserted in another route if all constraints—such as capacity and time—are still satisfied. This method can be generalized if more than one visit of a route is moved at the same time. When pairs of visits are moved, the neighborhood is useful for optimizing problems such as the Pickup-and-Delivery Problem (PDP). Relocate is implemented by the function `IloRelocate`.

The following figure shows the process. Here, we assume that the cost is proportional to the length of the route. The move destroys three arcs and creates three new arcs. As a result total travel distance, and thus cost, is less.

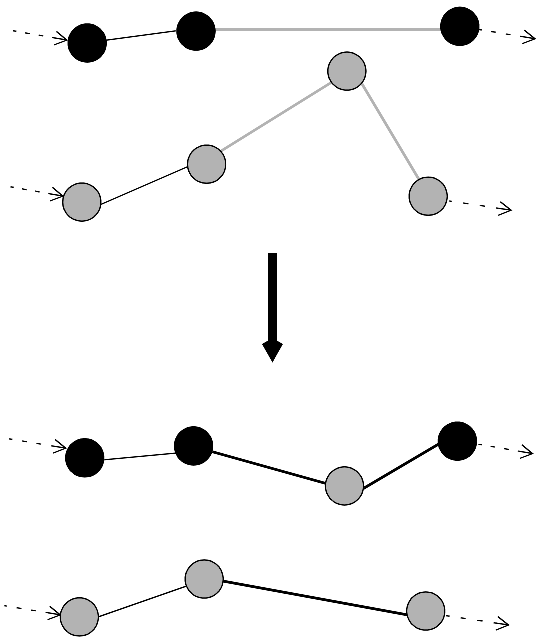

Figure B.3 *Using a Relocate neighborhood*

The function `IloFPRelocate` returns a neighborhood that modifies a solution by relocating individual visits to a new position in another route. This function is similar to `IloRelocate` for PDP problems, except that it explores more options for the delivery component of a pickup- delivery pair that is being relocated. For example, consider two pairs of pickup-delivery visits: *p1*-*d1* and *p2*-*d2*. `IloRelocate` would try to move *p2* after *p1* and *d2* immediately after *d1*. `IloFPRelocate` would try to move *p2* after *p1*, and then try to locate *d2* at every position after *d1*. Thus, this neighborhood is potentially larger than that created by `IloRelocate`.

Inter-Route Improvement: IloCross neighborhood

In a cross neighborhood the ends of two routes are exchanged: the first part of route A is connected to the last part (end) of route B and the first part of route B is connected to the last part (end) of route A. Cross is implemented by the function `IloCross`.

The following figure illustrates the process. The cost is assumed to be proportional to the length of the route. The move eliminates the crossing by destroying two arcs and creating two new arcs. The resulting routes are shorter, and thus less costly.

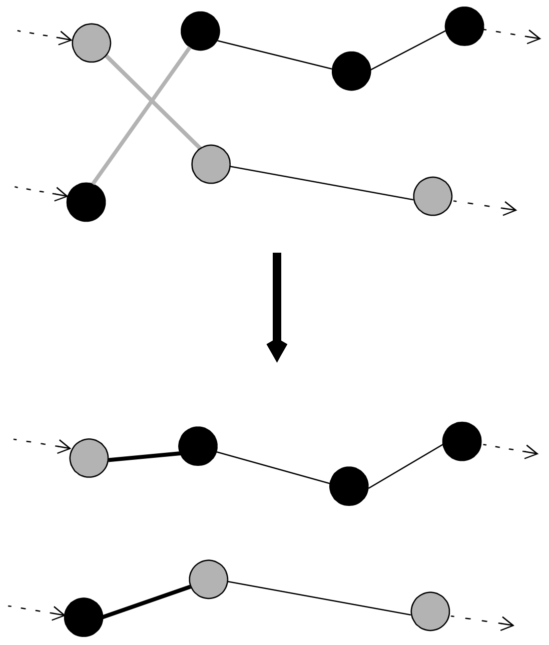

Figure B.4 *Using a Cross neighborhood*

Inter-Route Improvement: IloExchange neighborhood

In an exchange neighborhood, two visits of two different routes swap places if all constraints are still satisfied. This method can be generalized if more than one visit of a route is exchanged at the same time. When a pair of visits is exchanged, this neighborhood is useful for optimizing problems such as the Pickup-and-Delivery Problem (PDP). Exchange is implemented by the function `IloExchange`.

The following figure shows the process. The cost is assumed to be proportional to the length of the route. The move eliminates the crossings by destroying four arcs and creating four new arcs. The resulting routes are shorter, and thus less costly.

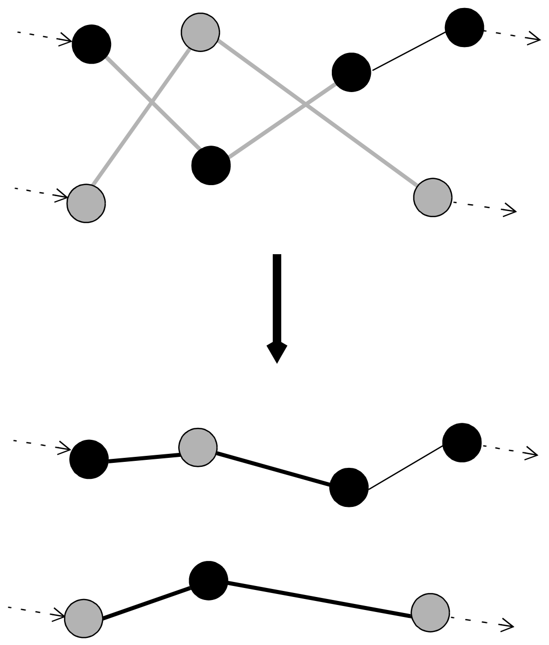

Figure B.5 *Using an Exchange neighborhood*

Other neighborhoods

Dispatcher provides predefined neighborhoods that change whether a visit is performed or not.

The function `IloMakePerformed` returns a neighborhood that modifies a solution by inserting an unperformed visit after a performed one. The function `IloMakePerformedPair` returns a neighborhood that modifies a solution by making an unperformed visit pair performed. For each vehicle route in which the pair is inserted, every combination of positions for the two visits will be tried. This behavior is different from the one of `IloMakePerformed` which will only try to move the pair immediately after a performed pair of visits.

The function `IloMakeUnperformed` returns a neighborhood that modifies a solution by causing a performed visit to be unperformed.

The function `IloSwapPerform` returns a neighborhood that modifies a solution by exchanging a performed visit with an unperformed one.

A P P E N D I X

C

Predefined Search Heuristics and Metaheuristics

Before reading this appendix, you should read the *IBM® ILOG® Solver User's Manual* chapter on local search. Dispatcher uses Solver's basic local search facilities, while defining its own neighborhoods and metaheuristics.

Basic Search Heuristics

Basic heuristics are those which accept only new routing plans that strictly decrease the cost. For Dispatcher, these comprise two search methods, *first accept* and *best accept*. First accept search takes any legal cost decreasing move (the first one encountered), whereas best accept search takes the legal move from the neighborhood that decreases cost by the greatest amount. Both searches continue to take such moves until the neighborhood contains no legal cost-reducing moves. This point is usually termed a local minimum (under the assumption that you are minimizing the objective), as all neighborhood moves are "uphill." The word "local" is used to signify that this point is not guaranteed (and in fact is not usually) a global minimum.

Although the above description entails making improving moves until a local minimum is reached, the local search methods provided by Solver are completely open in the sense that control is returned to user code after each move has been made. Therefore, local search can be stopped at any point along the way should a limit (such as time) be exhausted, or an external event occur requiring the optimization to stop.

First Accept Search

This method accepts any legal improving move and so is not too expensive in terms of computational cost. Thus, it is preferred when the problem is very large or an optimized solution is required as quickly as possible. Dispatcher code to implement a first accept search is shown below. We assume an `IloDispatcher` (`dispatcher`), `IloRoutingSolution` (`solution`), `IloEnv` (`env`), and `IloSolver` (`solver`) are already defined.

The following code creates a goal that instantiates the cost to its minimum value (Dispatcher's constraints only provide a lower bound on the cost) and a neighborhood composed of three basic routing neighborhoods (relocate, exchange, and 2-opt).

```
IloGoal instCost = IloDichotomize(env, dispatcher.getCostVar(), IloFalse);
IloNHood nhood = IloRelocate(env) + IloExchange(env) + IloTwoOpt(env);
```

A greedy heuristic (which allows only improvements in the objective) called `IloImprove` is used to reject all degrading moves. The `IloSingleMove` goal is used to construct a move. As no search selector is passed to it, the first legal move reducing cost results. The `instCost` goal is executed just before testing each move.

```
IloMetaHeuristic improve = IloImprove(env);
IloGoal move = IloSingleMove(env, solution, nhood, improve, instCost);
```

Then a loop is entered, where moves are accepted until no more legal improving ones exist. Finally, the improved solution is moved to Dispatcher's model using the `IloRestoreSolution` goal.

```
while (solver.solve(move));
solver.solve(IloRestoreSolution(env, solution));
```

Best Accept Search

This method accepts the most profitable (in terms of cost) legal improving move and so is more expensive in terms of computational cost than the first accept method just described. For some problems, however, the results can be better than those provided by first accept. You should experiment on your own problem to find the best fit.

In terms of code, performing a best accept search is very much like a first accept search except that a selector is passed to `IloSingleMove` to select the best neighborhood move. Code for implementing a best accept search is shown below:

```
IloGoal instCost = IloDichotomize(env, dispatcher.getCostVar(), IloFalse);
IloNHood nhood = IloRelocate(env) + IloExchange(env) + IloTwoOpt(env);
IloMetaHeuristic improve = IloImprove(env);
IloSearchSelector selBest = IloMinimizeVar(env, dispatcher.getCostVar());
IloGoal move = IloSingleMove(env, solution, nhood, improve, selBest, instCost);
while (solver.solve(move));
solver.solve(IloRestoreSolution(env, solution));
```

MetaHeuristics

The basic search heuristics (which use `IloImprove`) terminate when no cost-reducing move can be found, at a local minimum. However, in many cases we would like to go on searching, hoping to find better solutions than the one at the current local minimum. To do this, we need to allow neighborhood moves that *degrade* the current solution to allow the search to move from the current position. However, the ways in which degrading moves are accepted must be carefully controlled to prevent the search from moving to very poor solutions. This more subtle control is provided by what we term metaheuristics.

Dispatcher provides two basic metaheuristics specialized for routing problems. The first of these is a tabu search metaheuristic and the second one is based on guided local search. Various search mechanisms can be created depending upon the way in which these two basic metaheuristics are combined with search selectors. Tabu search and guided local search can also be combined to produce a hybrid, which might be termed guided tabu search.

The following sections detail ways to generate different search methods using the metaheuristics provided with Dispatcher.

Tabu Search Metaheuristic

A detailed description of the operation of Dispatcher's tabu search metaheuristic is given in the *IBM ILOG Dispatcher Reference Manual*. Here, we give a brief description of its operation.

Objects of the class `IloDispatcherTabuSearch` allow degrading moves to be accepted by the local search process and attempt to avoid moving the search cycle to places it has previously been. To prevent cycling, a metaheuristic could, of course, store every solution it had visited and forbid a return to any such point. However this has a significant memory burden. Therefore, `IloDispatcherTabuSearch` uses the popular technique of the tabu list. The tabu list is a list of "features" of a solution that are forbidden, or alternatively, that must be present. Features are added and dropped from a list when a neighborhood move is made.

In Dispatcher, the features are the arcs of the current solution, and two tabu lists are maintained, one that dictates which arcs must not be part of any new solution, and the other that dictates which arcs must be part of any new solution. By maintaining finite lists, tabu search is encouraged to explore solutions with different arcs. The length of time a feature remains on a list (the tenure) is important and affects how driven the search is to avoid solutions which resemble previous ones. This is a parameter of `IloDispatcherTabuSearch`, and can be altered dynamically during the search process.

Another aspect of `IloDispatcherTabuSearch` is the aspiration criterion, which allows a neighborhood move to be accepted despite being tabu. The purpose of the aspiration criterion is to prevent the search from being overly inhibited by the tabu process. The aspiration criterion in `IloDispatcherTabuSearch` makes it possible to accept any move that improves on the best cost found so far.

Guided Local Search Metaheuristic

Guided local search is an alternative to tabu search for allowing the search process to move out of local minima. As in tabu search, how the search can move around is restricted. Guided local search works by making a series of *greedy* searches, each to a local minimum. However, it reduces a different cost function from the original. If the original cost function is represented by c, then guided local search attempts to reduce the cost $c+wp$, where p is a penalty term that is adjusted every time a local minimum is reached, and w is a constant. So, in essence, guided local search tries to minimize a combination of the true cost and a penalty term. The weighting constant w is an important search parameter that determines how important the penalty term is with respect to the true cost.

Guided local search is constructed so that the penalty term is higher when the search moves to solutions that resemble previous ones, and in this sense it is similar to tabu search. This is done by recording certain arcs as "bad" each time a local minimum is reached. When these arcs appear in a solution, the penalty term is increased. Guided local search determines which arcs are bad based upon the cost of the arc in the solution and how often that arc has previously appeared at local minima. More complete details of the operation of guided local search are given in the *IBM ILOG Dispatcher Reference Manual*.

Using Metaheuristics In Search

The tabu search metaheuristic and the guided local search metaheuristic described above are implemented by the classes `IloDispatcherTabuSearch` and `IloDispatcherGLS.` `IloImprove` can be used to create both first and best accept search strategies depending on the choice of search selector used. Similarly, tabu and guided local search can be combined with selectors (and even each other), to produced various types of metaheuristic search strategies for Dispatcher.

Some templates for search strategies using these two metaheuristics are presented in the following sections. You, however, are free to implement more complex strategies. For example, maintaining a set of good solutions found during search, and occasionally restarting search from one of these can be an effective technique.

The following are several variations on a theme for performing search, although since they are largely similar, only the first is presented in detail.

Tabu Search

A basic tabu search process uses the `IloDispatcherTabuSearch` metaheuristic and takes the best move at each iteration. However, the tabu search metaheuristic is not well suited for taking the first move at each step, as search can move very quickly to poor quality areas of the search space. The following code shows how you might implement a function to perform such a tabu search process. In this code, the tabu tenure is varied during search, which can cut down on the search moving in cycles around the same area.

```
void RoutingSolver::improveWithTabu() {
  _nhood.reset();
  IloRoutingSolution rsol = _solution.makeClone(_env);
  IloRoutingSolution best = _solution.makeClone(_env);
  IloDispatcherTabuSearch dts(_env, 12);
  IloSearchSelector sel = IloMinimizeVar(_env, _cost);
  IloGoal move = IloSingleMove(_env, rsol, _nhood, dts, sel, _instantiateCost);
  move = move && IloStoreBestSolution(_env, best);
  for (IloInt i = 0; i < 150; i++) {
    if (i == 70) dts.setTenure(20);
    if (i == 85) dts.setTenure(5);
    if (i == 105) dts.setTenure(12);
    if (_solver.solve(move))
      _solver.out() << "Cost = " << _solver.getMax(_cost) << endl;
    else
      if (dts.complete()) break;
  }
  _solver.solve(_restoreSolution);
  rsol.end();
  best.end();
  dts.end();
}
```

The function resets the neighborhood and set up the various objects used in the search. Resetting Dispatcher's neighborhoods (and indeed metaheuristics) is always recommended before starting a new search, as neighborhoods and metaheuristics maintain internal data structures which are well defined within a single search, but not if search is restarted from an arbitrary point. As you are creating the tabu search metaheuristic, there is no need to reset it, but the neighborhood is passed here, and so a reset is performed.

Two duplicates are made of the current solution—one to be used during search and one to store the best solution found during search. The first is made as you do not wish to change the solution passed, only to present the improved solution on return from the function. The second is made because with search methods that can accept cost degrading moves, the best solution found is not necessarily the current solution.

The tabu search metaheuristic is created with an initial tenure of 12 moves.

The goal to make a single move is constructed using `IloSingleMove`. Then, a secondary goal (`IloStoreBestSolution`) is attached that stores the new solution as the best one if its cost is better than the previous best.

The optimization loop is then entered, making up to a maximum of 150 moves, or stopping when no move could be made and the metaheuristic returns `IloTrue` from its `complete` method.

The tenure can be dynamically changed during search, and this is demonstrated by changing its value after certain numbers of iterations. Other techniques you can use are to change the tenure to a random value in a range every so many iterations, or to perform such a change after a certain number of iterations without improvement to the best solution found.

Finally, the best solution is restored, and the memory for the temporary solutions reclaimed by using `IloSolution::end()`.

The following figure shows an example of the variation of the cost using such a tabu search on a VRP. The x-axis represents the number of iterations performed.The y-axis represents the value of the cost. The dashed line represents the best cost found so far for a given iteration, while the solid line represents the current cost function. From this figure, you can see the basic descent during the first iterations (when the dashed line is overlapped by the solid line), then degrading moves are used to explore the search space and eventually improve the best solution.

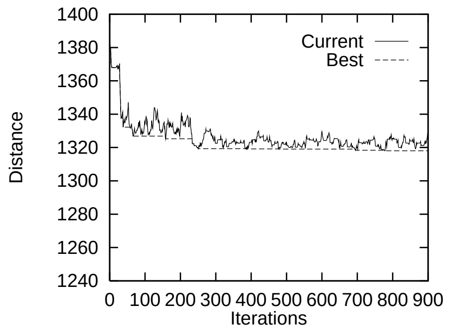

Figure C.1 *Cost Variation: Using Tabu Search on a VRP*

Guided Local Search

There are two basic types of search that can be performed using the guided local search (GLS) metaheuristic, depending upon which type of search selector is used to choose the neighborhood move. You can choose the best legal move at each stage, resulting in the code shown below:

```
void RoutingSolver::improveWithGLS() {
  _nhood.reset();
  IloRoutingSolution rsol = _solution.makeClone(_env);
  IloRoutingSolution best = _solution.makeClone(_env);
  IloDispatcherGLS dgls(_env, 0.2);
  IloSearchSelector sel = IloMinimizeVar(_env, dgls.getPenalizedCostVar());
  IloGoal move = IloSingleMove(_env, rsol, _nhood, dgls, sel,
_instantiateCost);
  move = move && IloStoreBestSolution(_env, best);
  IloCouple(_nhood, dgls);
  for (IloInt i = 0; i < 150; i++) {
    if (_solver.solve(move))
      _solver.out() << "Cost = " << _solver.getMax(_cost) << endl;
    else {
      _solver.out() << "---" << endl;
      if (dgls.complete()) break;
    }
  }
  IloDecouple(_nhood, dgls);
  IloGoal restoreSolution = IloRestoreSolution(_env, best) && _instantiateCost;
  _solver.solve(restoreSolution);
  rsol.end();
  best.end();
  dgls.end();
}
```

There are two peculiarities which are markedly different from the previous tabu search example. The first are the lines of code using `IloCouple` and `IloDecouple`. These functions connect (and disconnect) a neighborhood to a metaheuristic. The neighborhood must be coupled to an instance of `IloDispatcherGLS` otherwise an `IloException` is thrown when you try to use the instance in the search.

The following figure shows the variation of the cost using GLS on the same VRP as the Tabu Search example. The x-axis represents the number of iterations performed. The y-axis represents the value of the cost. The dashed line represents the best cost found so far for a given iteration, while the solid line represents the current cost function. From this figure, you can see the basic descent during the first iterations (when the dashed line is overlapped by the solid line), then degrading moves being used to explore the search space and eventually improve the best solution. What can be seen from this figure is that the “jumps” of the GLS metaheuristic are much larger than those of the basic Tabu Search. This is the consequence of the long-term memory feature of GLS. On this example also, GLS gives better results than Tabu Search.

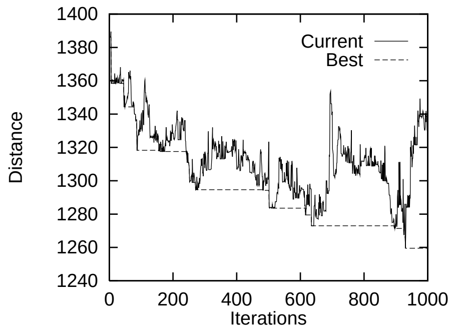

Figure C.2 *Cost Variation: Using GLS on a VRP*

Guided Tabu Search

Simple Tabu Search and Guided Local Search can be mixed together. Arcs present in the solution are penalized just as with GLS, while a tabu list is handled just as in tabu search. Therefore, the short-term memory feature of simple tabu search is enhanced with a long-term memory, acting as a diversifying scheme.This results in a very effective metaheuristic, that can often produce good quality solutions in much fewer iterations than either simple Guided Local Search or Tabu Search.

Index

Numerics

2-opt neighborhood **440**

A

add visits **119**, **131**
alternate sites **119**
arc **85**, **361**
  distances between nodes **362**
arc cost **86**, **362**
array of neighborhoods **130**

B

bays **305**
best accept search **445**, **446**
breaks **196**
building routes **409**

C

capacitated arc routing problem **361**
capacity **26**, **32**
  constraints on **32**
  limited **361**
  measuring **44**
  setting for vehicle **28**
  vehicle **240**
coherence of submodels **242**
compatibility constraints **348**
complex costs **283**
concrete delivery **406**
constrained variable **29**
constraint **26**
  compatibility **348**
  modeling vehicle break **35**
  on capacity **32**
  on cumul variable bounds **33**
  on late deliveries **26**
  on number of vehicles **26**
  on vehicle **194**
  setting up visit disjunctions **36**
  time **241**
  visit disjunction **127**
cost **26**, **37**
  arc **86**
  calculating routing plan **37**
  calculating vehicle **37**
  computing on arcs **362**
  early and late **224**
  fixed **37**
  from total length of route **283**
  minimizing **43**, **239**, **410**
  turn penalty **86**
  vehicle fixed **150**, **153**
  visit penalty **89**, **158**
cost coefficients **279**
cost ratios **365**
cranes **305**

cross neighborhood **442**
csv reader **47**
cumulative variables **30**
current partial route **410**
customer location
  alternative **26**
  definition **27**
  time constraints on **44**
customer support **20**

D

decision variable **29**, **39**
declaring
  length **250**
  time **250**
  weight **250**
decomposition
  problem **241**, **242**, **396**
delay variable **34**
delivery and pickup visits **183**
delta
  solution **424**
depot
  assigning trucks to multiple **239**
  definition **27**
  multiple **26**
depot docking bays **305**
designing Dispatcher models **395**
destructor **245**
dimension **29**, **46**
  associating constrained variable with **29**
  definition **29**
  extrinsic **30**
  intrinsic **30**
disjoint time window **33**
disjunctions
  visits **119**
dispatching technicians **323**, **339**
distance **30**, **46**
  Euclidean **30**, **250**
  grid pattern **30**
  Manhattan **30**
  minimize **240**
docking bays **305**

E

earliness **223**
enumeration heuristic **430**
environment **45**
Euclidean distance **30**, **250**
exchange neighborhood **443**
extrinsic dimension **30**

F

first accept search **61**, **445**, **446**
first solution **40**
  generating **407**, **409**
first solution framework **413**
first solution heuristics **429**
  relative complexity **407**
  writing **409**
first solution hints **407**
fixed cost **37**
  vehicle **150**, **153**

G

geographical decomposition **396**
GLS **367**
granularity **308**
graph functionality **83**
greedy search **448**
grid pattern distance **30**
guided local search **367**, **450**
guided local search (gls) **448**
guided tabu search **452**

H

heuristic **409**
  construction
    enumeration **430**
    insertion **437**
    nearest addition **435**
    nearest-to-depot **434**
    savings **58**, **432**
    sweep **433**

I

IBM ILOG Scheduler **305**
IlcGoal **412**, **420**
IloAny type definition **19**
IloBool type definition **19**
IloConcatenate **130**
IloCouple **451**
ILOCPGOALWRAPPER **412**, **420**
IloCross **92**, **442**
IloCsvReader **47**
IloDecouple **451**
IloDichotomize **57**
IloDimension **29**
IloDimension1 **30**, **32**, **46**
IloDimension2 **30**, **47**
IloDiscreteResource **307**
IloDispatcher **39**, **56**, **410**, **446**
  getCostVar **37**
  getIndex **29**, **38**
IloDispatcher::RouteIterator **42**
  operator* **42**
  operator++ **42**
IloDispatcherGenerate **430**
IloDispatcherGLS **368**
IloDispatcherGraph **83**, **363**
IloDispatcherTabuSearch **447**, **448**
IloDistance **30**
IloEarlinessFunction **226**
IloEnv **45**, **244**, **446**
IloEuclidean **30**
IloExchange **92**, **443**
IloFPRelocate **442**
IloGenerateRoute **410**
IloGoal **56**
IloImprove **61**, **447**
IloInfinity **37**
IloInsertionGenerate **67**, **349**, **438**
IloInsertVisit **159**
IloInstantiateVehicleBreaks **199**
IloInt type definition **19**
IloLimitSearch **199**
IloMakePerformed **92**, **444**
IloMakePerformedPair **444**
IloMakeUnperformed **92**, **444**
IloManhattan **30**
IloModel **39**, **45**, **241**, **397**
IloNearestAdditionGenerate **67**, **436**
IloNearestDepotGenerate **67**, **435**
IloNHood **60**
IloNHoodArray **130**
IloNode **28**, **48**
IloNum type definition **19**
IloNumToNumSegmentFunction **283**
IloOrderedVisitPair **184**
IloOrOpt **61**, **92**, **441**
IloRelocate **92**, **441**
IloRestoreSolution **92**, **446**
IloRoutingSolution **39**, **56**, **90**, **446**
  getNext **39**
  getPrev **39**
  isPerformed **39**
IloRoutingSolution::RouteIterator **42**
  operator* **42**
  operator++ **42**
IloSavingsGenerate **58**, **67**, **432**
IloSetTimesForward **310**
IloSimpleVisitToNumFunction **280**
IloSingleMove **61**, **245**, **400**, **446**, **449**
IloSolution
  end **450**
IloSolver **56**, **446**
IloStoreBestSolution **449**
IloSwapPerform **92**, **444**
IloSweepGenerate **67**, **433**
IloTardinessFunction **226**
IloTwoOpt **61**, **92**, **440**
IloVehicle **50**
  getCapacity **38**
  getCostVar **37**, **38**
  getFirstVisit **29**, **35**, **38**
  getLastVisit **29**, **35**, **38**
  getSpeed **32**, **38**
  setCapacity **33**
  setCost **37**
IloVisit **49**, **52**, **242**, **264**
  getCumul **33**, **34**
  getCumulVar **30**, **31**, **35**, **38**
  getDelayVar **38**
  getNextVar **28**, **38**

getPrevVar **29**, **38**
getTransitVar **38**
getTravelVar **39**
getVehicleVar **29**, **32**, **39**
getWaitVar **39**
performed **29**
IloVisitArray **157**
IloVisitToNumFunction **281**
IloVisitVehicleCompat **348**
improvement loop **259**, **406**
improving
solution **258**, **404**
insertion heuristic **437**
insertion of visits **90**
installing Solver **17**
intrinsic dimension **30**
iterative solution process **242**, **255**, **398**
iterative solving
mixed **242**
iterator **42**

L

lateness **223**
limited
capacity **361**
linking library to application **18**
local minimum
escaping **448**
local search **41**, **56**
loop
improvement **259**, **406**

M

Manhattan distance **30**
metaheuristic **244**, **367**
minimize
cost **410**
distance **240**
minimizing number of vehicles **90**
minimizing the number of vehicles **81**
mixed iterative solving **242**
model **45**, **241**, **397**
modeling hints **395**, **406**
move operators **244**
Multiple Depot Pickup and Delivery Problem **240**
Multiple Depot Vehicle Routing Problem (MDVRP) **239**
improving the solution **258**, **404**
multiple tours heuristic **156**
multiple tours per vehicle **149**

N

naming
accessors **18**
arguments **18**
Boolean accessors **18**
classes **18**
conventions **18**
data members **18**
instances **18**
member functions **18**
modifiers **18**
nearest addition heuristic **435**
nearest-to-depot heuristic **434**
neighborhood **244**
2-opt **440**
cross **442**
exchange **443**
inter-route **59**, **439**
intra-route **59**, **439**
or-opt **440**
relocate **441**
swap **127**
neighborhood array **130**
neighborhoods **60**, **421**
writing **421**
network, road **362**
next-variable
modifying **39**
node **27**
arc **362**
definition **27**
specifying coordinates **27**

O

operators
move **244**

ordered visit pairs **184**
or-opt neighborhood **440**

P

path constraint **31**
penalty cost **158**
  associating with visit **37**
  visit **150**
personal computer (pc)
  linking Solver library on **18**
Pickup and Delivery Problem (PDP) **26**
pickup and delivery visits **183**
problem decomposition **241**, **242**, **396**
problem size **26**

Q

quantities **28**

R

relocate neighborhood **441**
remove visit **119**, **132**
resources **305**
restore solution **61**
road network **362**
route
  building **409**
  building a set of **43**
  current partial **410**
  definition **42**
routing **26**
routing plan **26**
  calculating total cost **37**
  cost variable **37**
routing problem
  multiple depots **240**
routing solution **245**, **401**

S

savings heuristic **58**, **432**
search
  best accept **445**, **446**
  first accept **445**, **446**
  greedy **448**
  guided local **450**
  guided tabu **452**
  tabu **447**
search method
  guided local search (gls) **448**
  local **41**, **56**
shortest path **83**, **87**, **364**
simplify the model **395**
skill levels **324**
skills **339**
solution
  degrading current **447**
  improving **258**, **404**
  improving with degrading moves **447**
  iterative **242**, **255**, **398**
  restore **61**
  routing **245**, **401**
solution delta **424**
speed
  vehicle **195**
submodel
  TSP **160**
submodel coherence **242**
subproblems **241**
support
  customer **20**
swap array
  visits **126**
swap neighborhood **127**
sweep heuristic **433**
synchronization **242**, **245**, **255**, **258**, **398**, **401**, **405**

T

tabu list **447**
tabu search **447**
  aspiration criterion **447**
tardiness **223**
taxi dispatching **406**
technicians
  dispatching **323**, **339**, **397**
temporal decomposition **396**, **397**
time **46**

time constraints **241**
time window **26**, **33**, **240**
  disjoint **34**
  implementing constraint on **33**
  setting for visit **253**
  softening **223**
tour
  multiple **26**
TSP submodel **160**
turn **85**
turn penalty cost **86**
two-phase approach **56**
types in Solver **19**

U

UNIX
  linking Solver library on **18**
unperformed visits **411**, **412**

V

variable
  constrained **29**
  cost **37**
  cumulative **33**
  decision **29**, **39**
  delay **31**, **34**
  prev- **29**
  transit **31**
  travel **31**
  vehicle **29**
  wait **31**
vehicle
  and capacity constraints **44**
  calculating total cost **37**
  cost variable **37**
  definition **28**
  end-visit **29**
  fixed cost **37**
  implementing breaks on **35**
  implementing deadlines on **34**
  limit on number available **223**
  multiple tours **149**
  setting capacity **28**
  start-visit **29**
vehicle break **26**, **193**
  changing value of wait and cumul variables **32**
  definition **35**
  duration **35**, **193**
  implementing **35**
  interrupting customer visit **193**
  position **35**
  start time **35**, **193**
vehicle breaks **196**
vehicle capacity **240**
vehicle fixed cost **150**, **153**
Vehicle Routing Problem (VRP) **26**, **43**
vehicle speed **195**
vehicles **26**
  minimizing number of **81**, **90**
visit **26**
  add **119**
  associating penalty cost **37**
  creating disjunctions **36**
  creating node **253**
  cumul variable **33**
  definition **28**
  delay variable **31**
  identifier **28**
  inserting **159**
  interrupting with vehicle break **193**
  penalty cost **89**
  performed **36**
  performing in route **43**
  pickup and delivery **183**
  prev-variable **29**
  remove **119**
  transit variable **31**
  travel variable **31**
  unperformed **36**, **411**, **412**
  vehicle variable **29**
  wait variable **31**
visit array **155**
visit disjunction constraint **127**
visit disjunctions **119**
visit pairs **184**
visit penalty cost **150**, **158**
visit swap array **126**

W

weight **46**
writing a first solution heuristic **409**