IBM ILOG Solver V6.7

User's Manual

June 2009

Copyright notice

© Copyright International Business Machines Corporation 1987, 2009.

US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Trademarks

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

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

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

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

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

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

Table of Contents

Preface **Welcome to IBM ILOG Solver 21**

About this manual 21
How this manual is organized 22
Prerequisites 22
Related documentation 22
Installing IBM ILOG Solver 23
Linking IBM ILOG Solver 23
Typographic and naming conventions 23
Accessing software support 24
Contact via web 25
Contact via phone 26

Part I **The Basics 28**

Chapter 1 **Constraint Programming with IBM ILOG Solver 29**

The three-stage method 30
Describe 30
Model 31
Decision variables 31
Constraints 31

Solve **32**
Search space 33
Initial constraint propagation 33
Search tree. 34
Search strategy 35
Constraint propagation during search 35
Backtrack 36
Continue with search strategy: Try another branch 37
Solution 38
Compile and test **38**
Review exercises **39**
Suggested answers **40**
Exercise 1 40
Exercise 2 40
Exercise 3 40
Exercise 4 40

Chapter 2 **Modeling and Solving a Simple Problem: Map Coloring** **43**
Describe **43**
Discussion 44
Model. **45**
Solve **48**
Review exercises **50**
Suggested answers **50**
Exercise 1 50
Exercise 2 51
Exercise 3 51
Exercise 4 51
Complete program **52**

Chapter 3 **Using Arrays and Basic Search: Changing Money** **55**
Describe **55**

Discussion 56
Model 57
Solve 60
Review exercises 62
Suggested answers 63
Exercise 1 63
Exercise 2 63
Exercise 3 63
Exercise 4 64
Complete program 66

Chapter 4 Searching with Predefined Goals: Magic Square 69

Describe 69
Discussion 70
Model 71
Solve 74
Review exercises 78
Suggested answers 78
Exercise 1 78
Exercise 2 78
Exercise 3 79
Exercise 4 82
Complete program 83

Chapter 5 Using Objectives: Map Coloring with Minimum Colors 87

Describe 87
Discussion 89
Model 90
Solve 94
Review exercises 96
Suggested answers 97
Exercise 1 97

Exercise 2 97
Exercise 3 97
Exercise 4 98
Complete program 99

Part II More on Modeling. 102

Chapter 6 Using the Distribute Constraint: Car Sequencing 103
Describe 103
Discussion 104
Model 105
Solve 108
Review exercises 110
Suggested answers 111
Exercise 1 111
Exercise 2 111
Exercise 3 111
Exercise 4 113
Complete program 115

Chapter 7 Using Set Variables: Crew Scheduling 117
Describe 117
Discussion 119
Model 120
Parameters of TeamConstraints 124
Solve 128
Review exercises 130
Suggested answers 131
Exercise 1 131
Exercise 2 131
Exercise 3 131
Exercise 4 133

Complete program 134

Chapter 8 Combining Constraints: Bin Packing 139

Describe 139

Discussion 141

Model 142

Solve 148

Review exercises 155

Suggested answers 155

Exercise 1 155

Exercise 2 155

Exercise 3 155

Complete program 157

Chapter 9 Using Table Constraints: Scheduling Teams 161

Describe 161

Discussion 162

Model 163

Solve 170

Review exercises 171

Suggested answers 171

Exercise 1 171

Exercise 2 172

Exercise 3 172

Complete program 172

Chapter 10 Reducing Symmetry: Configuring Racks 177

Describe 177

Recognizing the constraints 178

Determining the optimization criterion 178

Choosing types of racks 178

Model 179

Representing the constraints 180

Implementing the constraints 181

Cost function 183

Solve 183

Complete program 184

Chapter 11 Using Constrained Floating-Point Variables: Modeling Equations 189

Declaring floating-point variables 190

Using floating-point variables in expressions 190

Using predefined functions with floating-point variables 192

Using IloDichotomize. 192

Using IloGenerateBounds 194

Using arrays of floating-point variables 196

Factored and canonical forms of expressions 197

Factored and canonical forms: Example 1 197

Factored and canonical forms: Example 2 200

Factored and canonical forms: Using IloGenerateBounds. 201

IEEE 754 and Solver. 203

Part III More on Solving 206

Chapter 12 Setting Filter Levels: Coloring Graphs 207

Describe 207

Discussion 211

Model. 212

Solve 214

Review exercises 221

Suggested answers 221

Exercise 1 221

Exercise 2 222

Exercise 3 222

Complete program 223

Chapter 13 Controlling the Search: Locating Warehouses 227

Describe . . . 227
Discussion . . . 229
Model . . . 229
Solve . . . 234
Review exercises . . . 247
Suggested answers . . . 247
Exercise 1 . . . 247
Exercise 2 . . . 247
Exercise 3 . . . 248
Exercise 4 . . . 250
Complete program . . . 251

Chapter 14 **Limits and Problem Decomposition: Locating Warehouses** . . . 255
Describe . . . 255
Model . . . 255
Solve . . . 256
Review exercises . . . 267
Suggested answers . . . 267
Exercise 1 . . . 267
Exercise 2 . . . 267
Exercise 3 . . . 267
Complete program . . . 272

Chapter 15 **Searching for Optimal Solutions: Replanning Warehouses** . . . 277
Finding feasible solutions . . . 278
Finding optimal solutions using objectives . . . 278
Storing solutions . . . 280
Multiphase search . . . 281
Multiphase search: Replanning warehouses . . . 283
Using embedded search . . . 284

Chapter 16 **Using Parallel Solver: Multithreaded Warehouse Location** . . . 287
Understanding the architecture . . . 287

Using Parallel Solver . . . 289
Using multiphase search . . . 291

Part IV Extending the Library . . . 296

Chapter 17 Extraction Concepts . . . 297

Extractable objects . . . 297
Extraction process . . . 298
Main rule of extraction: 1 to 1 correspondence . . . 299
Exceptions to the rule: The effect of preprocessing . . . 299
The case of goals . . . 301
Table of correspondences . . . 302
Synchronization of the model . . . 303
Using IloSynchronizeAndRestart . . . 303
Using IloSynchronizeAndContinue . . . 304
Environment and Solver memory allocation . . . 304

Chapter 18 Writing a Goal: Car Sequencing . . . 305

Understanding goals . . . 305
Using goals . . . 306
Subgoals . . . 307
Writing your own goal . . . 309
Using ILCGOALn to define a new class of goals . . . 309
Using ILOCPGOALWRAPPER to wrap the goals . . . 310
Example of writing your own goal: Implementing IlcInstantiate . . . 311
Arguments of goals: Traps and pitfalls . . . 313
Writing a goal: Car sequencing . . . 314
Inputting data . . . 314
Building the model . . . 317
Adding the distribute and sequence constraints . . . 318
Choosing the order of variables and values: Using slack . . . 319
Writing the goal . . . 322

Complete car sequencing program 323

Output 329

Chapter 19 Writing a Constraint: Allocating Frequencies 331

Understanding constraints 332

Elementary modifiers for variables 332

Exp and var: More about modifiers and expressions 333

Invariants: What to propagate? 336

The constraint propagation algorithm 336

Writing your own constraint 337

Implementation classes and handles 340

Using ILOCPCONSTRAINTWRAPPER to wrap constraints 343

Writing your own metaconstraint 343

Programming tips 346

Writing a constraint: Two examples 347

A new class of constraint using demons 348

A new class of constraint using iterators 350

Advanced issues: Propagating constraints after modifying variables 351

Writing a constraint: Frequency allocation 354

Problem representation: Designing the model 355

Search strategy: Choosing values 356

Using the constraint wrapper 357

Defining the Instantiate goal 357

Search strategy: Choice function 358

Search strategy: Algorithm 358

Main program 359

Complete frequency allocation program 360

Chapter 20 Writing a Search Limit: Car Sequencing 365

Understanding search limits 365

Writing your own search limits 366

Writing a search limit: Car sequencing 367

Writing the custom search limit . . . 368
Custom search limits in a Concert Technology model . . . 369
Using the search limit in a goal . . . 370
Complete program . . . 370

Chapter 21 **Using Impacts during Search . . . 373**

Understanding impacts . . . 373
Impact-based search strategies . . . 374
Initialization of impacts . . . 375
Customizable Goal for Integer Variables . . . 375
Variable Filters . . . 375
Value Filters . . . 377
Selection . . . 378
Restarts . . . 378

Chapter 22 **Advanced Modeling with Set Variables: Configuring Tankers . . . 381**

Describe . . . 381
Model . . . 382
Declaring the variables . . . 383
Defining accessors for attributes of set elements . . . 385
Adding constraints . . . 386
Solve . . . 388
Goal to choose truck . . . 389
Goal to choose tank . . . 390
Complete program . . . 391
Output . . . 403

Part V **Local Search . . . 408**

Chapter 23 **Basic Local Search . . . 409**

What is local search? . . . 409
A two-stage approach . . . 410
Solutions . . . 410

Using goals for local search 410
A simple local search procedure 411
Example: Map coloring 411
Finding a first solution 412
Improving the solution 413
Making it easier: Using solution deltas 415
Using solution deltas in the map coloring problem. 416
Using neighborhoods 418
Metaheuristics 420
Example: Using metaheuristics 421
Simulated annealing. 421
Building the model 423
Creating the first solution 423
Creating the metaheuristic object. 423
Scanning the neighborhood 424
Making a single move 424
Searching for a solution 425
Other metaheuristics 425
IloImprove 426
IloTabuSearch 426
Constructing moves 426
The map coloring example revisited 427
More about neighborhoods. 429
Basic neighborhoods 429
Neighborhood modifiers 429
Chapter 24 Writing a Neighborhood. 431
Writing a new neighborhood. 431
Model 432
The IloNHoodI class. 432
The class BubbleI. 435
Writing a new neighborhood modifier. 440

Chapter 25 **Writing a Metaheuristic 443**

Writing a new metaheuristic 443

Virtual functions of IloMetaHeuristicI 444

The class LimitedTabuI 446

Chapter 26 **Combining Complete and Local Search: Locating Warehouses 451**

Describe 451

Model 452

Create the model 452

Read problem data from a file 452

Build constraints and variables 454

Define search functions 455

Solve 457

Create the solution object 457

Use local search. 458

Use complete search for optimization 459

Displaying information about a solution 460

Complete program 460

Output 465

Chapter 27 **Minimizing Talent Wait Cost Using LNS 467**

Problem description 468

Problem representation: Reading the problem data 469

Problem representation: Model 470

Problem solving: Finding an initial solution 473

Problem solving: Large neighborhood search 474

Displaying the solution 477

Complete program 477

Output 481

About this problem 482

Tips on using Large Neighborhood Search 482

The completion goal. 482

The choice of fragment . . . 483
Making the most of neighborhood search . . . 486

Part VI Evolutionary Algorithms . . . 490

Chapter 28 Introduction and Basic Concepts . . . 491

Overview of concepts . . . 491
Genetic algorithms and constraint programming . . . 492
Representation . . . 493
Features of the library . . . 496
Solutions and solution pools . . . 496
Generation of the initial population . . . 497
Evaluating, Comparing, and Selecting Solutions . . . 498
Advanced Topic: Solution Pool (or Multiple) Evaluators . . . 501
Evolution of the population . . . 501
Solution pool processors . . . 501
Choice of parents . . . 503
Genetic operations . . . 504
Solution Replacement . . . 506
Advanced Topic: Monitoring and listeners . . . 507

Chapter 29 Modeling and Solving a Basic EA Problem . . . 509

Creating an initial solution pool . . . 510
Review Exercises . . . 514
Suggested Answers . . . 514
Writing a simple generation loop . . . 514
Review Exercises . . . 518
Suggested Answers . . . 519

Chapter 30 Using More Advanced EA Features . . . 521

Using custom selectors and multiple operators . . . 521
Review Exercises . . . 524
Suggested Answers . . . 525

Using listeners and comparators 525
Review Exercises 533
Suggested Answers 533
Using pool evaluators and partial replacement 533
Review Exercises 539
Suggested Answers 539

Chapter 31 **Bin Packing Using EA** 541
Problem description 541
Problem representation: Model 542
Problem solving: Search goal. 543
Problem solving: Initial population 544
Problem solving: Genetic operators 545
Problem solving: Genetic packing operator 547
Problem solving: Putting it together 550
Complete program 551
Output 555

Chapter 32 **Car Sequencing Using EA** 557
Problem description: Optimization model 557
Problem representation: Cost variable 558
Problem representation: Car priorities 560
Problem solving: Representations and decoding 561
Problem solving: Creating the initial population 565
Problem solving: A genetic algorithm cycle 567
Problem solving: Solution replacement 570
Complete Program 571
Output 578

Part VII **Developing Solver Applications** 580

Chapter 33 **Designing Models** 581
Decrease the number of variables 581

Principle 581
Use dual representation 582
Principle 582
Example 582
Remove symmetries 583
Group by type. 583
Introduce order among variables 583
Use constrained set variables 584
Back propagate costs 585
Principle 585
Example 586
Use appropriate search strategies 587
Principle 587
Example 588
Introduce redundant constraints 590
Principle 590
Example 590
Use objects 591
Principle 591
Use reversible and constrained data members 591
Exploit generic and class constraints 592
Use a known solution for testing 592

Chapter 34 Debugging and Tracing 595

Using a C++ development environment. 595
Tracing demons 595
Stopping on constraint propagation 597
Stopping on failures 597
Trace 598
Trace events 598
Tracing choice points and failures. 599
Tracing variable processing 600

Creating your own trace 607

Chapter 35 Developing Applications 611

Describing the problem 612

Define all constraints 612

Identify soft constraints 613

Separate constraints from strategies 613

Specifying objectives 613

Black-box and user acceptance 614

Response time 614

Interactive applications and user-modifiable constraints 614

Designing the model and prototyping 614

Extract the miniproblem 615

Decompose the model 615

Determine precision 615

Respect semantics 616

Avoid symmetry 616

Prototype rapidly and cleanly 616

Validate the model 617

Choose types of variables 617

Experiment with models 617

Implementing and optimizing 618

Use multiple data sets 618

Optimize propagation and search 619

Look at changes 619

Use active constraints 619

Set granularity 619

Use expertise 620

Preprocess variables 620

Exploit different strategies 620

Optimize locally 621

A few last words 621

Index 623

Welcome to IBM ILOG Solver

IBM® ILOG® Solver is a C++ library for solving complex combinatorial problems in diverse areas including production planning, resource allocation, time tabling, personnel scheduling, cutting materials, blending mixtures, assigning radio frequencies, and many others.

IBM ILOG Solver is based on IBM ILOG Concert Technology. Concert Technology offers a C++ library of classes and functions that enable you to define models for optimization problems and to apply algorithms to those models. Concert Technology supports algorithms for both constraint programming and mathematical programming (including linear programming, mixed integer programming, quadratic programming, and network programming) solutions.

For more information about IBM ILOG Solver and IBM ILOG Concert Technology, see Chapter 1, *Constraint Programming with IBM ILOG Solver*.

About this manual

This is the *IBM ILOG Solver User's Manual*. It is composed of chapters that use a procedural-based learning strategy as well as more conceptual chapters. Each procedural-based 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 IBM ILOG Concert Technology and IBM ILOG Solver installed. You work through the lessons and exercises.

If you are a novice Solver user, start at the beginning of this manual, since the lessons build on each other. If you are a more experienced Solver 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 six parts:

- Part I, *The Basics*
- Part II, *More on Modeling*
- Part III, *More on Solving*
- Part IV, *Extending the Library*
- Part V, *Local Search*
- Part VII, *Developing Solver Applications*

Prerequisites

Solver 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 is required to use this manual.

You should have 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 IBM ILOG Solver before starting to use this manual. For more information, see the sections *Installing IBM ILOG Solver* and *Linking IBM ILOG Solver*.

Related documentation

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

- The *IBM ILOG Solver Reference Manual* fully documents all the Solver 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.

◆ The *IBM ILOG Solver Release Notes* list new and improved features, changes in the library and documentation, and issues addressed for each release.

Installing IBM ILOG Solver

In this manual, it is assumed that you have already successfully installed the IBM ILOG Solver and IBM ILOG 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 Solver. The instructions cover all the details you need to know to install Concert Technology and Solver on your system.

Linking IBM ILOG Solver

When you use Solver, you link the IBM ILOG Solver and IBM ILOG 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 `YourSolverHome/README`; on a PC, you will find comparable information in `YourSolverHome/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 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. For more information, see Part IV, *Extending 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, Solver and Concert Technology isolate 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.

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, *Constraint Programming with IBM ILOG Solver*
- Chapter 2, *Modeling and Solving a Simple Problem: Map Coloring*
- Chapter 3, *Using Arrays and Basic Search: Changing Money*
- Chapter 4, *Searching with Predefined Goals: Magic Square*
- Chapter 5, *Using Objectives: Map Coloring with Minimum Colors*

C H A P T E R

1

Constraint Programming with IBM ILOG Solver

In this lesson, you will learn how to:

- use the three-stage method to describe, model, and solve problems
- identify decision variables and constraints
- understand basic search and constraint propagation
- compile a sample program to make sure your installation is working correctly

IBM® ILOG® Solver is a C++ library that uses constraint programming to find solutions to optimization problems. 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.

IBM ILOG Solver is based on IBM ILOG Concert Technology. Concert Technology is a C++ library that allows you to model optimization problems independently of the algorithms used to solve the problem. It provides an extensible modeling layer adapted to a variety of algorithms ready to use off the shelf. This modeling layer enables you to change your model, without rewriting your application.

Concert Technology supports algorithms for both constraint programming and mathematical programming (including linear programming, mixed integer programming, quadratic programming, and network programming) solutions.

This library is not a new programming language: it lets you use data structures and control structures provided by C++. Thus, the Concert Technology part of an application can be completely integrated with the rest of that application (for example, the graphic interface, connections to databases, and so on) because it can share the same objects.

The three-stage method

To find a solution to a problem using IBM® ILOG® Solver, 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 30.

The second stage is to use Concert Technology classes to *model* the problem. The model is composed of decision variables and constraints. *Decision variables* are the unknown information in a problem. Each decision variable has a *domain* of possible *values*. The *constraints* are limits or restrictions on combinations of values for these decision variables. The model may also contain an *objective,* an expression that can be maximized or minimized. For more information, see the section “Model” on page 31.

The third stage is to use Solver classes to *solve* the problem. Solving the problem consists of finding a value for each decision variable while simultaneously satisfying the constraints and maximizing or minimizing an objective, if one is included in the model. Solver uses two techniques for solving optimization problems: *search strategies* and *constraint propagation*. For more information, see the section “Solve” on page 32.

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

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

Describe

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

What is the unknown information—the decision variables—in this problem?

What are the values of x and y?

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

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

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

Model

The second stage is to use Concert Technology classes to *model* the problem. The model is composed of decision variables and constraints. The model may also contain an objective, although in this case it does not. For more information on modeling with an objective, see Chapter 5, *Using Objectives: Map Coloring with Minimum Colors*.

Decision variables

Decision variables are the unknown information in a problem. Decision variables differ from normal C++ variables in that you can place constraints on the values of decision variables. For this reason, decision variables are also known as *constrained variables*. In this example, the decision variables are x and y.

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

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

Constraints

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

Solve

The third stage is to use Solver classes to search for a solution and *solve* the problem. Solving the problem consists of finding a value for each variable while simultaneously satisfying the constraints and maximizing or minimizing an objective, if one is included in the model. For more information on solving a problem with an objective, see Chapter 5, *Using Objectives: Map Coloring with Minimum Colors*.

Solver uses two techniques to find a solution: search strategies and constraint propagation. Additionally, Solver performs two types of constraint propagation: *initial constraint propagation* and *constraint propagation during search*. Figure 1.1 shows the basic process used by Solver to find a solution. The concepts in the diagram will become clearer as you read later sections.

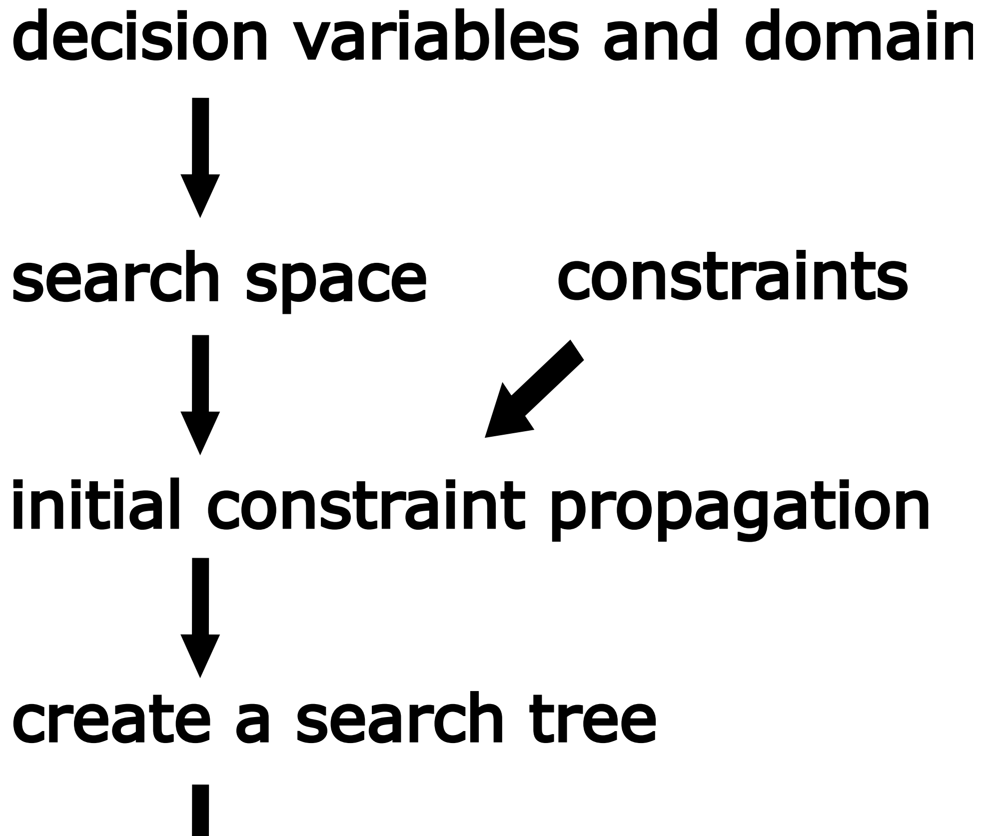

Figure 1.1 *The solve process*

Search space

Solver looks for a solution in the *search space*. The search space is all possible combinations of values. For this simple problem, the search space is shown in Table 1.1. One way to find a solution would be to explicitly study each combination of values until a solution was found. Even for this simple problem, this approach is obviously time-consuming and inefficient. For a more complicated problem with many variables, the approach would be unrealistic.

Table 1.1 *Search space*

x=5, y=2	x=6, y=2	x=7, y=2	x=8, y=2	x=9, y=2	x=10, y=2	x=11, y=2	x=12, y=2
x=5, y=3	x=6, y=3	x=7, y=3	x=8, y=3	x=9, y=3	x=10, y=3	x=11, y=3	x=12, y=3
x=5, y=4	x=6, y=4	x=7, y=4	x=8, y=4	x=9, y=4	x=10, y=4	x=11, y=4	x=12, y=4
x=5, y=5	x=6, y=5	x=7, y=5	x=8, y=5	x=9, y=5	x=10, y=5	x=11, y=5	x=12, y=5
x=5, y=6	x=6, y=6	x=7, y=6	x=8, y=6	x=9, y=6	x=10, y=6	x=11, y=6	x=12, y=6
x=5, y=7	x=6, y=7	x=7, y=7	x=8, y=7	x=9, y=7	x=10, y=7	x=11, y=7	x=12, y=7
x=5, y=8	x=6, y=8	x=7, y=8	x=8, y=8	x=9, y=8	x=10, y=8	x=11, y=8	x=12, y=8
x=5, y=9	x=6, y=9	x=7, y=9	x=8, y=9	x=9, y=9	x=10, y=9	x=11, y=9	x=12, y=9
x=5, y=10	x=6, y=10	x=7, y=10	x=8, y=10	x=9, y=10	x=10, y=10	x=11, y=10	x=12, y=10
x=5, y=11	x=6, y=11	x=7, y=11	x=8, y=11	x=9, y=11	x=10, y=11	x=11, y=11	x=12, y=11
x=5, y=12	x=6, y=12	x=7, y=12	x=8, y=12	x=9, y=12	x=10, y=12	x=11, y=12	x=12, y=12
x=5, y=13	x=6, y=13	x=7, y=13	x=8, y=13	x=9, y=13	x=10, y=13	x=11, y=13	x=12, y=13
x=5, y=14	x=6, y=14	x=7, y=14	x=8, y=14	x=9, y=14	x=10, y=14	x=11, y=14	x=12, y=14
x=5, y=15	x=6, y=15	x=7, y=15	x=8, y=15	x=9, y=15	x=10, y=15	x=11, y=15	x=12, y=15
x=5, y=16	x=6, y=16	x=7, y=16	x=8, y=16	x=9, y=16	x=10, y=16	x=11, y=16	x=12, y=16
x=5, y=17	x=6, y=17	x=7, y=17	x=8, y=17	x=9, y=17	x=10, y=17	x=11, y=17	x=12, y=17

Initial constraint propagation

First, Solver performs an initial constraint propagation. The initial constraint propagation removes all values from domains that will not take part in *any* solution. Before propagation, the domains are:

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

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

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

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

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

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

After initial propagation for both constraints, the search space has been greatly reduced in size. The domains are now:

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

Search tree

After initial constraint propagation, the search space is greatly reduced. This remaining part of the search space, where Solver will use a search strategy to search for a solution, is called the *search tree*. The *root* of the tree is the starting point in the search for a solution; each *branch* descending from the root represents an alternative in the search. Figure 1.2 shows a graphic representation of the search tree for this problem.

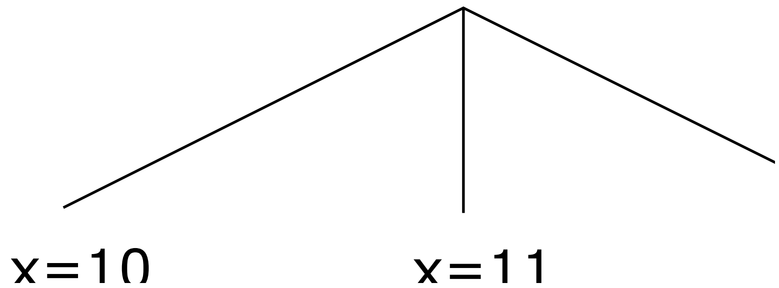

Figure 1.2 Search tree

Search strategy

Solver uses a search strategy to guide the search for a solution. A search strategy is a way to "try" a value for a variable to see if this will lead to a solution. To demonstrate how Solver uses search strategies to find a solution, consider a search strategy that takes variable *x* and assigns it the lowest value in the domain of *x*. For the first *search move* in this strategy, Solver assigns the value 10 to the variable *x*.

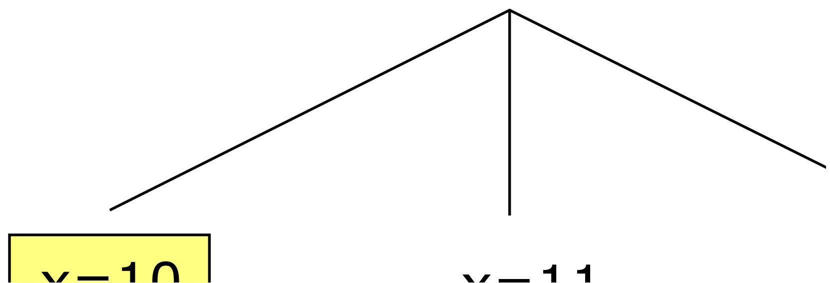

Figure 1.3 Search strategy- assign variable x the lowest value in its domain, 10

Constraint propagation during search

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

Examine the search tree. The search strategy has assigned the value 10 to the variable x. You try the value $y = 5$. This combination of values ($x = 10$, $y = 5$) violates the constraint $x + y = 17$. Solver removes the value $y = 5$ from the current domain of y. Next, you try the value $y = 6$. This combination of values ($x = 10$, $y = 6$) violates the constraint $x + y = 17$. Solver removes the value $y = 6$ from the current domain of y. Next, you try the value $y = 7$. This combination of values ($x = 10$, $y = 7$) satisfies the constraint $x + y = 17$. However, it violates the constraint $x - y = 5$. Solver removes the value $y = 7$ from the current domain of y.

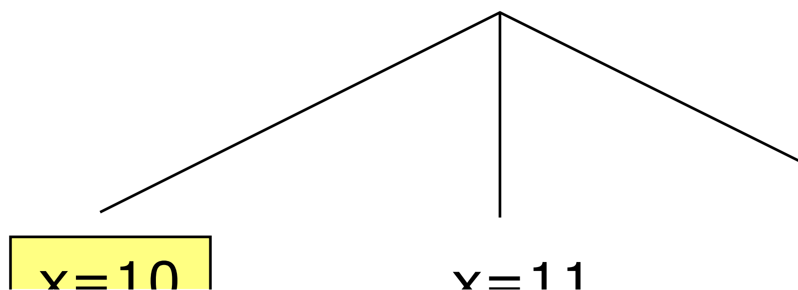

Figure 1.4 *Search tree after constraint propagation during search*

Backtrack

Using constraint propagation during search, Solver has removed the following values from the "test" or current domain of y: 5, 6, and 7. This means that there is no value of y that can combine with the value $x = 10$ and satisfy all the constraints. In other words, Solver has determined that the value $x = 10$ cannot take part in *any* solution. Solver removes the value $x = 10$ from the domain of x—not a "test" or current domain, but the actual domain of x. However, no values of y have been removed from the actual domain of y. They have only been removed from the "test" or current domain used in the search move $x = 10$. These values of y may still take part in a solution with a different value of x.

At this point, Solver needs to *backtrack* up the search tree and try a different value for the variable x. Backtracking gives Solver the flexibility to make search moves that can be wrong. An alternative can be tried and, if it does not succeed, can be reversed.

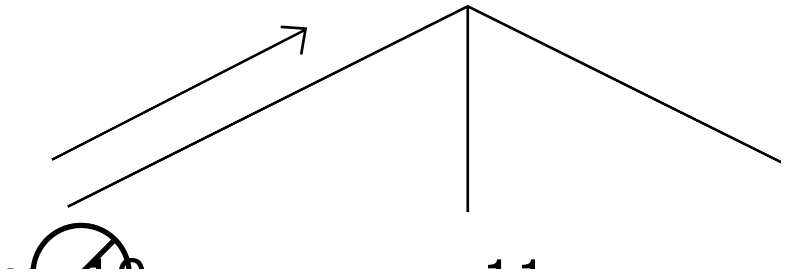

Figure 1.5 *Backtrack*

Continue with search strategy: Try another branch

After backtracking, the branch of the search tree that starts with $x = 10$ has been removed from the search tree. Solver has tried to find a solution in this branch and failed. The next move in the search strategy is to take variable x and assign it the lowest value in the domain of x, which is now `[11 12]`. Solver takes variable x and assigns it the value 11. Solver then searches for a solution in this branch of the search tree. The "test" or current domain of y includes the following values: 5, 6, and 7.

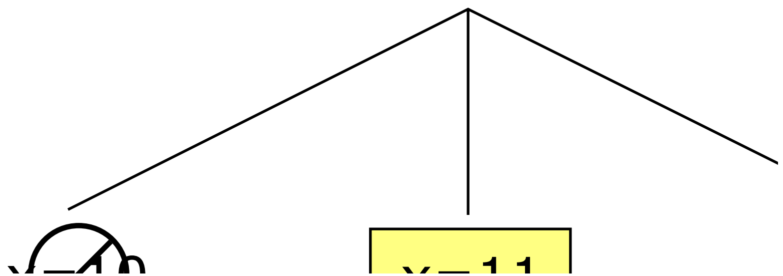

Figure 1.6 *Try another branch in the search tree*

After this second search move, constraint propagation during search again removes values from the "test" or current domains that violate the constraints.

Examine the constraints. The search strategy has assigned the value 11 to the variable x. You try the value $y = 5$. This combination of values ($x = 11$, $y = 5$) violates the constraint $x + y = 17$. Solver removes the value $y = 5$ from the "test" or current domain of y. Next, you try the value $y = 6$. This combination of values ($x = 11$, $y = 6$) satisfies the constraint $x + y = 17$. This combination of values also satisfies the constraint $x - y = 5$.

Solution

Solver has found a solution to the problem using a search strategy and constraint propagation during search. The combination of values ($x = 11$, $y = 6$) satisfies both constraints. There is no need to further explore the search tree because a solution has been found.

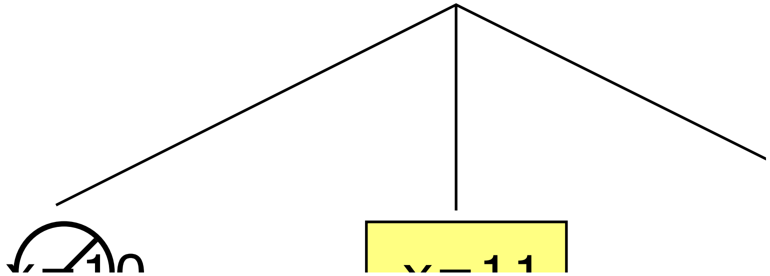

Figure 1.7 Solution

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

The initial constraint propagation removes all values from domains that will not take part in any solution. After initial constraint propagation, the search space is greatly reduced. This remaining part of the search space, where Solver will use a search strategy to search for a solution, is called the search tree. A search strategy is a way to "try" a value for a variable to see if this will lead to a solution. Solver performs constraint propagation during search. Constraint propagation during search removes all values from the current or "test" domains that violate the constraints. If Solver cannot find a solution in one branch of the search tree, Solver backtracks up the search tree. Backtracking gives Solver the flexibility to make search moves that can be wrong. An alternative can be tried and, if it does not succeed, can be reversed. Solver continues to search using the search strategy and constraint propagation during search until a solution is found.

Compile and test

This first lesson is designed to help you understand the basic concepts in constraint programming. In future lessons, you will work through problems by describing, modeling,

and solving problems using IBM® ILOG® Concert Technology and IBM ILOG Solver classes. In this lesson, you are provided with the completed example code so that you can test your installation of Solver.

In Chapter 2, *Modeling and Solving a Simple Problem: Map Coloring*, you will learn about the classes and member functions used in this program.

The following code models and solves the problem introduced in this lesson:

```
#include <ilsolver/ilosolverint.h>
ILOSTLBEGIN

int main(){
  IloEnv env;
  try {
    IloModel model(env);
    IloIntVar x(env, 5, 12);
    IloIntVar y(env, 2, 17);
    model.add(x + y == 17);
    model.add(x - y == 5);
    IloSolver solver(model);
    if (solver.solve()){
      solver.out() << "x = " << solver.getValue(x) << endl;
      solver.out() << "y = " << solver.getValue(y) << endl;
    }
  }
  catch (IloException& ex) {
    cout << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Open the example file `YourSolverHome/examples/src/intro.cpp` in your development environment. To test your installation of Solver, compile and run the program. If you have questions, see *Installing IBM ILOG Solver* in the *Preface* of this manual. When you run the program, you should get the following results:

```
x = 11
y = 6
```

Review exercises

For answers, see “Suggested answers” on page 40.

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

2. What are decision variables?

3. What are constraints?
4. What are the two techniques that Solver uses to find a solution?

Suggested answers

Exercise 1

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

Suggested Answer

1. Describe

   Write a natural language description of problem.

2. Model

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

3. Solve

   Use Solver classes to search for a solution.

Exercise 2

What are decision variables?

Suggested Answer

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

Exercise 3

What are constraints?

Suggested Answer

Constraints are limits on the combinations of values for variables.

Exercise 4

What are the two techniques that Solver uses to find a solution?

Suggested Answer

Solver uses two techniques to find a solution: search strategies and constraint propagation.

CHAPTER

2

Modeling and Solving a Simple Problem: Map Coloring

In this lesson, you will learn how to:

- use the classes `IloEnv` and `IloModel` to create an environment and a model
- use the class `IloIntVar` to declare constrained integer variables
- place simple arithmetic constraints on integer variables
- use the class `IloSolver` to search for solutions

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

Describe

The problem involves choosing colors for the countries on a map in such a way that at most four colors (blue, white, yellow, green) are used, and no neighboring countries are the same color. In this exercise, you will find a solution for a map coloring problem with six countries: Belgium, Denmark, France, Germany, Luxembourg, and the Netherlands. Map coloring problems are closely related to graph coloring problems and have many real world applications. For more information, see Chapter 12, *Setting Filter Levels: Coloring Graphs*.

Figure 2.1 *Map for coloring*

Step 1 Describe the problem

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

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

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

> **Decision variables and constraints**
>
> *Decision variables* are the unknown information in a problem. Each decision variable has a *domain* of possible *values*.
>
> *Constraints* are limits or restrictions on combinations of values for decision variables.
>
> For more information on decision variables and constraints, see the section "Model" on page 31.

Discussion

What is known?

- You have six known countries on a map.

◆ You have four known colors.

What is unknown?

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

What are the constraints?

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

Model

Once you have written a description of your problem, you can use Concert Technology classes to model it. After you create a model of your problem, you can use Solver classes and member functions to search for a solution.

Step 2 Open the example file

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

In this exercise, you use constrained integer variables and therefore you use the include file `<ilsolver/ilosolverint.h>`. To catch exceptions, you use a `try/catch` block. The code for creating the array of names for the color values and for printing out the solution is provided.

> ***Note:*** *When you compile the library with STL support, the macro* `ILOSTLBEGIN` *is equivalent to* `using namespace std`*. On other ports, it is a machine instruction that does nothing.*

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

Step 3 Create the environment

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

```
IloEnv env;
```

An environment, an instance of the class `IloEnv`, manages internal modeling issues. It handles output, memory management for modeling objects, and termination of search algorithms. Normally an application needs only one environment, but you can create as many environments as you wish.

The initialization of the environment creates internal data structures to be used in the rest of the code. Once this is done, you can create a model. A model, an instance of the class `IloModel`, is a container for modeling objects such as variables and constraints. A model is created using an environment.

Step 4 Create the model

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

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

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

> **IloInt**
>
> `IloInt` is a type definition that represents signed integers in Concert Technology.

Concert Technology gives you the means to represent the unknowns in this problem—the color of each country on the map—as constrained integer variables. A constrained integer variable is a variable that takes integers as possible values. The possible values are represented as a domain of integers with an upper bound and a lower bound. The variables are said to be constrained because you can place constraints on them.

Constrained integer variables are represented by the class `IloIntVar` in Concert Technology. The first parameter of the class `IloIntVar` constructor is always the environment. The second parameter is the lower bound of the domain of possible values, which defaults to 0. The third parameter is the upper bound of the domain of possible values. The upper bound defaults to `IloIntMax`, which represents the largest possible positive

integer on a given platform. The fourth parameter is an optional name used for debug and trace purposes. Here is a constructor:

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

After you create an environment and a model, you declare the constrained integer variables, one for each country. Each variable represents the unknown information—the color of the country. Each constrained integer variable takes a value in the domain of integers from `0` to `3`. These values represent the possible colors:

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

These decision variables will contain the solution to the problem, once it is solved.

Step 5 Declare the decision variables

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

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

Concert Technology allows you to express constraints involving integer variables using the following operators:

- equality `==`
- less than or equal to `<=`
- less than `<`
- greater than or equal to `>=`
- greater than `>`
- not equal to `!=`

In this example, you use a constraint to require that when two countries are neighbors, they cannot be the same color. For example, the statement `Belgium != France` indicates that France and Belgium are neighbors, so they should not share the same color. Explicitly, it means that the value Solver finds for the constrained integer variable `Belgium` cannot equal the value Solver finds for the constrained integer variable `France`.

You use the member function `IloModel::add` to add constraints to the model. You must explicitly add a constraint to the model or Solver will not be able to use it in the search for a solution.

Step 6 Add the constraints

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

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

Solve

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

How Solver finds a solution

Solver uses two techniques to find a solution: search strategies and constraint propagation.

For more information, see the section “Solve” on page 32.

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

Step 7 Create an instance of IloSolver

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

```
IloSolver solver(model);
```

You now use the member function `IloSolver::solve`, which solves a problem by using a *default goal* to launch the search. Goals are the mechanism by which Solver implements search strategies. But how do you decide what search strategy to use? The default goal defines the search strategy when you do not specify a goal. Using a default goal allows you

to concentrate on modeling the problem, without knowing the details of the search strategy. The default goal is one of a set of *predefined goals* that Solver offers (more on this in Chapter 4, *Searching with Predefined Goals: Magic Square*). You can also write your own goals (more on this in Part IV, *Extending the Library*).

Step 8 Search for a solution

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

```
if (solver.solve())
```

The member function `IloSolver::solve` returns a Boolean value of type `IloBool`. If a solution is found, an `IloTrue` value is returned and the program displays the solution.

IloBool

`IloBool` is a type definition that represents Boolean values in Concert Technology.

Those values are `IloTrue` and `IloFalse`. Booleans are, in fact, integers of type `IloInt`. `IloFalse` is 0 (zero), and `IloTrue` is 1 (one).

The member function `IloSolver::getValue` returns the value Solver found for the variable. When you print the solution, you associate each value with the name of a color using the array `Names[]`. The value `0` is associated with `Names[0]`, which is the first array element “blue”; the value `1` is associated with `Names[1]`, which is the second array element “white”; and so on. The array `Names[]` is already created for you in the exercise code.

The member function `IloAlgorithm::getStatus` returns a status indicating information about the solution that Solver has found. `IloAlgorithm::out` is the communication stream for general output. These general purpose functions and streams are members of `IloAlgorithm`, the base class for algorithms in Concert Technology. You can use `IloAlgorithm` to employ different algorithms in solving problems represented with Concert Technology modeling classes. `IloSolver` is a subclass of `IloAlgorithm`.

```
solver.out() << solver.getStatus() << " Solution" << endl;
solver.out() << "Belgium:     "
             << Names[(IloInt)solver.getValue(Belgium)]     << endl;
solver.out() << "Denmark:     "
             << Names[(IloInt)solver.getValue(Denmark)]     << endl;
solver.out() << "France:      "
             << Names[(IloInt)solver.getValue(France)]      << endl;
solver.out() << "Germany:     "
             << Names[(IloInt)solver.getValue(Germany)]     << endl;
solver.out() << "Luxembourg:  "
             << Names[(IloInt)solver.getValue(Luxembourg)]  << endl;
solver.out() << "Netherlands: "
             << Names[(IloInt)solver.getValue(Netherlands)] << endl;
```

> ***Note:*** *If there is more than one set of values for the variables that satisfy the constraints of the problem, there is more than one solution. Within your problem, there may be certain criteria that make one such set of values more appropriate than another as a solution. This appropriateness is usually measured in terms of a cost function that can be optimized. You can read more on this in Chapter 5, Using Objectives: Map Coloring with Minimum Colors.*

Step 9 Compile and run the program

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

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

As you can see, all four colors are used.

The complete program is listed in “Complete program” on page 52. You can also view it online in the `YourSolverHome/examples/src/color.cpp` file.

Review exercises

1. What is a constrained integer variable?
2. What are goals?
3. Change the code so that Germany and Denmark are always the same color. Except for Germany and Denmark, no two neighboring countries are the same color.
4. Change the original code to include Switzerland in the map. Switzerland shares borders with France and Germany. No two neighboring countries are the same color.

Suggested answers

Exercise 1

What is a constrained integer variable?

Suggested Answer

A constrained integer variable is a decision variable that takes integers as possible values. The possible values are represented as a domain of integers with an upper bound and a lower bound.

Exercise 2

What are goals?

Suggested Answer

Goals are the mechanism by which Solver implements search strategies.

Exercise 3

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

Suggested Answer

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

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

You should obtain the following result:

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

Exercise 4

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

Suggested Answer

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

The declaration of the variables is changed as follows:

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

These additional constraints are added to the model

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

This code is added to the solution display:

```
  solver.out() << "Switzerland:  "
               << Names[(IloInt)solver.getValue(Switzerland)] << endl;
```

You should obtain the following results:

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

Complete program

The complete map coloring program follows. You can also view it online in the file `YourSolverHome/examples/src/color.cpp`.

```
#include <ilsolver/ilosolverint.h>
ILOSTLBEGIN

const char* Names[] = {"blue", "white", "yellow", "green"};

int main(){
  IloEnv env;
  try {
    IloModel model(env);
    IloIntVar Belgium(env, 0, 3), Denmark(env, 0, 3),
      France(env, 0, 3), Germany(env, 0, 3),
      Luxembourg(env, 0, 3), Netherlands(env, 0, 3);
    model.add(Belgium != France);
    model.add(Belgium != Germany);
    model.add(Belgium != Netherlands);
    model.add(Belgium != Luxembourg);
    model.add(Denmark != Germany );
    model.add(France != Germany);
```

```
    model.add(France != Luxembourg);
    model.add(Germany != Luxembourg);
    model.add(Germany != Netherlands);
    IloSolver solver(model);
    if (solver.solve())
      {
      solver.out() << solver.getStatus() << " Solution" << endl;
      solver.out() << "Belgium:     "
                   << Names[(IloInt)solver.getValue(Belgium)]     << endl;
      solver.out() << "Denmark:     "
                   << Names[(IloInt)solver.getValue(Denmark)]     << endl;
      solver.out() << "France:      "
                   << Names[(IloInt)solver.getValue(France)]      << endl;
      solver.out() << "Germany:     "
                   << Names[(IloInt)solver.getValue(Germany)]     << endl;
      solver.out() << "Luxembourg:  "
                   << Names[(IloInt)solver.getValue(Luxembourg)]  << endl;
      solver.out() << "Netherlands: "
                   << Names[(IloInt)solver.getValue(Netherlands)] << endl;
    }
  }
  catch (IloException& ex) {
    cout << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Results

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

C H A P T E R

3

Using Arrays and Basic Search: Changing Money

In this lesson, you will learn how to:

- use the class `IloIntVarArray` to declare arrays of constrained integer variables
- use the function `IloScalProd`
- use the member functions `IloSolver::startNewSearch`, `IloSolver::next`, and `IloSolver::endSearch` to control the search for solutions

Describe

Assume that you want to pay for an item that costs 1.23 euros. You have a mixture of coins of the following types: 1 euro cent, 10 euro cents, 20 euro cents, and 1 euro. To make the problem more interesting, assume that you have only 5 coins of 1 euro cent. You want to identify all possible solutions to this problem—all possible combinations of coins that will add up to 1.23 euros.

Step 1 Describe the problem

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

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

Discussion

Try modeling the problem as an equation. Where a is the number of coins of 1 cent, b is the number of coins of 10 cents, c is the number of coins of 20 cents and d is the number of 1 euro coins:

$$123 = (a*1) + (b*10) + (c*20) + (d*100)$$

What is known?

- There are four types of coins: 1 cent, 10 cents, 20 cents, and 1 euro (or 100 cents).

What is unknown?

- The number of coins of each type needed to make the purchase.

What are the constraints?

- The sum of the coins must equal 1.23 euros (or 123 cents).
- The number of coins of 1 cent must be less than or equal to 5.

By describing the unknown information as the number of coins of each type, you are grouping the unknowns into types. This removes symmetric solutions from the model and thus significantly cuts back the number of possible (symmetric) solutions.

Reduce symmetry

The apparent complexity of a problem can often be reduced to a much smaller *practical complexity* by detecting intrinsic symmetries. One way to reduce symmetry in a problem is to group variables by type.

Group by type

When two or more variables have identical characteristics, it is pointless to differentiate them. It is better to design a model that takes into account the *types* into which these symmetric variables can be grouped. Each such type, of course, *quantitatively* handles the elements that belong to it.

To demonstrate this concept in simple terms, think about the coins of 1 cent. You can use up to 5 coins of 1 cent. If you have 5 individual coins and you want to track whether each individual coin is used, you need to create 5 different decision variables, one to represent each coin. However, it is not important to know whether any individual coin is used or not. It is important to know how many of each type of coin are used. Therefore, you need only create one decision variable to represent the number of 1 cent coins.

By grouping by types, you cut down on the number of variables and improve the efficacy of your solution search. This is just one example of how thoughtful modeling practices can improve search results.

Model

Once you have written a description of your problem, you can use Concert Technology classes to model it. Before declaring the variables and adding constraints, you must create an environment and a model. Once you create a model of your problem, you can use Solver classes to search for a solution.

Step 2 Open the example file

Open the example file `YourSolverHome/examples/src/tutorial/money_partial.cpp` in your development environment. In this exercise, you use arrays of constrained integer variables and therefore you use the include file `<ilsolver/ilosolverint.h>`. The code for printing out the solution is provided. The number of types of coins, `nCoins`, is set to 4 in this problem and the amount of the purchase, `Sum`, is set to 123.

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

Step 3 Create the environment

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

```
IloEnv env;
```

Step 4 Create the model

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

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

Concert Technology gives you the means to represent the unknowns in this problem—the number of coins of each type—as an array of constrained integer variables. You associate one constrained integer variable in the array with each type of coin.

Arrays of constrained integer variables are represented by the class `IloIntVarArray` in Concert Technology. The first parameter of the constructor is the environment. The second parameter is the number of variables in the array. Arrays are extensible, so you can later

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

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

After you create an environment and a model, you declare the array of constrained integer variables. Each variable in the array represents the unknown information—the number of coins of that type. These decision variables will contain the solutions to the problem, once it is solved.

Step 5 Declare the decision variables

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

```
IloIntVarArray coins(env, nCoins, 0, Sum);
```

The number of variables in the array can be no larger than the number of different types of coins, `nCoins`, which is set to 4 in this problem. The lower bound of the domain of possible values for each variable is 0, meaning that no coin of that type is used. The upper bound of the domain of possible values for each variable is `Sum`, set to 123 in this problem, which would be the largest possible value if the entire purchase amount were composed only of 1 cent coins. (This is not a possible solution in this problem because there is a constraint that only 5 coins of this type are available.)

Now you add the constraint that the sum of the coins must equal 1.23 euros (or 123 cents). To express this constraint, you use the Concert Technology function `IloScalProd`. `IloScalProd` takes two arguments, both arrays, and returns an instance of `IloExprBase` that represents the scalar product of the two arrays. The scalar product is also called the inner product or the weighted sum. The class `IloExprBase` is used internally by Concert Technology to build expressions. You should not use `IloExprBase` directly. Here is a constructor for `IloScalProd`:

```
IloExprBase IloScalProd(const IloNumArray vals, const IloNumVarArray vars);
```

IloNum

`IloNum` is a type definition that represents numeric values as floating-point numbers in Concert Technology. By its nature, `IloNum` can include `IloInt`.

Likewise, the class `IloIntArray` is a subclass of `IloNumArray` and the class `IloIntVarArray` is a subclass of `IloNumVarArray`.

You will learn more about using floating-point numbers and variables in *Chapter 11, Using Constrained Floating-Point Variables: Modeling Equations.*

You use the array of constrained variables that you just created, `coins`, as one argument. The array represents the number of each type of coin. What are you going to use as the other argument? You need to create another array that represents the value of each type of coin. Since you know the value of each type of coin, this array is not an array of variables, but instead an array of integer values. This array represents the coefficients of the equation you used to model the problem.

Arrays of integer values are represented in Concert Technology by the class `IloIntArray`. The first parameter of the constructor is the environment. The second parameter is the number of integer values in the array, which is extensible. The elements of the new array take the corresponding values: *v0, v1, ..., v(n-1)*. When you use an array, you can access a value in that array by its index, and the operator`[]` is overloaded for this purpose. Here is a constructor:

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

Step 6 Declare the array of coefficients

Add the following code after the comment `//Declare the array of coefficients`

```
IloIntArray coeffs(env, nCoins, 1, 10, 20, 100);
```

Now that you have declared the array of variables `coins` and the array of coefficients `coeffs` that are used in `IloScalProd`, you can add constraints to the model. You must explicitly add a constraint to the model or Solver will not be able to use it in the search for a solution.

Step 7 Add the constraint using IloScalProd

Add the following code after the comment `//Add the constraint using IloScalProd`

```
model.add(IloScalProd(coins, coeffs) == Sum);
```

Now you add the second constraint that the number of coins of 1 cent must be less than or equal to 5. You used the operator `!=` in Chapter 2, *Modeling and Solving a Simple Problem: Map Coloring* to express the idea that two or more variables cannot assume the same value. In this lesson, you use the operator `<=` to express the idea that the number of coins of 1 cent must be less than or equal to 5.

Step 8 Add the constraint on the number of 1 cent coins

Add the following code after the comment
`//Add the constraint on the number of 1 cent coins`

```
model.add(coins[0] <= 5);
```

Solve

Solving a problem using constraint programming consists of finding a value for each variable that satisfies the constraints. In this problem, there will be more than one solution that satisfies the constraints. You can use Solver to find multiple solutions for the problem and display them.

You start by creating an instance of the class `IloSolver` to solve the problem expressed in the model.

Step 9 Create an instance of IloSolver

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

```
IloSolver solver(model);
```

In Chapter 2, *Modeling and Solving a Simple Problem: Map Coloring*, you used the member function `IloSolver::solve` to search for a solution. In this lesson, you want to search for all the possible solutions so you will use three other member functions of the class `IloSolver` to do this. You first use the member function `IloSolver::startNewSearch`, which uses a default goal to launch the search.

Step 10 Search for a solution

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

```
solver.startNewSearch();
```

Next, you use a `while` loop and the member function `IloSolver::next` to search for multiple solutions. `IloSolver::next` searches for the next solution in the search tree.

The search space and search tree

The *search space* is all possible combinations of values. When you model a problem, aim to reduce the search space. In this lesson, you reduced symmetry and limited the number of constrained variables. By doing so, you reduced the size of the search space.

After initial constraint propagation, the search space is greatly reduced. This remaining part of the search space is called the *search tree*. Solver uses search strategies to search for a solution in the search tree.

For more information on the search space, the search tree, and initial constraint propagation, see the section "Solve" on page 32.

As you learned in Chapter 2, *Modeling and Solving a Simple Problem: Map Coloring*, Solver uses search strategies to guide the search. Goals are the mechanism by which Solver implements search strategies.

The member function `IloSolver::next`, like `IloSolver::solve`, controls the execution of goals. The first time one of these member functions is called, it creates a *goal stack*, an internal data structure that Solver uses to manage goals during a solution search.

The member function `IloSolver::next` returns a Boolean value of type `IloBool`. If a solution is found, an `IloTrue` value is returned and the program displays the solution.

After the `while` loop terminates, you must use the member function `IloSolver::endSearch` to terminate the search and delete internal objects created by Solver to carry out the search (such as the search tree, goal stack, and so on).

The member functions and streams `IloAlgorithm::out` and `IloSolver::getValue` are used to display the solution. The following code is provided for you:

```
IloInt solutionCounter = 0;
while(solver.next()) {
  solver.out() << "solution  " << ++solutionCounter << ":\t";
  for (IloInt i = 0; i < nCoins; i++)
    solver.out() << solver.getValue(coins[i]) << " ";
  solver.out() << endl;
}
solver.endSearch();
```

Step 11 Compile and run the program

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

```
[1, 10, 20, 100]
solution  1:    3 0 1 1
solution  2:    3 0 6 0
solution  3:    3 2 0 1
solution  4:    3 2 5 0
solution  5:    3 4 4 0
solution  6:    3 6 3 0
solution  7:    3 8 2 0
solution  8:    3 10 1 0
solution  9:    3 12 0 0
```

As you can see, there are nine solutions:

- a solution that uses 3 coins of 1 cent, 1 coin of 20 cents, and 1 coin of 1 euro
- a solution that uses 3 coins of 1 cent and 6 coins of 20 cents
- a solution that uses 3 coins of 1 cent, 2 coins of 10 cents, and 1 coin of 1 euro
- a solution that uses 3 coins of 1 cent, 2 coins of 10 cents, and 5 coins of 20 cents
- a solution that uses 3 coins of 1 cent, 4 coins of 10 cents, and 4 coins of 20 cents
- a solution that uses 3 coins of 1 cent, 6 coins of 10 cents, and 3 coins of 20 cents
- a solution that uses 3 coins of 1 cent, 8 coins of 10 cents, and 2 coins of 20 cents
- a solution that uses 3 coins of 1 cent, 10 coins of 10 cents, and 1 coin of 20 cents
- a solution that uses 3 coins of 1 cent and 12 coins of 10 cents

The complete program is listed in "Complete program" on page 66. You can also view it online in the file `YourSolverHome/examples/src/money.cpp`.

Review exercises

For answers, see "Suggested answers" on page 63.

1. What are the search space and the search tree?
2. Why should you reduce symmetry in a model?

3. Change the code to solve the following problem. Assume that you want to pay for an item that costs 1.23 euros. You have a mixture of coins: 1 euro cent, 2 euro cents, 5 euro cents, 10 euro cents, 20 euro cents, and 1 euro. To make the problem more interesting, assume that you have only 5 coins of 1 euro cent, 5 coins of 2 euro cents, and 10 coins of 5 euro cents. You must also use at least 1 coin of 1 euro (100 cents).

4. Use what you have learned in this lesson to model and solve the following problem. You are the manager of a coffee shop. You sell a house blend that is composed of the following five types of coffee: Costa Rican (cost per unit is $2), Hawaiian (cost per unit $5), Jamaican (cost per unit $3), Kenyan (cost per unit $9), and Nicaraguan (cost per unit $4). You need to make a batch of the house blend. In this batch, you must use at least 2 units of each type of coffee. You can use no more than 3 units of the Costa Rican coffee, no more than 3 units of the Jamaican coffee, and no more than 4 units of the Kenyan coffee. You want the total cost for the batch of house blend to be equal to $100. Write a natural language description of this problem and then model and solve it.

Suggested answers

Exercise 1

What are the search space and the search tree?

Suggested Answer

The search space is all possible combinations of values. After initial constraint propagation, the search space is greatly reduced. The part of the search space that remains is called the search tree. Solver uses search strategies to search for a solution in the search tree.

Exercise 2

Why should you reduce symmetry in a model?

Suggested Answer

By detecting and removing intrinsic symmetries, parts of the search tree can be safely ignored. This reduces the computational effort required to solve the problem.

Exercise 3

Assume that you want to pay for an item that costs 1.23 euros. You have a mixture of coins: 1 euro cent, 2 euro cents, 5 euro cents, 10 euro cents, 20 euro cents, and 1 euro. To make the problem more interesting, assume that you have only 5 coins of 1 euro cent, 5 coins of 2 euro cents, and 10 coins of 5 euro cents. You must also use at least 1 coin of 1 euro (100 cents).

Suggested Answer

The code that has changed from `money.cpp` follows. You can view the complete program online in the file `YourSolverHome/examples/src/money_ex3.cpp`.

The coefficients change as follows:

```
IloInt nCoins = 6, Sum = 123;
IloIntArray coeffs(env, nCoins, 1, 2, 5, 10, 20, 100);
```

These additional constraints are added to the model:

```
model.add(coins[1] <= 5);
model.add(coins[2] <= 10);
model.add(coins[5] >= 1);
```

You should obtain the following results:

```
[1, 2, 5, 10, 20, 100]
solution  1:    0 4 1 1 0 1
solution  2:    0 4 3 0 0 1
solution  3:    1 1 0 0 1 1
solution  4:    1 1 0 2 0 1
solution  5:    1 1 2 1 0 1
solution  6:    1 1 4 0 0 1
solution  7:    2 3 1 1 0 1
solution  8:    2 3 3 0 0 1
solution  9:    3 0 0 0 1 1
solution  10:   3 0 0 2 0 1
solution  11:   3 0 2 1 0 1
solution  12:   3 0 4 0 0 1
solution  13:   3 5 0 1 0 1
solution  14:   3 5 2 0 0 1
solution  15:   4 2 1 1 0 1
solution  16:   4 2 3 0 0 1
solution  17:   5 4 0 1 0 1
solution  18:   5 4 2 0 0 1
```

Exercise 4

Use what you have learned in this lesson to model and solve the following problem. You are the manager of a coffee shop. You sell a house blend that is composed of the following five types of coffee: Costa Rican (cost per unit is $2), Hawaiian (cost per unit $5), Jamaican (cost per unit $3), Kenyan (cost per unit $9), and Nicaraguan (cost per unit $4). You need to make a batch of the house blend. In this batch, you must use at least 2 units of each type of coffee. You can use no more than 3 units of the Costa Rican coffee, no more than 3 units of the Jamaican coffee, and no more than 4 units of the Kenyan coffee. You want the total cost for the batch of house blend to be equal to $100. Write a natural language description of this problem and then model and solve it.

Suggested Answer

Describe

- The known information is that there are 5 types of coffee. You know the price per unit of each type of coffee. The unknown information is the number of units of each type of coffee needed to make the batch of house blend. The constraints are: the total cost of the house blend equals $100; you must use at least 2 units of each type; and you can use no more than 3 units of Costa Rican coffee, no more than 3 units of Jamaican coffee, and no more than 4 units of the Kenyan coffee.

Model

- As in the changing money problem, you model the decision variables—the number of units of each type of coffee—using `IloIntVarArray`. You create an array of coefficients, which represent the price of each type of coffee. You use these two arrays as arguments of `IloScalProd` to constrain that the total cost for the house blend equals $100. You use `IloModel::add` to add the constraints on how many units of each type are allowed in the house blend to the model.

Solve

- You solve the problem in the same way as you solved the changing money example, using `IloSolver::startNewSearch`, `IloSolver::next`, and `IloSolver::endSearch`.

The complete program follows. You can also view it online in the file `YourSolverHome/examples/src/money_ex4.cpp`.

```
#include <ilsolver/ilosolverint.h>
ILOSTLBEGIN

int main() {
  IloEnv env;
  try {
    IloModel model(env);
    IloInt nCoffee = 5, Sum = 100;
    IloIntArray coeffs(env, nCoffee, 2, 5, 3, 9, 4);
    env.out() << coeffs << endl;
    IloIntVarArray units(env, nCoffee, 0, Sum);
    model.add(units[0] >= 2);
    model.add(units[1] >= 2);
    model.add(units[2] >= 2);
    model.add(units[3] >= 2);
    model.add(units[4] >= 2);
    model.add(units[0] <= 3);
    model.add(units[2] <= 3);
    model.add(units[4] <= 4);
    model.add(IloScalProd(units, coeffs) == Sum);
    IloSolver solver(model);
    solver.startNewSearch();
    IloInt solutionCounter = 0;
    while(solver.next()) {
      solver.out() << "solution  " << ++solutionCounter << ":\t";
```

```
      for (IloInt i = 0; i < nCoffee; i++)
        solver.out() << solver.getValue(units[i]) << " ";
      solver.out() << endl;
    }
    solver.endSearch();
  }
  catch (IloException& ex) {
    cout << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Results

```
[2, 5, 3, 9, 4]
solution 1:     2 2 2 8 2
solution 2:     2 3 2 7 3
solution 3:     2 4 2 6 4
solution 4:     2 5 3 6 2
solution 5:     2 6 3 5 3
solution 6:     2 7 3 4 4
solution 7:     2 11 2 3 2
solution 8:     2 12 2 2 3
solution 9:     3 2 3 7 3
solution 10:    3 3 3 6 4
solution 11:    3 7 2 5 2
solution 12:    3 8 2 4 3
solution 13:    3 9 2 3 4
solution 14:    3 10 3 3 2
solution 15:    3 11 3 2 3
```

Complete program

The complete changing money program follows. You can also view it online in the file `YourSolverHome/examples/src/money.cpp`.

```
#include <ilsolver/ilosolverint.h>
ILOSTLBEGIN

int main() {
  IloEnv env;
  try {
    IloModel model(env);
    IloInt nCoins = 4, Sum = 123;
    IloIntArray coeffs(env, nCoins, 1, 10, 20, 100);
    env.out() << coeffs << endl;
    IloIntVarArray coins(env, nCoins, 0, Sum);
    model.add(coins[0] <= 5);
    model.add(IloScalProd(coins, coeffs) == Sum);
    IloSolver solver(model);
    solver.startNewSearch();
    IloInt solutionCounter = 0;
```

```
    while(solver.next()) {
      solver.out() << "solution  " << ++solutionCounter << ":\t";
      for (IloInt i = 0; i < nCoins; i++)
        solver.out() << solver.getValue(coins[i]) << " ";
      solver.out() << endl;
    }
    solver.endSearch();
  }
  catch (IloException& ex) {
    cout << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Results

```
[1, 10, 20, 100]
solution  1:    3 0 1 1
solution  2:    3 0 6 0
solution  3:    3 2 0 1
solution  4:    3 2 5 0
solution  5:    3 4 4 0
solution  6:    3 6 3 0
solution  7:    3 8 2 0
solution  8:    3 10 1 0
solution  9:    3 12 0 0
```

C H A P T E R

4

Searching with Predefined Goals: Magic Square

In this lesson, you will learn how to:

- use expressions with `IloExpr`
- use the predefined constraint `IloAllDiff`
- use the predefined goal `IloGenerate` and its parameters
- use the member function `IloSolver::printInformation` to display details about the solution search

Describe

The problem in this lesson is to find the number in each position in a magic square. A magic square is a square matrix where the sum of every row, column, and main diagonal is the same value; the numbers in the magic square are consecutive and start with 1. For example, here is a magic square of size 3:

8	3	4
1	5	9

8	3	4
6	7	2

Step 1 Describe the problem

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

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

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

Discussion

As in Chapter 3, *Using Arrays and Basic Search: Changing Money*, try modeling the problem using equations. Consider a magic square of size n. It contains n^2 elements, so that the sum of all its elements is the sum of the first n^2 positive integers. The sum of the first n^2 natural numbers can be calculated like this: $n^2(n^2 + 1)/2$.

Because this is a magic square, each row has the same sum, r. The number of rows equals n. The sum of all the rows, nr, also equals the sum of all the elements. Therefore, $nr = n^2(n^2 + 1)/2$ or $r = n(n^2 + 1)/2$. The sum of each column or main diagonal must equal the sum of each row. Therefore, all rows, columns, and main diagonals have a sum of $n(n^2 + 1)/2$.

For example, a magic square of size $n = 3$ contains n^2 or 9 elements. These elements are the first nine natural numbers: 1, 2, 3, 4, 5, 6, 7, 8, and 9. The sum of these elements is $n^2(n^2 + 1)/2$ or 45. There are 3 rows. Each row has the same sum, r. The sum of all the rows, $3*r$, equals $n2^2(n^2 + 1)/2$ or 45. Since each row has the same sum, the sum of each row is $45/3$ or 15. The sum of each column or main diagonal must equal the sum of each row. Therefore, all rows, columns, and main diagonals have a sum of 15.

You can check this for yourself, using the $n = 3$ magic square:

	8	3	4	**=15**
	1	5	9	**=15**
	6	7	2	**=15**
=15	**=15**	**=15**	**=15**	**=15**

What is known?

- For a magic square of size n, you have n^2 numbers, starting at 1 and going consecutively to n^2.

What is unknown?

- The number in each position in the square.

What are the constraints?

- Each number must be different from all the others.
- The sum of each row must equal $n(n^2 + 1)/2$.
- The sum of each column must equal $n(n^2 + 1)/2$.
- The sum of the first main diagonal (from upper left-hand corner to lower right-hand corner) must equal $n(n^2 + 1)/2$.
- The sum of the second main diagonal (from lower left-hand corner to upper right-hand corner) must equal $n(n^2 + 1)/2$.

Model

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

Step 2 Open the example file

Open the example file `YourSolverHome/examples/src/tutorial/magicsq_partial.cpp` in your development environment. In this exercise, you use arrays of constrained integer variables and therefore you use the include file `<ilsolver/ilosolverint.h>`. This example accepts command line arguments, but is set to default to a magic square of size 5. Some of the code for adding constraints to the model is provided for you, as is the code for printing out the solution.

As you know by now, the first step in converting your natural language description of the problem into code using Concert Technology classes is to create an environment and a model. This code is provided for you:

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

Concert Technology gives you the means to represent the unknown information in this problem—the number in each position in the square—as an array of constrained integer

variables. After you create an environment and a model, you declare the variables as an instance of `IloIntVarArray` with n^2 elements, ranging from 1 to n^2. These decision variables will contain the solution to the problem, once it is solved.

Step 3 Declare the decision variables

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

```
    IloIntVarArray square(env, n*n, 1, n*n);
```

This array will give you the number in each position in a magic square of size n. The element `[0]` in the array is the number in the first position in the magic square (row 0, column 0)—remember that Solver arrays start at `[0]`. The element `[1]` in the array is the number in the second position in the magic square (row 0, column 1), and so on.

You use two nested "for" loops to create a matrix to represent the magic square. In this matrix, each element of the array `squares` is situated in a particular row, column, or main diagonal. You will use loops like this to create the constraints on the sums of rows, columns, and main diagonals.

Examine the following matrix:

```
for (i = 0; i < n; i++){
  for (j = 0; j < n; j ++)
    square[n*i+j];
}
```

In this matrix, the variable `i` represents the row number. The variable `j` represents the column number. For a magic square of size $n=3$, the following matrix is created:

	`j = 0` column 0	`j = 1` column 1	`j = 2` column 2
`i = 0` row 0	*3*0 + 0 = 0* `square[0]`	*3*0 + 1 = 1* `square[1]`	*3*0 + 2 = 2* `square[2]`
`i = 1` row 1	*3*1 + 0 = 3* `square[3]`	*3*1 + 1 = 4* `square[4]`	*3*1 + 2 = 5* `square[5]`
`i = 2` row 2	*3*2 + 0 = 6* `square[6]`	*3*2 + 1 = 7* `square[7]`	*3*2 + 2 = 8* `square[8]`

You can use these loops to identify the elements that make up each row. Once you have identified these elements, you can place constraints on their sums.

To model the sum of each row you use expressions, which are represented by the class `IloExpr` in Concert Technology. Values may be combined with variables to form

expressions. The first parameter of the constructor is always the environment. Here is a constructor:

```
IloExpr(const IloEnv env);
```

To combine values with variables to form an expression, you can use the operators:

- addition `+`
- subtraction `-`
- multiplication `*`
- division `/`
- self-assigned addition `+=`
- self-assigned subtraction `-=`

You use `IloExpr` and "for" loops to add the constraints on the sum of each row to the model. The constraints on columns and main diagonals, which are declared in a similar fashion, are provided for you in the code.

Step 4 Add the constraints on rows

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

```
for (i = 0; i < n; i++){
  IloExpr exp(env);
  for(j = 0; j < n; j++)
    exp += square[n*i+j];
  model.add(exp == sum);
  exp.end();
}
```

In this code, the variable `i` represents the row and the variable `j` represents the column. You create an expression `exp` for the first row `i = 0`. Using the nested loop with variable `j`, you traverse that row, column by column, and use the self-assigned addition operator `+=` to add the elements of the array `square` that are located in that row. You then add a constraint to the model that the expression `exp` for each row equals `sum`. In the code provided for you, `sum` is declared to equal $n(n^2 + 1)/2$. Then you add the constraints for the other rows.

IloAllDiff

Using predefined constraints, such as `IloAllDiff`, makes modeling simpler.

The single global constraint `IloAllDiff` on n variables is logically equivalent to $n(n+1)/2$ instances of the "not equal" constraint, `!=`, between each pair of variables in the set.

The member function `IloExpr::end` lets Solver know that an `IloExpr` object will no longer be directly used. Solver can still use the object indirectly, for example through a

constraint that is built on this object. Solver will not delete the object until it is no longer used either directly or indirectly.

Now you add the constraint that all the numbers in the square must be different. To do this, you use the Concert Technology predefined constraint `IloAllDiff`. It states that the value of each variable in an array must be different from that of every other variable in that array. The first parameter is the environment. The second parameter is the array of variables. The third parameter is an optional name used for debug and trace purposes. Here is a constructor:

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

Step 5 Add the constraint IloAllDiff

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

```
    model.add(IloAllDiff(env, square));
```

Solve

After you have declared the decision variables and added the constraints to the model, you are ready to search for a solution. In this lesson, you use the predefined goal `IloGenerate` to search for a solution instead of using a default goal. The predefined goal `IloGenerate` allows you to control a choice in the solution search: you choose which constrained variable Solver tries first.

> **Bound variables**
>
> Each decision variable is associated with a set of possible values called the *domain* of the variable. When the domain of a variable contains only one value, the variable is *bound*.

The predefined goal `IloGenerate` takes an environment as the first parameter and an array of constrained variables as the second parameter. When Solver executes `IloGenerate`, this goal gives a value to each variable in that array. The third parameter `IloChooseIntIndex` determines the order in which variables are chosen. The default is `IloChooseFirstUnboundInt`, which will choose the first variable that is not yet bound. Here is a constructor:

```
IloGoal IloGenerate(IloEnv env,
                    IloNumVarArray vars,
                    IloChooseIntIndex = IloChooseFirstUnboundInt);
```

The goal `IloGenerate` uses the following algorithm:

As long as there exist any unknown variables:

- choose one of those variables
- choose a value to assign to this variable
- assign the value to the variable—the variable is now bound

After this search move, Solver performs constraint propagation during search.

The most critical parameter of `IloGenerate` is, without doubt, the choice of variable because that choice affects performance so directly. If you start your search with the "wrong" variable, you will effectively lose a great deal of computation time. Of course, the "right" order for choosing the variables depends very much on the nature of the problem.

You can use the following parameters to choose the order in which variables are bound in the goal `IloGenerate`:

- `IloChooseMinSizeInt` chooses the variable with the smallest domain.
- `IloChooseMaxSizeInt` chooses the variable with the largest domain.
- `IloChooseMinMinInt` chooses the variable with the least minimal bound.
- `IloChooseMaxMinInt` chooses the variable with the greatest minimal bound.
- `IloChooseMinMaxInt` chooses the variable with the least maximal bound.
- `IloChooseMaxMaxInt` chooses the variable with the greatest maximal bound.

Given the variety of problems in the real world, there are no definitive rules for choosing the best strategy. However, propagation in Solver rests on eliminating branches in the search tree. The sooner Solver can remove or *prune* branches of the search tree, the more efficiently it works. The predefined goal `IloGenerate` controls which branches of that tree are examined and in which order.

A good strategy—as good as a strategy can be and still be general—is to choose first the variable with the smallest domain using the parameter `IloChooseMinSizeInt`. This choice usually develops a search tree where pruning a branch will eliminate a great many possibilities, known as the *first-fail principle*. First-fail is so named because it is a good idea to make the search fail as soon as possible, thereby eliminating a whole branch of the search tree without having spent too much time searching it; to that end, first-fail usually chooses the "most difficult" variable first.

You start the solution search by creating an instance of the class `IloSolver` to solve the problem expressed in the model.

Step 6 Create an instance of IloSolver

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

```
IloSolver solver(model);
```

Next you create an instance of the predefined goal `IloGenerate`. The first parameter is the environment and the second parameter is the array of variables `square`. Since you do not know whether any particular strategy makes sense in this case, use `IloChooseMinSizeInt` as the third parameter.

Step 7 Create the goal

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

```
IloGoal goal = IloGenerate(env, square, IloChooseMinSizeInt);
```

You now use the member function `IloSolver::solve`. In this lesson, you use `goal` as a parameter of `IloSolver::solve`. As you have used the parameter `IloChooseMinSizeInt`, Solver would normally first choose the unbound variable with the smallest domain and select a value for this variable. However, all the variables have the same initial domain in this problem. Therefore, Solver chooses the first variable that is not yet bound and selects a value (by default the minimum value in the domain). Solver then propagates the changes resulting from this decision. Solver next chooses the unbound variable with the smallest domain and propagates the effects of this decision, and so on. If a decision leads to a situation where a constraint is not satisfied, it is undone and Solver backtracks to the last decision.

Step 8 Search for a solution

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

```
if (solver.solve(goal))
```

The member function `IloSolver::solve` returns a Boolean value of type `IloBool`. If a solution is found, an `IloTrue` value is returned and the program displays the solution. The member functions and streams `IloAlgorithm::out` and `IloSolver::getValue` display the solution. The member function `IloSolver::printInformation` displays information about the solution 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.

`IloSolver::printInformation` also provides information about the size of the goal stack, memory usage, and so on.

A matrix created by two “for” loops is again used to display the elements of the array `square` in the format of a square.

The following code is provided for you:

```
    {
    for (i = 0; i < n; i++){
      for (j = 0; j < n; j++)
        solver.out() << " " << solver.getValue(square[n*i+j]);
      solver.out() << endl;
    }
    solver.out() << endl;
  }
  else
    solver.out() << "No solution " << endl;
    solver.printInformation();
```

Step 9 Compile and run the program

Compile and run the program. The program finds a solution for a magic square of size $n = 5$. You can use command line arguments to try to find magic squares of other sizes. You should get the following results for a magic square of size $n = 5$, though the information displayed by `IloSolver::printInformation` will vary depending on platform, machine, configuration, and so on.

```
 1 2 13 24 25
 3 23 17 6 16
 20 21 11 8 5
 22 4 14 18 7
 19 15 10 9 12

Number of fails              : 791
Number of choice points      : 799
Number of variables          : 25
Number of constraints        : 13
Reversible stack (bytes)     : 8064
Solver heap (bytes)          : 20124
Solver global heap (bytes)   : 4044
And stack (bytes)            : 4044
Or stack (bytes)             : 4044
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11152
Total memory used (bytes)    : 55516
Elapsed time since creation  : 0.11
```

The complete program is listed in “Complete program” on page 83. You can also view it online in the file `YourSolverHome/examples/src/magicsq.cpp`.

Review exercises

For answers, see "Suggested answers" on page 78.

1. What information does the member function `IloSolver::printInformation` display?
2. How does the predefined goal `IloGenerate` work?
3. Use the magic square code that you just wrote and experiment with changing the order in which variables are bound in the solution search. Note how the number of fails, number of choice points, and elapsed time since creation change as you try the following parameters for the goal `IloGenerate`: the default (`IloChooseFirstUnboundInt`), `IloChooseMinSizeInt`, `IloChooseMaxSizeInt`, `IloChooseMinMinInt`, `IloChooseMaxMinInt`, `IloChooseMinMaxInt`, and `IloChooseMaxMaxInt`.

   To do this, write a line of code for each `IloGenerate` goal, using the default and the six other parameters. Comment out all the goals except for the one you are testing. Then comment out that goal and test another.
4. Change the program to search for a Gnomon magic square, which is a $n = 4$ magic square in which the elements in each quarter (2 x 2 corner) have the same sum.

Suggested answers

Exercise 1

What information does the member function `IloSolver::printInformation` display?

Suggested Answer

The member function `IloSolver::printInformation` displays information about the solution search, including number of variables, number of constraints, elapsed time since creation of search, number of fails, and number of choice points. `IloSolver::printInformation` also provides information about the size of the goal stack, memory usage, and so on.

Exercise 2

How does the predefined goal `IloGenerate` work?

Suggested Answer

As long as there exist any unknown variables:

1. choose one of those variables

2. choose a value to assign to this variable
3. assign the value to the variable

Exercise 3

Use the magic square code that you just wrote and experiment with changing the order in which variables are bound in the solution search. Note how the number of fails, number of choice points, and elapsed time since creation change as you try the following parameters for the goal `IloGenerate`: the default (`IloChooseFirstUnboundInt`), `IloChooseMinSizeInt`, `IloChooseMaxSizeInt`, `IloChooseMinMinInt`, `IloChooseMaxMinInt`, `IloChooseMinMaxInt`, and `IloChooseMaxMaxInt`.

To do this, write a line of code for each `IloGenerate` goal, using the default and the six other parameters. Comment out all the goals except for the one you are testing. Then comment out that goal and test another.

Suggested Answer

Results will vary widely depending on the strategy you choose. In the results provided in the following sections, using the parameter `IloChooseMinSizeInt` solved the problem the most quickly, in less than one tenth of a second. Solving the problem using the parameter `IloChooseMaxSizeInt` took over 16 minutes. These results vividly demonstrate the importance of parameter choice for the goal `IloGenerate`. The results may vary depending on platform, machine, configuration, and so on.

The code that has changed from `magicsq.cpp` follows. You can view the complete program online in the file `YourSolverHome/examples/src/magicsq_ex3.cpp`.

```
IloGoal goal = IloGenerate(env, square);
//IloGoal goal = IloGenerate(env, square, IloChooseMinSizeInt);
//IloGoal goal = IloGenerate(env, square, IloChooseMaxSizeInt);
//IloGoal goal = IloGenerate(env, square, IloChooseMinMinInt);
//IloGoal goal = IloGenerate(env, square, IloChooseMaxMinInt);
//IloGoal goal = IloGenerate(env, square, IloChooseMinMaxInt);
//IloGoal goal = IloGenerate(env, square, IloChooseMaxMaxInt);
```

You should get the following results, though the information displayed by `IloSolver::printInformation` will vary depending on platform, machine, configuration, and so on.

```
//Feasible Solution using IloGoal goal = IloGenerate(env, square);
 1 2 13 24 25
 3 22 19 6 15
 23 16 10 11 5
 21 7 9 20 8
 17 18 14 4 12

Number of fails               : 5486
Number of choice points       : 5497
Number of variables           : 25
```

```
Number of constraints         : 13
Reversible stack (bytes)      : 8064
Solver heap (bytes)           : 24144
Solver global heap (bytes)    : 4044
And stack (bytes)             : 4044
Or stack (bytes)              : 4044
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 11152
Total memory used (bytes)     : 59536
Elapsed time since creation   : 0.371
//Feasible Solution using IloGoal goal = IloGenerate(env, square,
IloChooseMinSizeInt);
 1 2 13 24 25
 3 23 17 6 16
 20 21 11 8 5
 22 4 14 18 7
 19 15 10 9 12

Number of fails               : 791
Number of choice points       : 799
Number of variables           : 25
Number of constraints         : 13
Reversible stack (bytes)      : 8064
Solver heap (bytes)           : 20124
Solver global heap (bytes)    : 4044
And stack (bytes)             : 4044
Or stack (bytes)              : 4044
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 11152
Total memory used (bytes)     : 55516
Elapsed time since creation   : 0.09
//Feasible Solution using IloGoal goal = IloGenerate(env, square,
IloChooseMaxSizeInt);
 1 2 24 21 17
 3 16 12 11 23
 25 18 13 5 4
 22 10 7 20 6
 14 19 9 8 15

Number of fails               : 15650718
Number of choice points       : 15650729
Number of variables           : 25
Number of constraints         : 13
Reversible stack (bytes)      : 12084
Solver heap (bytes)           : 28164
Solver global heap (bytes)    : 4044
And stack (bytes)             : 4044
Or stack (bytes)              : 4044
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 11152
Total memory used (bytes)     : 67576
Elapsed time since creation   : 997.564

//Feasible Solution using IloGoal goal = IloGenerate(env, square,
IloChooseMinMinInt);
 1 2 24 21 17
 3 16 12 11 23
 25 18 13 5 4
```

```
 22 10 7 20 6
 14 19 9 8 15

Number of fails              : 1325537
Number of choice points      : 1325548
Number of variables          : 25
Number of constraints        : 13
Reversible stack (bytes)     : 12084
Solver heap (bytes)          : 24144
Solver global heap (bytes)   : 4044
And stack (bytes)            : 4044
Or stack (bytes)             : 4044
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11152
Total memory used (bytes)    : 63556
Elapsed time since creation  : 97.169
//Feasible Solution using IloGoal goal = IloGenerate(env, square,
IloChooseMaxMinInt);
 1 2 13 24 25
 3 23 17 6 16
 20 21 11 8 5
 22 4 14 18 7
 19 15 10 9 12

Number of fails              : 849
Number of choice points      : 859
Number of variables          : 25
Number of constraints        : 13
Reversible stack (bytes)     : 8064
Solver heap (bytes)          : 20124
Solver global heap (bytes)   : 4044
And stack (bytes)            : 4044
Or stack (bytes)             : 4044
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11152
Total memory used (bytes)    : 55516
Elapsed time since creation  : 0.1
//Feasible Solution using IloGoal goal = IloGenerate(env, square,
IloChooseMinMaxInt);
 1 2 13 24 25
 3 23 19 4 16
 22 15 10 12 6
 21 8 9 20 7
 18 17 14 5 11

Number of fails              : 612
Number of choice points      : 623
Number of variables          : 25
Number of constraints        : 13
Reversible stack (bytes)     : 8064
Solver heap (bytes)          : 16104
Solver global heap (bytes)   : 4044
And stack (bytes)            : 4044
Or stack (bytes)             : 4044
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11152
Total memory used (bytes)    : 51496
Elapsed time since creation  : 0.09
```

```
//Feasible Solution using IloGoal goal = IloGenerate(env, square,
IloChooseMaxMaxInt);
 1 2 13 24 25
 3 22 19 6 15
 23 16 10 11 5
 21 7 9 20 8
 17 18 14 4 12

Number of fails              : 9020
Number of choice points      : 9029
Number of variables          : 25
Number of constraints        : 13
Reversible stack (bytes)     : 12084
Solver heap (bytes)          : 24144
Solver global heap (bytes)   : 4044
And stack (bytes)            : 4044
Or stack (bytes)             : 4044
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11152
Total memory used (bytes)    : 63556
Elapsed time since creation  : 0.601
```

Exercise 4

Change the program to search for a Gnomon magic square, which is a $n = 4$ magic square in which the elements in each quarter (2 x 2 corner) have the same sum.

Suggested Answer

Describe

- You describe the problem in the same way as you described the magic square problem, except that there are additional constraints: the sum of each quarter (2 x 2 corner) must equal $n(n^2 + 1)/2$.

Model

- You model the problem in the same way as you modeled the magic square problem, except that you add four additional constraints to the model. You use `IloExpr` and “for” loops to add the constraints on the sum of each quarter to the model.

Solve

- You solve the problem in the same way as you solved the magic square problem.

The code that has changed from `magicsq.cpp` follows. You can view the complete program online in the file `YourSolverHome/examples/src/magicsq_ex4.cpp`.

The size of the magic square is set to 4:

```
int main() {
  IloEnv env;
  try {
    IloModel model(env);
    IloInt n = 4;
```

You should add the following additional constraints:

```
    // Constraint on upper left quarter
    IloExpr exp3(env);
    for (i = 0; i < 2; i++)
      for(j = 0; j < 2; j++)
        exp3 += square[i+n*j];
        model.add(exp3 == sum);
        exp3.end();
    // Constraint on upper right quarter
    IloExpr exp4(env);
    for (i = 2; i < n; i++)
      for(j = 0; j < 2; j++)
        exp4 += square[i+n*j];
        model.add(exp4 == sum);
        exp4.end();
    // Constraint on lower left quarter
    IloExpr exp5(env);
    for (i = 0; i < 2; i++)
      for(j = 2; j < n; j++)
        exp5 += square[i+n*j];
        model.add(exp5 == sum);
        exp5.end();
    // Constraint on lower right quarter
    IloExpr exp6(env);
    for (i = 2; i < n; i++)
      for(j = 2; j < n; j++)
        exp6 += square[i+n*j];
        model.add(exp6 == sum);
        exp6.end();
```

You should obtain the following result:

```
Feasible Solution
 1 4 13 16
 14 15 2 3
 8 5 12 9
 11 10 7 6
```

Complete program

The complete magic squares program follows. You can also view it online in the file `YourSolverHome/examples/src/magicsq.cpp`.

```
#include <ilsolver/ilosolverint.h>
ILOSTLBEGIN

int main(int argc, char* argv[]){
  IloEnv env;
  try {
    IloModel model(env);
    IloInt n = (argc > 1) ? atoi(argv[1]) : 5;
    IloInt sum = n*(n*n+1)/2;
    IloInt i, j;
    IloIntVarArray square(env, n*n, 1, n*n);
    model.add(IloAllDiff(env, square));
    // Constraints on rows
    for (i = 0; i < n; i++){
      IloExpr exp(env);
      for(j = 0; j < n; j++)
        exp += square[n*i+j];
      model.add(exp == sum);
      exp.end();
    }
    // Constraints on columns
    for (i = 0; i < n; i++){
      IloExpr exp(env);
      for(j = 0; j < n; j++)
        exp += square[n*j+i];
      model.add(exp == sum);
      exp.end();
    }
    // Constraint on 1st diagonal
    IloExpr exp1(env);
    for (i = 0; i < n; i++)
      exp1 += square[n*i+i];
    model.add(exp1 == sum);
    exp1.end();
    // Constraint on 2nd diagonal
    IloExpr exp2(env);
    for (i = 0; i < n; i++)
      exp2 += square[n*i+n-i-1];
    model.add(exp2 == sum);
    exp2.end();
    IloSolver solver(model);
    IloGoal goal = IloGenerate(env, square, IloChooseMinSizeInt);
    if (solver.solve(goal))
      {
      for (i = 0; i < n; i++){
        for (j = 0; j < n; j++)
          solver.out() << " " << solver.getValue(square[n*i+j]);
        solver.out() << endl;
      }
      solver.out() << endl;
    }
    else
      solver.out() << "No solution " << endl;
      solver.printInformation();
  }
  catch (IloException& ex) {
    cout << "Error: " << ex << endl;
  }
```

```
  env.end();
  return 0;
}
```

Results

You should get the following results, though the information displayed by `IloSolver::printInformation` will vary depending on platform, machine, configuration, and so on.

```
 1 2 13 24 25
 3 23 17 6 16
 20 21 11 8 5
 22 4 14 18 7
 19 15 10 9 12

Number of fails               : 791
Number of choice points       : 799
Number of variables           : 25
Number of constraints         : 13
Reversible stack (bytes)      : 8064
Solver heap (bytes)           : 20124
Solver global heap (bytes)    : 4044
And stack (bytes)             : 4044
Or stack (bytes)              : 4044
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 11152
Total memory used (bytes)     : 55516
Elapsed time since creation   : 0.11
```

C H A P T E R

5

Using Objectives: Map Coloring with Minimum Colors

In this lesson, you will learn how to:

- use objectives
- relax constraints
- use metaconstraints
- use enumerated variables
- use `IloAnyVar`, `IloObjective`, and `IloMaximize`

Describe

You will model and solve a map coloring problem similar to the one that you worked on in Chapter 2, *Modeling and Solving a Simple Problem: Map Coloring*. As you remember, that problem involves choosing colors for the countries on a map in such a way that at most four colors (blue, white, yellow, green) are used, and no neighboring countries are the same color. In this lesson, you will find a solution for a map coloring problem with six countries: Belgium, Denmark, France, Germany, Luxembourg, and the Netherlands. However, in this lesson only three colors (blue, white, and yellow) can be used to color the map.

In this problem, three colors are not going to be enough to color the map in such a way that no neighboring countries are the same color. So, what is the best approach for dealing with this situation? One option is to select what countries will share a color. If you look at the map, you could guess that if you removed one country, for example Luxembourg, from the set of constraints that no neighboring countries are the same color, you might be able to find a solution using just three colors. This is known as *relaxing constraints*.

Relaxing constraints

Relaxing constraints means to remove some of the constraints from the model of an insolvable problem in order to find a solution. The constraints that remain must still be satisfied.

Objective

An *objective* is an expression which can be maximized or minimized. The relative importance of different constraints can be expressed in an objective.

However, you might want to try to have Luxembourg be a different color from some its neighbors, even if it cannot be a different color from all its neighbors. Suppose that you would most like to have Germany and Luxembourg be different colors if possible, then Belgium and Luxembourg if possible, and then France and Luxembourg if possible. You can represent these wishes as an *objective*. The relative importance of different constraints can be expressed in an objective. You can also use an objective to select the most appropriate solution when a problem has more than one solution—if there is more than one set of values for the variables that satisfy the constraints of the problem.

Figure 5.1 *Map for coloring*

Step 1 Describe the problem

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

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

Discussion

What is known?

- You have six countries on a map.
- You have three colors.

What are the decision variables?

- The unknown is the connection between the country and the color. In other words, there are six variables; each variable represents the color of a specific country on the map.

What are the constraints?

- Each constraint is that a country cannot be the same color as its neighbors, except for Luxembourg.

What are the relaxed constraints?

- Luxembourg must be a different color than its neighbors: Germany, Belgium, and France.

What is the objective?

- Try to avoid having Luxembourg share a color with its neighbors in the following order of priority: Germany, Belgium, and France.

Model

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

Step 2 Open the example file

Open the example file `YourSolverHome/examples/src/tutorial/colormin_partial.cpp` in your development environment.

In this exercise, you use constrained enumerated variables (more on this later) and therefore you use the include file `<ilsolver/ilosolverany.h>`.

The first step in converting your natural language description of the problem into code using Concert Technology classes is to create an environment and a model. Since you already know how to do this, it is provided for you in the exercise code:

```
int main(){
  IloEnv env;
  try {
    IloModel model(env);
```

After you create an environment and a model, you declare the variables. In Chapter 2, *Modeling and Solving a Simple Problem: Map Coloring*, you used constrained integer variables, instances of the class `IloIntVar`, to represent the colors of the different countries. Each variable had a possible integer value of `0` to `3`, representing the colors blue (`0`), white (`1`), yellow (`2`), and green (`3`). In this lesson, you will use constrained enumerated variables to represent the unknown information—the color of the country.

Constrained enumerated variables represent constrained variables whose values are objects. There is no restriction on the type of the values for constrained enumerated variables because these variables work with only the addresses of their values. This class of variable

may not seem necessary to you for simple problems, but as your application grows bigger, using object encoding instead of integer encoding can prove quite practical. If you use objects directly as values, you can simplify the representation of your problem, making the model and implementation more intuitive. In this way, object encoding leads to programs that are more legible and more efficient since they are more straightforward.

IloAny

`IloAny` is a type definition that represents objects in a model handled by Concert Technology enumerated variables. A pointer to any object, whether a predefined Concert Technology type or an instance of a C++ class, is implicitly converted to `IloAny`.

Concert Technology uses instances of the class `IloAnyVar` to represent constrained enumerated variables. The first parameter of the constructor is the environment.The second parameter is an array of `IloAny` objects. These represent the possible values for the variable represented by the instance of `IloAnyVar`. The last parameter is an optional name used for debug and trace purposes. Here is a constructor:

```
IloAnyVar(const IloEnv env,
          const IloAnyArray array,
          const char* name = 0);
```

Before creating the constrained enumerated variables to represent the colors of the different countries, you create an instance of `IloAnyArray` to represent the colors. The first parameter is the environment and the second parameter is the size of the array. The other parameters are the `IloAny` objects in the array. Here is a constructor:

```
IloAnyArray(const IloEnv env, IloInt n,
            const IloAny p0,
            const IloAny p1, . . .);
```

Step 3 Create the array of values

Add the following code after the comment `//Create the array of values`

```
IloAnyArray Colors(env, 3, blue, white, yellow);
```

The array `Colors` contains three objects of type `IloAny`, `blue`, `white`, and `yellow`, representing the three colors available for coloring the map.

Next, you declare the constrained enumerated variables, one for each country. Each variable represents the unknown information—the color of the country. The possible values of the variable are those given in the array `Colors`. These decision variables will contain the solution to the problem, once it is solved.

Step 4 Create the decision variables

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

```
IloAnyVar Belgium(env, Colors), Denmark(env, Colors),
          France(env, Colors), Germany(env, Colors),
          Luxembourg(env, Colors), Netherlands(env, Colors);
```

Now, you add the set of constraints that all neighboring countries, with the exception of Luxembourg, cannot be the same color. Since you have already done this in Chapter 2, *Modeling and Solving a Simple Problem: Map Coloring*, this code is provided for you:

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

Then you create an objective. This objective will express the relative importance, in order of priority, of the following relaxed constraints:

- Germany and Luxembourg are different colors if possible.
- Belgium and Luxembourg are different colors if possible.
- France and Luxembourg are different colors if possible.

An objective is simply an expression which can be maximized or minimized. In this case, you are going to maximize the expression. You need to find a way to model the three relaxed constraints and assign a relative importance to them. You already know how to model the three relaxed constraints using the `!=` operator:

```
Luxembourg != Germany
Luxembourg != Belgium
Luxembourg != France
```

Now instead of adding them directly to the model, you combine them, creating a *metaconstraint*. You can do this because constraints have value. The value of a constraint is `IloTrue` if the constraint is satisfied or `IloFalse` if the constraint is not satisfied. Another way of looking at the value of a constraint is as a Boolean value that can be represented as a binary integer. In this representation, constraints that are true have an integer value of 1 and constraints that are false have an integer value of 0. This convention makes it possible to use constraints themselves in an expression. You can use arithmetic or logical operators with

constraints, combine them with each other, or maximize an expression that contains constraints.

> **Metaconstraints**
>
> *Metaconstraints* are created by combining constraints. Constraints can be combined by using arithmetic and logical operators. Constraints can be imposed on other constraints.
>
> For more information on metaconstraints, see "More on metaconstraints" on page 96.

To create the objective, you use the function `IloMaximize`, which creates an instance of the class `IloObjective`. The function `IloMaximize` takes the environment as its first parameter and an expression as its second parameter. The last parameter is an optional name used for debug and trace purposes. Here is a constructor for `IloMaximize`:

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

Step 5 Create the objective

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

```
    IloObjective obj = IloMaximize(env, 9043 * (Luxembourg != Germany)
                                      +  568 * (Luxembourg != Belgium)
                                      +  257 * (Luxembourg != France));
```

When you create the objective, you use the multiplication operator to link each of the relaxed constraints with a coefficient that represents the relative importance of that relaxed constraint. If the constraint is true, the coefficient will be multiplied by 1. If the constraint is false, the coefficient will be multiplied by 0. You create a metaconstraint by combining the three relaxed constraints in an expression. When Solver searches for a solution, it tries to maximize the value of this expression. Therefore, the relaxed constraints which have a larger coefficient (because they are considered more important) will be more likely to form part of the solution.

For example, if Solver found a solution where France and Luxembourg were different colors, then the relaxed constraint `Luxembourg != France` would be true and would be represented by the integer value of 1. If, in the same solution, Luxembourg was the same color as both Germany and Belgium, then the relaxed constraints `Luxembourg != Germany` and `Luxembourg != Belgium` would both be false and would be represented by the integer value of 0. The expression would then be:

$(257 * 1) + (9043 * 0) + (568 * 0) = 257$

You use the member function `IloModel::add` to add the objective to the model. You must explicitly add an objective to the model or Solver will not be able to use it in the search for a solution.

Step 6 Add the objective to the model

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

```
model.add(obj);
```

Solve

Solving the problem consists of finding a value for each variable that satisfies the constraints and maximizes the objective containing the relaxed constraints.

Step 7 Create an instance of IloSolver

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

```
IloSolver solver(model);
```

You now use the member function `IloSolver::solve`, which solves a problem by using a default goal to launch the search.

Step 8 Search for a solution

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

```
if (solver.solve())
```

You use the member functions and streams `IloAlgorithm::getStatus`, `IloSolver::getValue`, and the function `print` to display the solution. These are already provided for you in the exercise code.

```
  {
  solver.out() << solver.getStatus() << " Solution" << endl;
  solver.out() << "Objective = " << solver.getValue(obj) << endl;
  print(solver, "Belgium:     ", Belgium);
  print(solver, "Denmark:     ", Denmark);
  print(solver, "France:      ", France);
  print(solver, "Germany:     ", Germany);
  print(solver, "Netherlands: ", Netherlands);
  print(solver, "Luxembourg:  ", Luxembourg);
}
```

Step 9 Compile and run the program

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

```
Optimal Solution
Objective = 9611
Belgium:     white
Denmark:     blue
France:      blue
Germany:     yellow
Netherlands: blue
Luxembourg:  blue
```

As you can see, only three colors are used. The “unrelaxed” constraints are all satisfied. The objective is maximized. The relaxed constraint `Luxembourg != Germany` is satisfied. This adds 9043 to the expression. The relaxed constraint `Luxembourg != Belgium` is also satisfied. This adds 568 to the expression. This gives a value of 9611 to the objective. The only way that the value of this objective expression could be greater would be if the relaxed constraint `Luxembourg != France` were also satisfied. However, this is not a possible solution since it would mean that “unrelaxed” constraints would not be satisfied.

The complete program is listed in “Complete program” on page 99. You can also view it online in the file `YourSolverHome/examples/src/colormin.cpp`.

More on metaconstraints

Metaconstraints are created by combining constraints or placing constraints on other constraints. Metaconstraints are based on the idea that constraints have value. Solver handles constraints not only as if they have a Boolean value, such as true or false, but effectively as if the value is 0 or 1. This allows you to combine constraints into expressions or impose constraints on constraints, as you have done in this lesson.

Constraints can be combined using arithmetic operators. You can also use the following logical operators to combine constraints:

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

Review exercises

For answers, see “Suggested answers” on page 97.

1. What is relaxing constraints?
2. What is an objective?
3. Add an objective to the changing money problem you worked on in Chapter 3, *Using Arrays and Basic Search: Changing Money*. Assume that you want to pay for an item that costs 1.23 euros. You have a mixture of coins of the following types: 1 euro cent, 10 euro cents, 20 euro cents, and 1 euro. To make the problem more interesting, assume that you have only 5 coins of 1 euro cent. You prefer to pay for the item using as many of your lower denomination coins as possible. Create an objective that uses the following coefficients to represent the importance of using each type of coin: 1000 (1 euro cent), 800 (10 euro cents), 300 (20 euro cents), and 10 (1 euro). Change the code so that you only search for one maximized solution.

4. Add a different objective to the original changing money problem you worked on in Chapter 3, *Using Arrays and Basic Search: Changing Money*. This time, instead of trying to pay for the item using as many of your lower denomination coins as possible as in Exercise 3, try to use as few coins as possible. Assume that you want to pay for an item that costs 1.23 euros. You have a mixture of coins of the following types: 1 euro cent, 10 euro cents, 20 euro cents, and 1 euro. To make the problem more interesting, assume that you have only 5 coins of 1 euro cent. Create an objective that minimizes the number of coins using the function `IloMinimize`. Change the code so that you only search for the minimized solution.

Suggested answers

Exercise 1

What is relaxing constraints?

Suggested Answer

Relaxing constraints means to remove some of the constraints from the model of an insolvable problem in order to find a solution. The constraints that remain must still be satisfied.

Exercise 2

What is an objective?

Suggested Answer

An objective is an expression that can be maximized or minimized. The relative importance of different constraints can be expressed in an objective.

An objective can also be used to select the most appropriate solution when a problem has more than one solution.

Exercise 3

Add an objective to the changing money problem you worked on in Chapter 3, *Using Arrays and Basic Search: Changing Money*. Assume that you want to pay for an item that costs 1.23 euros. You have a mixture of coins of the following types: 1 euro cent, 10 euro cents, 20 euro cents, and 1 euro. To make the problem more interesting, assume that you have only 5 coins of 1 euro cent. You prefer to pay for the item using as many of your lower denomination coins as possible. Create an objective that uses the following coefficients to represent the importance of using each type of coin: 1000 (1 euro cent), 800 (10 euro cents), 300 (20 euro cents), and 10 (1 euro). Change the code so that you only search for one maximized solution.

Suggested Answer

The code that has changed from `money.cpp` follows. You can view the complete program online in the file `YourSolverHome/examples/src/colormin_ex3.cpp`.

You add the objective:

```
IloObjective obj = IloMaximize(env, (1000 * (coins[0]))
                                 +  (800 * (coins[1]))
                                 +  (300 * (coins[2]))
                                 +   (10 * (coins[3])));
model.add(obj);
```

You change the following code to display the solution:

```
if (solver.solve())
  {
  solver.out() << solver.getStatus() << " Solution" << endl;
  solver.out() << "Objective = " << solver.getValue(obj) << endl;
  for (IloInt i = 0; i < nCoins; i++)
    solver.out() << solver.getValue(coins[i]) << " ";
    solver.out() << endl;
}
else
  solver.out() << "No Solution" << endl;
```

You should get the following results:

```
[1, 10, 20, 100]
Optimal Solution
Objective = 12600
3 12 0 0
```

Exercise 4

Add a different objective to the original changing money problem you worked on in Chapter 3, *Using Arrays and Basic Search: Changing Money*. This time, instead of trying to pay for the item using as many of your lower denomination coins as possible as in Exercise 3, try to use as few coins as possible. Assume that you want to pay for an item that costs 1.23 euros. You have a mixture of coins of the following types: 1 euro cent, 10 euro cents, 20 euro cents, and 1 euro. To make the problem more interesting, assume that you have only 5 coins of 1 euro cent. Create an objective that minimizes the number of coins using the function `IloMinimize`. Change the code so that you only search for one minimized solution.

Suggested Answer

The code that has changed from `money.cpp` follows. You can view the complete program online in the file `YourSolverHome/examples/src/colormin_ex4.cpp`.

You add the following objective:

```
    IloObjective obj = IloMinimize(env, coins[0] + coins[1] + coins[2] +
coins[3]);
```

You change the following code to display the solution:

```
      if (solver.solve())
        {
        for (IloInt i = 0; i < nCoins; i++)
        solver.out() << solver.getValue(coins[i]) << " ";
        solver.out() << endl;
      }
    else
    solver.out() << "No Solution" << endl;
```

You should get the following results:

```
[1, 10, 20, 100]
3 0 1 1
```

Complete program

The complete map coloring with minimum colors program follows. You can also view it online in the file `YourSolverHome/examples/src/colormin.cpp`.

```
#include <ilsolver/ilosolverany.h>
ILOSTLBEGIN

char blue[]="blue";
char white[]="white";
char yellow[]="yellow";

void print(IloSolver solver, char* name, IloAnyVar color) {
        solver.out() << name << (char*)solver.getAnyValue(color) << endl;
}


int main(){
  IloEnv env;
  try {
    IloModel model(env);
    IloAnyArray Colors(env, 3, blue, white, yellow);
    IloAnyVar Belgium(env, Colors), Denmark(env, Colors),
              France(env, Colors), Germany(env, Colors),
              Luxembourg(env, Colors), Netherlands(env, Colors);
    model.add(France != Belgium);
    model.add(France != Germany);
    model.add(Belgium != Netherlands);
    model.add(Germany != Netherlands);
    model.add(Germany != Denmark);
```

```
    model.add(Germany != Belgium);
    IloObjective obj = IloMaximize(env, 9043 * (Luxembourg != Germany)
                                  +  568 * (Luxembourg != Belgium)
                                  +  257 * (Luxembourg != France));
    model.add(obj);
    IloSolver solver(model);
    if (solver.solve())
      {
      solver.out() << solver.getStatus() << " Solution" << endl;
      solver.out() << "Objective = " << solver.getValue(obj) << endl;
      print(solver, "Belgium:     ", Belgium);
      print(solver, "Denmark:     ", Denmark);
      print(solver, "France:      ", France);
      print(solver, "Germany:     ", Germany);
      print(solver, "Netherlands: ", Netherlands);
      print(solver, "Luxembourg:  ", Luxembourg);
    }
    else
      solver.out() << "No Solution" << endl;
  }
  catch (IloException& ex) {
    cerr << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Results

```
Optimal Solution
Objective = 9611
Belgium:     white
Denmark:     blue
France:      blue
Germany:     yellow
Netherlands: blue
Luxembourg:  blue
```

Part II

More on Modeling

This part consists of the following lessons:

- Chapter 6, *Using the Distribute Constraint: Car Sequencing*
- Chapter 7, *Using Set Variables: Crew Scheduling*
- Chapter 8, *Combining Constraints: Bin Packing*
- Chapter 9, *Using Table Constraints: Scheduling Teams*
- Chapter 10, *Reducing Symmetry: Configuring Racks*
- Chapter 11, *Using Constrained Floating-Point Variables: Modeling Equations*

CHAPTER

6

Using the Distribute Constraint: Car Sequencing

In this lesson, you will learn how to:

- use the predefined constraint `IloDistribute`
- model a simplified version of an industrial optimization problem, car sequencing

Describe

A car sequencing problem is an example of an industrial optimization problem. Cars are available with many different options. Car color, leather seats, wood interior, sunroof, type of motor, Global Positioning System (GPS), air conditioning, and so on are all examples of options. Cars can be configured with different sets of options. For example, a manufacturer will produce a certain number of blue cars with leather seats, wood interior, and air conditioning, a certain number of green cars with sunroof and GPS, and so on for each model of car.

The problem in car sequencing is to determine the order in which cars of each configuration should be assembled. The assembly line must produce a certain number of cars of each configuration. Not every car can be fitted with every option, due to technical limitations. For technical and financial reasons, the order in which cars are sent along the assembly line, or the *sequence* of the cars, is important.

The sunroof option is one example of how technical limitations restrict the number of cars with a certain option and restrict the order of the cars on the assembly line. As the car moves along the assembly line, a machine and crew move along with it, installing the sunroof. The machine can only move a certain distance along the assembly line. In the time it takes the machine to move along the assembly line with the car, the sunroof is installed. After the sunroof is installed, the machine moves back to its original position on the assembly line. At this stage, however, several cars have already passed the starting point of the sunroof machine and crew. Therefore, only the next car to pass the starting point can have the sunroof option. Those cars that have already passed cannot have it installed. It is important that cars that require a sunroof are in the right place at the right time or in the correct sequence. If an option becomes so popular that it needs to be installed on every car, then the assembly line is modified to reflect this.

The car paint color option is another example of how technical limitations restrict the number of cars with a certain option and restrict the order of the cars on the assembly line. The machine that paints the car must be serviced after every four cars. The paint is emptied, the tubes and pipes are cleaned, and a new color of paint is added. Only four cars can be painted a certain color before the machine is serviced. Then four cars can be painted another color before the machine is again serviced. The order of these cars is important to make sure that the color of the car is matched with the other options in a particular configuration.

In this lesson, you will a model a simplified car sequencing problem to learn how to use the distribute constraint. The distribute constraint is a constraint that is much more complex than the constraints you have worked with so far. For a more complex model of a car sequencing problem, see Chapter 18, *Writing a Goal: Car Sequencing*.

Assume that you have eight cars to paint in three available colors: green, yellow, and blue. Due to technical limitations on the assembly line, no more than three cars can be painted green, exactly three cars must be painted yellow, and no more than two cars can be painted blue. Due to limitations on the order of the cars, the first car off the assembly line cannot be painted green.

Step 1 Describe the problem

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

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

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

Discussion

What is known?

- there are eight cars that must be painted in order on the assembly line
- there are three paint colors: green, yellow, and blue

What is unknown?

- the color each car is painted

What are the constraints?

- no more than three cars can be painted green
- exactly three cars must be painted yellow
- no more than two cars can be painted blue
- the first car cannot be painted green

Model

Once you have written a description of your problem, you can use Concert Technology classes to model it. Before declaring the decision variables and adding constraints, you must create an environment and a model. Once you create a model of your problem, you can use Solver classes to search for a solution.

In the lessons so far, you created decision variables that contained the solution to the problem, once it was solved. In this part, you will use decision variables in more sophisticated ways. You will still declare decision variables that will contain the solution to the problem, but you will also declare other decision variables that are used in finding the solution.

Step 2 Open the example file

Open the example file `YourSolverHome/examples/src/tutorial/carseq_basic_partial.cpp` in your development environment. In this exercise, you use arrays of constrained integer variables and therefore you use the include file `<ilsolver/ilosolverint.h>`. The number of cars, `nbCars`, is set to 8 in this problem and the number of colors, `nbColors`, is set to 3.

As you know by now, the first step in converting your natural language description of the problem into code using Concert Technology classes is to create an environment and a model. This code is provided for you:

```
int main() {
  IloEnv env;
  try {
    IloModel model(env);
```

Concert Technology gives you the means to represent the unknown information in this problem—the color each car is painted—as an array of constrained integer variables. You use `IloIntVarArray` to create an array. Each element in the array will represent the color of a car, and so each variable can take a value from `0` to `nbColors-1`. The array of decision variables `cars` will contain the solution to the problem, once it is solved.

Step 3 Declare the decision variables

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

```
IloIntVarArray cars(env, nbCars, 0, nbColors-1);
```

To represent the constraint that the first car off the assembly line cannot be painted green, you could declare an array of values to represent the three colors and then use the arithmetic constraint `!=` to represent that the first car off the assembly line cannot be painted green.

But how are you going to represent the other constraints? No more than three cars can be painted green, exactly three cars must be painted yellow, and no more than two cars can be painted blue. It is also important to know the order of the cars on the assembly line, since the first car off the assembly line cannot be painted green. You need a constraint that allows you to count the number of variables (the cars) that take a given value (the color of paint) in an array, while respecting the order of the variables.

Concert Technology provides a type of global constraint that does just that. The global constraint `IloDistribute` allows you to count the number of variables that take a given value.

Global constraints

Global constraints make it possible to express complicated relations between variables, relations that would require an exponential number of arithmetic constraints, or relations that could not be expressed at all in arithmetic terms.

The predefined constraint `IloAllDiff` is a global constraint that can be substituted for a very large number of arithmetic inequality constraints (`!=`).

It would not be possible to express the predefined constraint `IloDistribute` using only arithmetic constraints.

`IloDistribute` takes the following parameters: the environment, a counting array, an array of values, and an array of constrained variables. The last parameter is an optional name used for debug and trace purposes. Here is a constructor:

```
IloDistribute (const IloEnv env,
               const IloIntVarArray cards,
               const IloNumArray values,
               const IloNumVarArray vars,
               const char* name = 0);
```

The constraint `IloDistribute` takes three arrays: `cards`, `values`, and `vars`. The constrained variables in the array `cards` are equal to the number of occurrences in the array `vars` of the values in the array `values`. More precisely, for each `i`, `cards[i]` is equal to the number of occurrences of `values[i]` in the array `vars`.

What do you need to do to use the constraint `IloDistribute` in your car sequencing model? You need to create the three arrays (`cards`, `values`, and `vars`) before you can add the constraint `IloDistribute` to the model:

- You have already declared an array of decision variables `cars`. This array corresponds to the array `vars` in the constructor of `IloDistribute`.
- Next, you use `IloIntArray` to declare an array of values `colors`. In this simple example, the array `colors` represents the three colors the cars can be painted. This array corresponds to the array `values` in the constructor.
- Finally, you need to create the `cards` array of cardinal numbers used to count the variables. The `cards` array must be the same size as the `colors` array. The `cards` array is used to provide the information to link each color with the number of times it can be used in configuring the cars. First, you create `cards`, an instance of `IloIntVarArray` of the same size as the array `colors`. Then, you use the variables in the array `cards` to represent the number of times each value in the array `colors` can be used as a value in the array of variables `cars`.

Step 4 Declare the array of values

Add the following code after the comment `//Declare the array of values`

```
IloIntArray colors(env, 3, 0, 1, 2);
```

Now you declare the counting array `cards`. For each `i`, `cards[i]` is equal to the number of occurrences of `colors[i]` in the array `cars`. For example, the color green is represented by the element `[0]` in the array `colors`. Since no more than three cars can be painted green, you set the element `[0]` in the array of counting variables `cards` to be equal to a constrained integer variable with a domain of 0 to 3.

Step 5 Declare the array of decision variables cards

Add the following code after the comment
`//Declare the array of decision variables cards`

```
IloIntVarArray cards(env, nbColors);
cards[0] = IloIntVar(env, 0, 3);
cards[1] = IloIntVar(env, 3, 3);
cards[2] = IloIntVar(env, 0, 2);
```

Though `cards` is an array of decision variables, it will not contain the solution to the problem once it is solved. This array of decision variables is instead used in creating the `IloDistribute` constraint.

Now that you have declared the three arrays that are used in `IloDistribute`, you can add the constraint to the model.

Step 6 Add the distribute constraint

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

```
model.add(IloDistribute(env, cards, colors, cars));
```

You also add the arithmetic constraint `!=` to represent that the first car off the assembly line cannot be painted green. Explicitly, this constraint means that the value Solver finds for the first element in the array of constrained integer variables `cars` cannot equal the value for the color green.

Step 7 Add the constraint that the first car cannot be green

Add the following code after the comment
`//Add the constraint that the first car cannot be green`

```
model.add(cars[0] != 0);
```

Solve

After you have declared the decision variables and added the constraints to the model, you are ready to search for a solution. In this lesson, you use the predefined goal `IloGenerate` to search for a solution, as you did in Chapter 4, *Searching with Predefined Goals: Magic Square*. The predefined goal `IloGenerate` allows you to control a choice in the solution search: you choose which constrained variable Solver tries to bind first.

You start the solution search by creating an instance of the class `IloSolver` to solve the problem expressed in the model.

Step 8 Create an instance of IloSolver

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

```
IloSolver solver(model);
```

You now use the member function `IloSolver::solve`. In this lesson, you use the predefined goal `IloGenerate` as a parameter of `IloSolver::solve`. The predefined goal itself takes the following parameters: the environment, the array of variables `cars`, and `IloChooseMinSizeInt`. From the array of variables `cars`, Solver will first choose the unbound variable with the smallest domain and select a value from that domain. Solver will propagate the effects of this decision and repeat the process, moving to the variable with the next smallest domain. If a decision leads to a situation where a constraint is not satisfied, it is undone and Solver backtracks to the last decision.

Step 9 Search for a solution

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

```
if (solver.solve(IloGenerate(env, cars, IloChooseMinSizeInt)))
```

The member functions and streams `IloAlgorithm::out` and `IloSolver::getValue` are used to display the solution. When you print the solution, you associate each value with the name of a color using the array `Names[]`, which is provided for you in the exercise code. As in Chapter 4, *Searching with Predefined Goals: Magic Square*, the member function `IloSolver::printInformation` displays information about the solution search, including number of variables, number of constraints, elapsed time since creation of search, number of fails, and number of choice points.

The following code is provided for you:

```
  {
  solver.out() << solver.getStatus() << " Solution" << endl;
  for (i = 0; i < nbCars; i++) {
    solver.out() << "Car " << i+1 <<  " color:     "
                 << Names[(IloInt)solver.getValue(cars[i])]
                 << endl;
  }
}
else
  solver.out() << "No solution " << endl;
  solver.printInformation();
```

Step 10 Compile and run the program

Compile and run the program. You should get the following results, though the information displayed by `IloSolver::printInformation` will vary depending on platform, machine, configuration, and so on.

```
Feasible Solution
Car 1 color:     yellow
Car 2 color:     green
Car 3 color:     green
Car 4 color:     green
Car 5 color:     yellow
Car 6 color:     yellow
Car 7 color:     blue
Car 8 color:     blue
Number of fails              : 0
Number of choice points      : 6
Number of variables          : 12
Number of constraints        : 2
Reversible stack (bytes)     : 4044
Solver heap (bytes)          : 8064
Solver global heap (bytes)   : 4044
And stack (bytes)            : 4044
Or stack (bytes)             : 4044
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11152
Total memory used (bytes)    : 39436
Elapsed time since creation  : 0.17
```

As you can see, three cars are painted green; three cars are painted yellow; and two cars are painted blue. The first car off the assembly line is not painted green. This is obviously a very simple example with a very straightforward result. You will model and solve a more complex car sequencing problem, involving different options and configurations, in Chapter 18, *Writing a Goal: Car Sequencing*.

The complete program is listed in "Complete program" on page 115. You can also view it online in the file `YourSolverHome/examples/src/carseq_basic.cpp`.

Review exercises

For answers, see "Suggested answers" on page 111.

1. What are global constraints?

2. What parameters does the global constraint `IloDistribute` take?

3. Modify the car sequencing problem. Assume that you have 20 cars to paint in 4 available colors: green, yellow, blue, and white. Due to technical limitations on the assembly line, no more than 6 cars can be painted green, exactly 5 cars must be painted yellow, no more than 7 cars can be painted blue, and no less than 5 cars and no more than 12 cars can be painted white. The first car off the assembly line cannot be painted green.

4. Use the `IloDistribute` constraint to solve for a magic sequence. A magic sequence is a sequence of $n + 1$ values ($x0, x1, \ldots, xn$) such that 0 appears in the sequence $x0$ times, 1 appears $x1$ times, …, and n appears in the sequence xn times. The numbers in the magic sequence have a sum equal to $n + 1$.

   For example, for $n = 3$, the following sequence is a solution: $(1, 2, 1, 0)$. That is, 0 is present once, 1 is present twice, 2 is present once, and 3 is not present (present zero times, as it were).

   Describe, model, and solve this problem. Write the code so that you can solve magic sequences of different sizes, but find a solution for a magic sequence of size $n = 10$. This is a problem with a high level of modeling difficulty.

Suggested answers

Exercise 1

What are global constraints?

Suggested Answer

Global constraints make it possible to express complicated relations between variables, relations that would require an exponential number of arithmetic constraints, or relations that could not be expressed at all in arithmetic terms.

Exercise 2

What parameters does the global constraint `IloDistribute` take?

Suggested Answer

`IloDistribute` takes the following parameters: the environment, a counting array, an array of values, and an array of constrained variables.

Exercise 3

Modify the car sequencing problem. Assume that you have 20 cars to paint in 4 available colors: green, yellow, blue, and white. Due to technical limitations on the assembly line, no more than 6 cars can be painted green, exactly 5 cars must be painted yellow, no more than 7

cars can be painted blue, and no less than 5 cars and no more than 12 cars can be painted white. The first car off the assembly line cannot be painted green.

Suggested Answer

The code that has changed from `carseq_basic.cpp` follows. You can view the complete program online in the file `YourSolverHome/examples/src/carseq_basic_ex3.cpp`.

Add the fourth color to the array `Names`:

```
const char* Names[] = {"green", "yellow", "blue", "white"};
```

The following code changes the number of cars and colors:

```
    const IloInt nbCars    = 20;
    const IloInt nbColors = 4;
```

You modify the distribute constraint:

```
    IloIntArray colors(env, 4, 0, 1, 2, 3);
    IloIntVarArray cards(env, nbColors);
    cards[0] = IloIntVar(env, 0, 6);
    cards[1] = IloIntVar(env, 5, 5);
    cards[2] = IloIntVar(env, 0, 7);
    cards[3] = IloIntVar(env, 5, 12);
```

You should get the following results, though the information displayed by `IloSolver::printInformation` will vary depending on platform, machine, configuration, and so on.

```
Feasible Solution
Car 1 color:     yellow
Car 2 color:     green
Car 3 color:     green
Car 4 color:     green
Car 5 color:     green
Car 6 color:     green
Car 7 color:     green
Car 8 color:     yellow
Car 9 color:     yellow
Car 10 color:     yellow
Car 11 color:     yellow
Car 12 color:     blue
Car 13 color:     blue
Car 14 color:     blue
Car 15 color:     blue
Car 16 color:     white
Car 17 color:     white
Car 18 color:     white
Car 19 color:     white
Car 20 color:     white
Number of fails              : 0
Number of choice points      : 15
```

```
Number of variables          : 25
Number of constraints        : 2
Reversible stack (bytes)     : 4044
Solver heap (bytes)          : 12084
Solver global heap (bytes)   : 4044
And stack (bytes)            : 4044
Or stack (bytes)             : 4044
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11152
Total memory used (bytes)    : 43456
Elapsed time since creation  : 0.16
```

Exercise 4

Use the `IloDistribute` constraint to solve for a magic sequence. A magic sequence is a sequence of $n + 1$ values ($x0$, $x1$, …, xn) such that 0 appears in the sequence $x0$ times, 1 appears $x1$ times, …, and n appears in the sequence xn times. The numbers in the magic sequence have a sum equal to $n + 1$.

For example, for $n = 3$, the following sequence is a solution: $(1, 2, 1, 0)$. That is, 0 is present once, 1 is present twice, 2 is present once, and 3 is not present (present zero times, as it were).

Describe, model, and solve this problem. Write the code so that you can solve magic sequences of different sizes, but find a solution for a magic sequence of size $n = 10$. This is a problem with a high level of modeling difficulty.

Suggested Answer

Describe

- The known information is the size of the magic sequence. The unknown information is the sequence of numbers. The first constraint is that the numbers in the magic sequence have a sum equal to $n + 1$. The second constraint is that the number 0 appears in the sequence $x0$ times, 1 appears $x1$ times, and so on.

Model

- To represent the numbers in the sequence, you use an array of integer variables `vars`. You also create an array of coefficients `coeffs`. This array contains the numbers 0, 1, 2, 3, and so on.

  To model the first constraint that the numbers in the magic sequence have a sum equal to $n + 1$, you use `IloScalProd` to constrain that the sum of each variable times its coefficient is equal to $n + 1$. For example, using the magic sequence given in the exercise for $n = 3$, which is $(1, 2, 1, 0)$, this equation is:

  $(x0)(0) + (x1)(1) + (x2)(2) + (x3)(3) = n + 1$, which is equivalent to

  $(1)(0) + (2)(1) + (1)(2) + (0)(3) = 4$

To model the second constraint that the number 0 appears in the sequence $x0$ times, 1 appears $x1$ times, and so on, you use the `IloDistribute` constraint. As you remember, `IloDistribute` takes three arrays as parameters: an array of variables used to count, an array of values, and an array of variables. In this exercise, the array of values is `coeffs` and the array of variables is `vars`, the sequence of numbers. You need to use an array to count how many times each value in `coeffs` can be used as a value in the array of variables `vars`. In this problem, the array of variables `vars` is also used as the counting array. This contributes to the difficulty level of modeling this problem. For example, using the magic sequence given in the exercise for $n = 3$, which is $(1, 2, 1, 0)$, the `IloDistribute` constraint would work as follows:

counting array `vars`: `[1, 2, 1, 0]`

array of values `coeffs`: `[0, 1, 2, 3]`

array of variables `vars`: `[1, 2, 1, 0]`

As you can see, the first element in the array of values `coeffs`, 0, appears in the array of variables `vars` 1 time. This is because the first element in the counting array, which is also `vars`, is 1.

Solve

- You solve the problem in the same way as you solved the car sequencing example, using the predefined goal `IloGenerate`.

The complete program follows. You can also view it online in the file `YourSolverHome/examples/src/carseq_basic_ex4.cpp`.

```
# include <ilsolver/ilosolverint.h>

ILOSTLBEGIN

int main(int argc, char** argv){
  IloEnv env;
  try {
    IloModel model(env);
    IloInt n = (argc > 1) ? atoi(argv[1]) : 10;
    IloInt i;
    IloIntVarArray vars(env, n + 1, 0, n + 1);
    IloIntArray coeffs(env, n + 1);
    for (i = 0; i < n + 1; i++)
      coeffs[i] = i;
    model.add(IloDistribute(env, vars, coeffs, vars));
    // redundant constraint
    model.add(IloScalProd(vars,coeffs) == n + 1);
    IloSolver solver(model);
    if (solver.solve(IloGenerate(env, vars, IloChooseMinSizeInt))) {
      for (i = 0; i < n + 1; i++)
        solver.out() << solver.getValue(vars[i]) << " ";
      solver.out() << endl;
    }
    else
      solver.out() << "No solution" << endl;
```

```
    solver.printInformation();
  }
  catch (IloException& ex) {
    cerr << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Results

You should get the following results, though the information displayed by `IloSolver::printInformation` will vary depending on platform, machine, configuration, and so on.

```
7 2 1 0 0 0 0 1 0 0 0
Number of fails              : 5
Number of choice points      : 6
Number of variables          : 23
Number of constraints        : 14
Reversible stack (bytes)     : 4044
Solver heap (bytes)          : 12084
Solver global heap (bytes)   : 8088
And stack (bytes)            : 4044
Or stack (bytes)             : 4044
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11144
Total memory used (bytes)    : 47492
Running time since creation  : 0.01
```

Complete program

The complete simplified car sequencing program follows. You can also view it online in the file `YourSolverHome/examples/src/carseq_basic.cpp`.

```
#include <ilsolver/ilosolverint.h>

ILOSTLBEGIN
const char* Names[] = {"green", "yellow", "blue"};
int main() {
  IloEnv env;
  try {
    IloModel model(env);
    const IloInt nbCars    = 8;
    const IloInt nbColors = 3;
    IloInt i;
    IloIntVarArray cars(env, nbCars, 0, nbColors-1);
    IloIntArray colors(env, 3, 0, 1, 2);
    IloIntVarArray cards(env, nbColors);
    cards[0] = IloIntVar(env, 0, 3);
    cards[1] = IloIntVar(env, 3, 3);
    cards[2] = IloIntVar(env, 0, 2);
```

```
    model.add(IloDistribute(env, cards, colors, cars));
    model.add(cars[0] != 0);
    IloSolver solver(model);
    if (solver.solve(IloGenerate(env, cars, IloChooseMinSizeInt)))
      {
      solver.out() << solver.getStatus() << " Solution" << endl;
      for (i = 0; i < nbCars; i++) {
        solver.out() << "Car " << i+1 <<  " color:     "
                     << Names[(IloInt)solver.getValue(cars[i])]
                     << endl;
      }
    }
    else
      solver.out() << "No solution " << endl;
      solver.printInformation();
  }
  catch (IloException& ex) {
    cout << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Results

You should get the following results, though the information displayed by `IloSolver::printInformation` will vary depending on platform, machine, configuration, and so on.

```
Feasible Solution
Car 1 color:     yellow
Car 2 color:     green
Car 3 color:     green
Car 4 color:     green
Car 5 color:     yellow
Car 6 color:     yellow
Car 7 color:     blue
Car 8 color:     blue
Number of fails               : 0
Number of choice points       : 6
Number of variables           : 12
Number of constraints         : 2
Reversible stack (bytes)      : 4044
Solver heap (bytes)           : 8064
Solver global heap (bytes)    : 4044
And stack (bytes)             : 4044
Or stack (bytes)              : 4044
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 11152
Total memory used (bytes)     : 39436
Elapsed time since creation   : 0.17
```

CHAPTER

7

Using Set Variables: Crew Scheduling

In this lesson, you will learn how to:

- use constrained sets of variables
- design a model for extensibility
- use `IloNumSetVar`, `IloNumSetVarArray`, `IloNullIntersect`, `IloCard`, and `IloEqIntersection`

Describe

An airline has to assign flight attendants to flights. There are 10 flights a day. Each flight requires a certain number of flight attendants, which varies with the size of the plane. The company employs 20 flight attendants, all of whom speak English.

There are certain constraints on the composition of each crew. Each crew must contain a certain minimum number of senior and junior staff and certain minimum numbers of attendants fluent in the following languages besides English: French, German, and Spanish. Flight attendants must rest at least two flights between assignments.

You will design the model so that it can be extended; the number of flights and the constraints on the composition of the crew can be easily adjusted.

Here is a table with the 10 flights and their staffing requirements:

Table 7.1 *Flight requirements*

Flight Number	Size of Crew	Minimum Number of Senior Staff	Minimum Number of Junior Staff	Minimum Number of French Staff	Minimum Number of German Staff	Minimum Number of Spanish Staff
#1	4	1	1	1	1	1
#2	5	1	1	1	1	1
#3	5	1	1	1	1	1
#4	6	2	2	2	1	2
#5	7	3	2	2	1	2
#6	4	1	1	1	1	1
#7	5	1	1	1	1	1
#8	6	1	1	1	1	1
#9	6	2	2	1	1	1
#10	7	3	3	1	1	1

Here is a table listing staff and their attributes:

Table 7.2 *Staff Attributes*

Employee Name	Experience Level	Language Skills other than English
Bill	Senior	French, Spanish
Bob	Senior	
Carol	Senior	
Carolyn	Senior	
Cathy	Junior	German
David	Junior	
Ed	Senior	
Fred	Senior	Spanish
Heather	Junior	Spanish
Inez	Junior	French, Spanish

Table 7.2 Staff Attributes (Continued)

Employee Name	Experience Level	Language Skills other than English
Janet	Senior	
Jean	Junior	French
Jeremy	Junior	German
Joe	Junior	Spanish
Juliet	Junior	French, German
Marilyn	Senior	Spanish
Mario	Senior	German, Spanish
Ron	Junior	French
Tom	Junior	German
Tracy	Senior	

Step 1 Describe the problem

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

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

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

Discussion

What is the known information in this problem?

- there are 20 flight attendants
- there are 10 flights
- flight attendant experience level and language skills

What are the decision variables in this problem?

- which attendants fly on which flights

What are the constraints in this problem?

- flight attendants must rest at least two flights between assignments
- each flight requires a certain crew size
- each flight has minimum staffing requirements based on experience level
- each flight has minimum staffing requirements based on language skills

Model

Once you have written a description of your problem, you can use Concert Technology classes to model it. In this lesson, you declare two functions before the `main` function. One is used to display the solution and the other is used to add constraints to the model.

As in Chapter 6, *Using the Distribute Constraint: Car Sequencing*, you use decision variables in more sophisticated ways in this lesson. You still declare decision variables that will contain the solution to the problem, but you will also declare other decision variables that are used in finding the solution.

Step 2 Open the example file

Open the example file `YourSolverHome/examples/src/tutorial/crews_partial.cpp` in your development environment. In the lessons in *Part I*, you completed most of the exercise code yourself. In this lesson and later lessons, the examples are more complex and more of the code is provided for you. You follow the steps to learn how to do specific tasks within the example code.

In this exercise, you use constrained set variables (more on that later) and therefore you use the include file `<ilsolver/ilosolverset.h>`.

The first step in converting your natural language description of the problem into code using Concert Technology classes is to create an environment and a model in the `main` function. The variable `i` is used in a “for” loop. Since you already know how to do this, it is provided for you in the exercise code:

```
int main() {
  IloEnv env;
  try {
    IloModel model(env);
    IloInt i;
```

Next, you create the variables to represent the number of crews, `nCrews`, and the number of attributes required for each flight, `nAttributes`. The attributes include crew experience level and language skills. In this lesson, the number of crews is initialized to 10 and the number of attributes is initialized to 5 (the minimum number of crew on a flight with a senior experience level, a junior experience level, French skills, German skills, and Spanish

skills.) You can easily increase the number of crews or the number of attributes by changing the values of these variables. This code is provided for you:

```
const IloInt nAttributes = 5;
const IloInt nCrews = 10;
```

Then, you represent the data of the program.

You represent the names of the flight attendants using a C++ enumeration to associate meaningful names with the integer values. This code is provided for you:

```
enum Employee {Bill, Bob, Carol, Carolyn, Cathy,
               David, Ed, Fred, Heather, Inez,
               Janet, Jean, Jeremy, Joe, Juliet,
               Marilyn, Mario, Ron, Tom, Tracy};
```

You create an array `Staff` to represent all employees. This code is provided for you:

```
IloNumArray Staff(env, 20,
                  Bill, Bob, Carol, Carolyn, Cathy,
                  David, Ed, Fred, Heather, Inez,
                  Janet, Jean, Jeremy, Joe, Juliet,
                  Marilyn, Mario, Ron, Tom, Tracy);
```

Now, you create arrays to represent all employees by experience level and by language skills. If you wanted to extend the example by adding another crew attribute to the model (for example, Italian language skills), you would create another array representing the flight attendants who possess that attribute. This code is provided for you:

```
IloNumArray SeniorArray(env, 10,
                        Bill, Bob, Carol, Carolyn, Ed,
                        Fred, Janet, Marilyn, Mario, Tracy);
IloNumArray JuniorArray(env, 10,
                        Cathy, David, Heather, Inez, Jean,
                        Jeremy, Joe, Juliet, Ron, Tom);
IloNumArray FrenchArray(env, 5, Bill, Inez, Jean, Juliet, Ron);
IloNumArray GermanArray(env, 5, Cathy, Jeremy, Juliet, Mario, Tom);
IloNumArray SpanishArray(env, 7, Bill, Fred, Heather, Inez, Joe,
                         Marilyn, Mario);
```

After representing the known information of the problem, you declare the variables. The unknown information is the crew composition for each flight (which crew members work on which flights). Concert Technology gives you the means to represent the unknowns in this

problem—the crew composition for each flight—as constrained set variables. A constrained set variable is a variable that represents a set of numeric values.

> **Using constrained sets of variables**
>
> A *set* is a group of elements. These elements have no specific order. You can use constrained sets of variables to model variables as sets. This is useful for avoiding symmetric solutions and reducing the complexity of a model.
>
> For example, you could represent each crew member on each flight as a separate variable. But this would give you many solutions where the order of the crew members was different, but the actual crew members were the same.
>
> A better model—one that avoids symmetric solutions—uses a single constrained set of variables to represent each crew. This model reduces symmetry since it represents a crew as a set.

A constrained set variable is defined in terms of two other sets: its required elements and its possible elements. Its required elements are those that must be in the set. Its possible elements are those that may be in the set.

Constrained set variables are represented by the class `IloNumSetVar` in Concert Technology. The first parameter of the constructor is the environment. The second parameter is the possible elements of the set and the third parameter is the required elements of the set. You have the option of creating a name for the variable, but you will not do so in this exercise. If only one array is given as a parameter, this array represents the set of possible elements. The last parameter is an optional name used for debug and trace purposes. Here is a constructor:

```
IloNumSetVar(const IloEnv env,
             const IloNumArray possible,
             const IloNumArray required,
             const char* name=0);
```

You use an instance of `IloNumSetVarArray`, an array of constrained set numeric variables, to represent the crew composition for all flights. These decision variables will contain the solution to the problem, once it is solved.

Step 3 Declare the decision variables

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

```
IloNumSetVarArray crews(env, nCrews);
for(i = 0; i < nCrews; i++)
  crews[i] = IloNumSetVar(env, Staff);
```

Each flight crew is represented by an instance of `IloNumSetVar` and the crews for each of the 10 flights are represented by an instance of `IloNumSetVarArray`. The array is created

by using a “for” loop to create one instance of `IloNumSetVar` to represent each crew. Since only one array is given as a parameter, this array represents the set of possible elements. The set of possible elements for each crew is the array `Staff`, or all the flight attendants.

Now that you have added the data and declared the variables, you can add the constraints to the model.

First, you add the constraint that the flight attendants must rest at least two flights between assignments. To do this, you use the predefined constraint `IloNullIntersect`. This constraint takes two constrained set variables as arguments as well as the environment. It forces the set `var1` to have no elements in common with the set `var2`. In other words, the intersection of `var1` with `var2` will be empty when this constraint is satisfied. Here is a constructor:

```
IloConstraint IloNullIntersect(const IloEnv env,
                               const IloNumSetVar var1,
                               const IloNumSetVar var2);
```

You can use `IloNullIntersect` to make sure that flight attendants rest at least two flights between assignments.

Step 4 Add the constraint on resting between flights

Add the following code after the comment
`//Add the constraint on resting between flights`

```
    for(i = 0 ; i < nCrews-1 ; i++) {
      model.add(IloNullIntersect(env, crews[i], crews[i+1]));
      if (i < nCrews-2)
        model.add(IloNullIntersect(env, crews[i], crews[i+2]));
    }
```

You first add the constraint that the intersection of the first crew and the second crew should be empty. In other words, these two crews will not share any staff. Then you add the constraint that the intersection of the first crew and the third crew should be empty. This is repeated in a “for” loop for all the flights, up to and including `nCrews-1` or Flight # 9. An “if” clause is introduced because there is no need to add the second part of the constraint after Flight # 7.

The rest of the constraints, concerning crew size and requirements for experience levels and language skills, are added to the model by calling the function `TeamConstraints`, which is declared at the start of the program. This function takes six parameters:

- ◆ The first argument is `model`.
- ◆ The second argument is an element of the array `crews`, or the crew members selected for each flight.

- The third argument is an element of the array `crewSize`.
- The fourth argument is an element of the array of arrays `crewRequirements`, or the requirements for each flight in terms of crew experience levels and language skills.
- The fifth argument is `attributeSets`, the data reflecting the experience levels and language skills of each flight attendant in sets of decision variables.
- The sixth parameter is `dataArrays`, the data reflecting experience levels and language skills of each flight attendant in numeric arrays.

Step 5 Declare the TeamConstraints function

Add the following code after the comment `//Declare the TeamConstraints function`

```
void TeamConstraints(IloModel model,
                     const IloNumSetVar crews,
                     const IloNum crewSize,
                     const IloNumArray crewRequirements,
                     const IloNumSetVarArray attributeSets,
                     const IloArray<IloNumArray> dataArrays)  {
  IloEnv env = model.getEnv();
```

Parameters of TeamConstraints

You have already declared the model and the array of set variables `crews`. The other parameters are arrays or elements of arrays that are also declared in the `main` function: `crewSize`, `dataArrays`, `crewRequirements`, and `attributeSets`.

crewSize

The array `crewSize` is declared in the `main` function. You declare an instance of the class `IloNumArray` with `nCrews` elements. The elements of this array are the crew size requirements for each flight. The following code is provided for you:

```
    IloNumArray crewSize(env, nCrews, 4, 5, 5, 6, 7, 4, 5, 6, 6, 7);
```

crewRequirements

The array `crewRequirements` is declared in the `main` function. You use the class `IloArray` to create an array of `IloNumArray` arrays. The array contains 10 arrays, one for each flight. Each element in the array `crewRequirements` is an array representing the crew requirements for each flight. Each array contains `nAttributes` elements, which is set to 5 in this example. The first element is the number of required senior staff members and is initialized to 1. The second element is the number of required junior staff members and is initialized to 1. The third element is the number of required French speakers. The fourth element is the number of required German speakers. The fifth element is the number of

required Spanish speakers. These three elements are all initialized to 1 because each flight must contain at least one of the following: senior staff member, junior staff member, French speaker, German speaker, and Spanish speaker. A "for" loop is used to create the 10 arrays. The following code is provided:

```
IloArray<IloNumArray> crewRequirements(env, nCrews);
for (i =0; i < nCrews; i++)
  crewRequirements[i] = IloNumArray(env, nAttributes,
                                   1, 1, 1, 1, 1);
```

You use a C++ enumeration to associate meaningful names with the elements of the `crewRequirements` array. This code is provided:

```
enum CrewRequirementsElements {SeniorSet, JuniorSet,
                               FrenchSet, GermanSet, SpanishSet};
```

You adjust the `crewRequirements` arrays for those requirements that are not the same as the initialized defaults. For example, Flight #4 has a minimum senior staff requirement of 2, not 1. Therefore, you initialize the `seniorSet` element of `crewRequirements[3]` to 2. The code is provided for you:

```
  crewRequirements[3][SeniorSet] = 2;
  crewRequirements[3][JuniorSet] = 2;
  crewRequirements[3][FrenchSet] = 2;
  crewRequirements[3][SpanishSet] = 2;
  crewRequirements[4][SeniorSet] = 3;
  crewRequirements[4][JuniorSet] = 2;
  crewRequirements[4][FrenchSet] = 2;
  crewRequirements[4][SpanishSet] = 2;
  crewRequirements[8][SeniorSet] = 2;
  crewRequirements[8][JuniorSet] = 2;
  crewRequirements[9][SeniorSet] = 3;
  crewRequirements[9][JuniorSet] = 3;
```

attributeSets

The array of set variables `attributeSets` is declared in the `main` function. This is an array of sets that represents the experience levels and language skills of each flight attendant.

First, you create the attribute sets using the class `IloNumSetVar`. For example, you create the set of variables `Senior` with three parameters. The first parameter is the environment. The second parameter, the set of possible values, is the array `SeniorArray`. The elements of `SeniorArray` are all flight attendants with senior experience levels. The third parameter, the set of required values, is the same array `SeniorArray`. In effect, this means that `SeniorArray` and `Senior` contain the same exact elements. However, both the numeric array, `SeniorArray`, and the constrained set of variables, `Senior`, will be used in creating

the constraints. Again, if you wanted to add another attribute, you would create a set representing the employees with that attribute. The following code is provided for you:

```
IloNumSetVar Senior(env, SeniorArray, SeniorArray);
IloNumSetVar Junior(env, JuniorArray, JuniorArray);
IloNumSetVar French(env, FrenchArray, FrenchArray);
IloNumSetVar German(env, GermanArray, GermanArray);
IloNumSetVar Spanish(env, SpanishArray, SpanishArray);
```

Next, you create the array of constrained set variables `attributeSets`. This array has `nAttributes` elements. The elements are the five sets of decision variables you just created. They will be used in creating constraints. The following code is provided:

```
IloNumSetVarArray attributeSets(env, nAttributes, Senior, Junior,
                                French, German, Spanish);
```

dataArrays

The array `dataArrays` is declared in the `main` function. You use the class `IloArray` to create an array of `IloNumArray` arrays. The elements of this array are the five arrays used to represent the experience levels and language skills of the employees. This is the same information as that represented by `attributeSets`. Both forms of this data, numeric arrays and set variables, are required to create the constraints. The array contains `NumAttributes` elements, which is set to 5 in this example. This array represents all the attributes. If you wanted to add another attribute, for example another language skill, you would have to create an array representing the flight attendants who had that skill and then add it to the array of arrays `dataArrays`. The following code is provided for you:

```
IloArray<IloNumArray> dataArrays(env, nAttributes);
dataArrays[0] = SeniorArray;
dataArrays[1] = JuniorArray;
dataArrays[2] = FrenchArray;
dataArrays[3] = GermanArray;
dataArrays[4] = SpanishArray;
```

Now that you know what the parameters represent, you can add constraints to the `TeamConstraints` function. The first constraint states that the size of the crew should be equal to the crew size requirement for the flight. To do this, you use the function `IloCard`. This function constrains the number of elements in a set variable. You use `IloCard` to constrain the size of the set `crews` to equal `CrewSize`.

Step 6 Add the constraint on crew size

Add the following code after the comment `//Add the constraint on crew size`

```
model.add(IloCard(crews) == crewSize);
```

Next, you state the constraint that each crew must contain staff members that meet the requirements for experience level and language skills. To do this, you use the predefined constraint `IloEqIntersection`. This constraint takes four parameters, the environment and three sets of variables. The constraint forces the intersection of the sets `var2` and `var3` to be precisely the elements of the set `intersection`. Here is a constructor:

```
IloConstraint IloEqIntersection (const IloEnv env,
                                 const IloNumSetVar intersection,
                                 const IloNumSetVar var2,
                                 const IloNumSetVar var3);
```

The code works in a loop. The first time you take the intersection of two sets: the set of crew members for Flight #1 and the set `Senior`, representing all employees with a senior experience level. The intersection of these sets represents the Flight #1 crew members who have a senior experience level. You then add a constraint that the number of crew members in this intersection (Flight #1 crew members with a senior experience level) is greater than or equal to the crew requirements for senior staff on Flight #1. You then loop back and take the intersection of the set of crew members for Flight #1 and the set `Junior`, representing all employees with a junior experience level. The intersection of these sets represents the Flight #1 crew members who have a junior experience level. You add the constraint that the number of crew members in this intersection (Flight #1 crew members with a junior experience level) is greater than or equal to the crew requirements for junior staff on Flight #1. You continue looping until you have done this for all the crew requirements: senior experience level, junior experience level, French, German, and Spanish.

Step 7 Add the constraints on crew requirements

Add the following code after the comment
`//Add the constraints on crew requirements`

```
  IloInt size = crewRequirements.getSize();
  for (IloInt i = 0; i < size; i++) {
    IloNumSetVar intersection(env, dataArrays[i]);
    model.add(IloEqIntersection(env, intersection, crews, attributeSets[i]));
    model.add(IloCard(intersection) >= crewRequirements[i]);
    }
}
```

The number of times you loop is determined by `crewRequirements.getSize`. If, for example, you call the `TeamConstraints` function for Flight #1, you would have the following scenario. The set of variables `intersection` has a possible set of the element of `dataArrays`. In the first loop, this element is `dataArrays[0]` or `SeniorArray`, an `IloNumArray` which consists of all flight attendants who have a senior experience level. You use `IloEqIntersection` to add the constraint that `intersection` is a set that contains elements that are in both the set `crews` (the crew for Flight # 1) and the set of the specified element of `attributeSets`. In the first loop, this element is

attributeSets[0] or Senior, an IloNumSetVar which consists of all flight attendants who have a senior experience level. Then you add a constraint using IloCard, to state that the number of members in the set intersection must be greater than or equal to the specified element of crewRequirements. In the first loop, this element is crewRequirements[0] or, in the case of Flight #1, 1 (a minimum of 1 flight attendant with a senior experience level must be in the crew).

After you create the TeamConstraints function, you call it in the main function. A "for" loop is used to call the function 10 times, once for each flight.

Step 8 Call the TeamConstraints function

Add the following code after the comment //Call the TeamConstraints function

```
for(i=0; i< nCrews;i++)
  TeamConstraints(model, crews[i], crewSize[i],
                  crewRequirements[i], attributeSets, dataArrays);
```

Solve

Now you create an instance of the class IloSolver to solve the problem expressed in the model.

Step 9 Create an instance of IloSolver

Add the following code after the comment //Create an instance of IloSolver

```
IloSolver solver(model);
```

Then you search for a solution using the predefined goal IloGenerate. You use a "for" loop to display the solution using the function Print.

Step 10 Search for a solution

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

```
if (solver.solve(IloGenerate(env, crews))){
  solver.out() << "Solution" << endl;
  for (IloInt j=0 ; j< crews.getSize() ; j++) {
    solver.out() << "Crew #" << (j+1) << ": " ;
    Print(solver, crews[j], Names);
  }
}
else
  solver.out() << "No solution" << endl;
solver.printInformation();
```

You use the `Print` function to display the solution. You use an iterator to display the values in each set in the array of set variables `crews`. This code is provided for you:

```
void Print(IloSolver solver, IloNumSetVar crews, char** Names) {
  for(IloNumSet::Iterator iter(solver.getIntSetValue(crews));iter.ok();++iter){
    solver.out() << Names[(IloInt)*iter] << " ";
  }
  solver.out() << endl;
}
```

Step 11 Compile and run the program

Compile and run the program. You should get the following results, though the information displayed by `IloSolver::printInformation` will vary depending on platform, machine, configuration, and so on:

```
Solution
Crew #1: Bill Bob Carol Cathy
Crew #2: Carolyn David Ed Fred Juliet
Crew #3: Heather Inez Janet Jeremy Joe
Crew #4: Bill Bob Carol Cathy Jean Marilyn
Crew #5: Carolyn David Ed Fred Juliet Mario Ron
Crew #6: Heather Inez Janet Jeremy
Crew #7: Bill Bob Carol Cathy Jean
Crew #8: Carolyn David Ed Fred Joe Juliet
Crew #9: Heather Inez Janet Jeremy Marilyn Mario
Crew #10: Bill Bob Carol Cathy Jean Ron Tom
Number of fails               : 8
Number of choice points       : 57
Number of variables           : 130
Number of constraints         : 232
Reversible stack (bytes)      : 20124
Solver heap (bytes)           : 112584
Solver global heap (bytes)    : 4044
And stack (bytes)             : 4044
Or stack (bytes)              : 4044
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 11152
Total memory used (bytes)     : 160036
Elapsed time since creation   : 0.09
```

As you can see, the crew members assigned to each flight are displayed. The complete program is listed in "Complete program" on page 134. You can also view it online in the file `YourSolverHome/examples/src/crews.cpp`.

Review exercises

For answers, see "Suggested answers" on page 131.

1. What are some of the benefits of using sets of constrained variables?
2. How does the constraint `IloEqIntersection` work?
3. Using the crews example as a starting point, extend the example to deal with a new crew requirement. Each flight crew must have at least one Italian speaker, except for Flight #4, which must have at least two Italian speakers. The following flight attendants speak Italian: Bob, Carol, Janet, Marilyn, and Tracy.

4. Using the original crews example as a starting point, extend the example by adding five more flights with the following crew requirements:

Table 7.3 *Flight requirements*

Flight Number	Size of Crew	Minimum Number of Senior Staff	Minimum Number of Junior Staff	Minimum Number of French Staff	Minimum Number of German Staff	Minimum Number of Spanish Staff
#11	4	1	1	1	1	1
#12	4	1	1	1	1	1
#13	7	2	3	2	2	2
#14	4	1	1	1	1	1
#15	5	2	1	2	1	2

Suggested answers

Exercise 1

What are some of the benefits of using sets of constrained variables?

Suggested Answer

You can use constrained sets of variables to represent certain problems as sets. This is useful for avoiding symmetric solutions and reducing the apparent complexity of a model.

Exercise 2

How does the constraint `IloEqIntersection` work?

Suggested Answer

The constraint `IloEqIntersection` forces the intersection of two sets to be precisely the elements of a third set.

Exercise 3

Using the crews example as a starting point, extend the example to deal with a new crew requirement. Each flight crew must have at least one Italian speaker, except for Flight #4, which must have at least two Italian speakers. The following flight attendants speak Italian: Bob, Carol, Janet, Marilyn, and Tracy.

Suggested Answer

The code that has changed from `crews.cpp` follows. You can view the complete program online in the file `YourSolverHome/examples/src/crews_ex3.cpp`.

You change the number of attributes:

```
const IloInt nAttributes = 6;
```

You create an array of Italian speakers:

```
IloNumArray ItalianArray(env, 5, Bob, Carol, Janet, Marilyn, Tracy);
```

You add an element to `dataArrays`:

```
dataArrays[5] = ItalianArray;
```

You create an `Italian` set of variables:

```
IloNumSetVar Italian(env, ItalianArray, ItalianArray);
```

You add this set of variables to the array of sets of variables and add the requirement that a default flight has one Italian speaker:

```
IloNumSetVarArray attributeSets(env, nAttributes, Senior, Junior,
                                French, German, Spanish, Italian);

enum CrewRequirementsElements {SeniorSet, JuniorSet, FrenchSet,
                               GermanSet, SpanishSet, ItalianSet};

IloArray<IloNumArray> crewRequirements(env, nCrews);
for (i =0; i < nCrews; i++)
  crewRequirements[i] = IloNumArray(env, nAttributes,
                                    1, 1, 1, 1, 1, 1);
```

You add the requirements that Flight #4 have two Italian speakers:

```
crewRequirements[3][ItalianSet] = 2;
```

You should get the following results, though the information displayed by `IloSolver::printInformation` will vary depending on platform, machine, configuration, and so on.

```
Solution
Crew #1: Bill Bob Carol Cathy
Crew #2: Carolyn David Ed Juliet Marilyn
Crew #3: Fred Heather Inez Janet Jeremy
Crew #4: Bill Bob Carol Cathy Jean Joe
Crew #5: Carolyn David Ed Juliet Marilyn Mario Ron
Crew #6: Fred Inez Janet Jeremy
Crew #7: Bill Bob Carol Cathy Heather
Crew #8: Carolyn David Ed Jean Juliet Marilyn
Crew #9: Fred Inez Janet Jeremy Joe Mario
Crew #10: Bill Bob Carol Cathy Heather Ron Tom
Number of fails               : 44
Number of choice points       : 94
Number of variables           : 152
Number of constraints         : 273
Reversible stack (bytes)      : 24144
Solver heap (bytes)           : 128664
Solver global heap (bytes)    : 4044
And stack (bytes)             : 4044
Or stack (bytes)              : 4044
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 11152
Total memory used (bytes)     : 180136
Elapsed time since creation   : 0.07
```

Exercise 4

Using the original crews example as a starting point, extend the example by adding five more flights with the crew requirements as shown in Table 7.3 on page 131.

Suggested Answer

The code that has changed from `crews.cpp` follows. You can view the complete program online in the file `YourSolverHome/examples/src/crews_ex4.cpp`.

You change the number of crews:

```
const IloInt nCrews = 15;
```

You specify the crew size of the additional flights:

```
IloNumArray crewSize(env, nCrews, 4, 5, 5, 6, 7, 4, 5, 6,
                                  6, 7, 4, 4, 7, 4, 5);
```

You specify specific crew requirements for the additional flights:

```
crewRequirements[12][SeniorSet] = 2;
crewRequirements[12][JuniorSet] = 3;
crewRequirements[12][FrenchSet] = 2;
crewRequirements[12][GermanSet] = 2;
crewRequirements[12][SpanishSet] = 2;
crewRequirements[14][SeniorSet] = 2;
crewRequirements[14][FrenchSet] = 2;
crewRequirements[14][SpanishSet] = 2;
```

You should get the following results, though the information displayed by `IloSolver::printInformation` will vary depending on platform, machine, configuration, and so on.

```
Solution
Crew #1: Bill Bob Carol Cathy
Crew #2: Carolyn David Ed Fred Juliet
Crew #3: Heather Inez Janet Jeremy Joe
Crew #4: Bill Bob Carol Cathy Jean Marilyn
Crew #5: Carolyn David Ed Fred Juliet Mario Ron
Crew #6: Heather Inez Janet Jeremy
Crew #7: Bill Bob Carol Cathy Jean
Crew #8: Carolyn David Ed Fred Joe Juliet
Crew #9: Heather Inez Janet Jeremy Marilyn Mario
Crew #10: Bill Bob Carol Cathy Jean Ron Tom
Crew #11: Carolyn David Fred Juliet
Crew #12: Ed Heather Inez Jeremy
Crew #13: Bill Bob Carol Cathy Jean Joe Mario
Crew #14: Carolyn David Fred Juliet
Crew #15: Ed Inez Jeremy Marilyn Ron
Number of fails               : 21
Number of choice points       : 89
Number of variables           : 190
Number of constraints         : 347
Reversible stack (bytes)      : 28164
Solver heap (bytes)           : 164844
Solver global heap (bytes)    : 4044
And stack (bytes)             : 4044
Or stack (bytes)              : 4044
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 11152
Total memory used (bytes)     : 220336
Elapsed time since creation   : 0.08
```

Complete program

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

```
#include <ilsolver/ilosolverset.h>
ILOSTLBEGIN
```

```
void TeamConstraints(IloModel model,
                     const IloNumSetVar crews,
                     const IloNum crewSize,
                     const IloNumArray crewRequirements,
                     const IloNumSetVarArray attributeSets,
                     const IloArray<IloNumArray> dataArrays)  {
  IloEnv env = model.getEnv();
  model.add(IloCard(crews) == crewSize);
  IloInt size = crewRequirements.getSize();
  for (IloInt i = 0; i < size; i++) {
    IloNumSetVar intersection(env, dataArrays[i]);
    model.add(IloEqIntersection(env, intersection, crews, attributeSets[i]));
    model.add(IloCard(intersection) >= crewRequirements[i]);
    }
}
void Print(IloSolver solver, IloNumSetVar crews, char** Names) {
  for(IloNumSet::Iterator iter(solver.getIntSetValue(crews));iter.ok();++iter){
    solver.out() << Names[(IloInt)*iter] << " ";
  }
  solver.out() << endl;
}
int main() {
  IloEnv env;
  try {
    IloModel model(env);
    IloInt i;
    char* Names[20];

    for (i=0; i < 20; i++){
      Names[i]=new (env) char[10];
    }
    strcpy(Names[0],"Bill");
    strcpy(Names[1],"Bob");
    strcpy(Names[2],"Carol");
    strcpy(Names[3],"Carolyn");
    strcpy(Names[4],"Cathy");
    strcpy(Names[5],"David");
    strcpy(Names[6],"Ed");
    strcpy(Names[7],"Fred");
    strcpy(Names[8],"Heather");
    strcpy(Names[9],"Inez");
    strcpy(Names[10],"Janet");
    strcpy(Names[11],"Jean");
    strcpy(Names[12],"Jeremy");
    strcpy(Names[13],"Joe");
    strcpy(Names[14],"Juliet");
    strcpy(Names[15],"Marilyn");
    strcpy(Names[16],"Mario");
    strcpy(Names[17],"Ron");
    strcpy(Names[18],"Tom");
    strcpy(Names[19],"Tracy");

    enum Employee {Bill, Bob, Carol, Carolyn, Cathy,
                   David, Ed, Fred, Heather, Inez,
                   Janet, Jean, Jeremy, Joe, Juliet,
                   Marilyn, Mario, Ron, Tom, Tracy};
    IloNumArray Staff(env, 20,
```

```
                  Bill, Bob, Carol, Carolyn, Cathy,
                  David, Ed, Fred, Heather, Inez,
                  Janet, Jean, Jeremy, Joe, Juliet,
                  Marilyn, Mario, Ron, Tom, Tracy);

const IloInt nAttributes = 5;
const IloInt nCrews = 10;
IloNumSetVarArray crews(env, nCrews);
for(i = 0; i < nCrews; i++)
  crews[i] = IloNumSetVar(env, Staff);
IloNumArray SeniorArray(env, 10,
                        Bill, Bob, Carol, Carolyn, Ed,
                        Fred, Janet, Marilyn, Mario, Tracy);
IloNumArray JuniorArray(env, 10,
                        Cathy, David, Heather, Inez, Jean,
                        Jeremy, Joe, Juliet, Ron, Tom);

IloNumArray FrenchArray(env, 5, Bill, Inez, Jean, Juliet, Ron);
IloNumArray GermanArray(env, 5, Cathy, Jeremy, Juliet, Mario, Tom);
IloNumArray SpanishArray(env, 7, Bill, Fred, Heather, Inez, Joe,
                         Marilyn, Mario);

IloArray<IloNumArray> dataArrays(env, nAttributes);
dataArrays[0] = SeniorArray;
dataArrays[1] = JuniorArray;
dataArrays[2] = FrenchArray;
dataArrays[3] = GermanArray;
dataArrays[4] = SpanishArray;

IloNumSetVar Senior(env, SeniorArray, SeniorArray);
IloNumSetVar Junior(env, JuniorArray, JuniorArray);
IloNumSetVar French(env, FrenchArray, FrenchArray);
IloNumSetVar German(env, GermanArray, GermanArray);
IloNumSetVar Spanish(env, SpanishArray, SpanishArray);
IloNumArray crewSize(env, nCrews, 4, 5, 5, 6, 7, 4, 5, 6, 6, 7);
IloNumSetVarArray attributeSets(env, nAttributes, Senior, Junior,
                                French, German, Spanish);

enum CrewRequirementsElements {SeniorSet, JuniorSet,
                               FrenchSet, GermanSet, SpanishSet};

IloArray<IloNumArray> crewRequirements(env, nCrews);
for (i =0; i < nCrews; i++)
  crewRequirements[i] = IloNumArray(env, nAttributes,
                                    1, 1, 1, 1, 1);
  crewRequirements[3][SeniorSet] = 2;
  crewRequirements[3][JuniorSet] = 2;
  crewRequirements[3][FrenchSet] = 2;
  crewRequirements[3][SpanishSet] = 2;
  crewRequirements[4][SeniorSet] = 3;
  crewRequirements[4][JuniorSet] = 2;
  crewRequirements[4][FrenchSet] = 2;
  crewRequirements[4][SpanishSet] = 2;
  crewRequirements[8][SeniorSet] = 2;
  crewRequirements[8][JuniorSet] = 2;
  crewRequirements[9][SeniorSet] = 3;
  crewRequirements[9][JuniorSet] = 3;
for(i=0; i< nCrews;i++)
```

```
      TeamConstraints(model, crews[i], crewSize[i],
                      crewRequirements[i], attributeSets, dataArrays);
    for(i = 0 ; i < nCrews-1 ; i++) {
      model.add(IloNullIntersect(env, crews[i], crews[i+1]));
      if (i < nCrews-2)
        model.add(IloNullIntersect(env, crews[i], crews[i+2]));
    }

    IloSolver solver(model);
    if (solver.solve(IloGenerate(env, crews))){
      solver.out() << "Solution" << endl;
      for (IloInt j=0 ; j< crews.getSize() ; j++) {
        solver.out() << "Crew #" << (j+1) << ": " ;
        Print(solver, crews[j], Names);
      }
    }
    else
      solver.out() << "No solution" << endl;
    solver.printInformation();
  }
  catch (IloException& ex) {
    cerr << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Results

You should get the following results, though the information displayed by `IloSolver::printInformation` will vary depending on platform, machine, configuration, and so on.

```
Solution
Crew #1: Bill Bob Carol Cathy
Crew #2: Carolyn David Ed Fred Juliet
Crew #3: Heather Inez Janet Jeremy Joe
Crew #4: Bill Bob Carol Cathy Jean Marilyn
Crew #5: Carolyn David Ed Fred Juliet Mario Ron
Crew #6: Heather Inez Janet Jeremy
Crew #7: Bill Bob Carol Cathy Jean
Crew #8: Carolyn David Ed Fred Joe Juliet
Crew #9: Heather Inez Janet Jeremy Marilyn Mario
Crew #10: Bill Bob Carol Cathy Jean Ron Tom
Number of fails               : 8
Number of choice points       : 57
Number of variables           : 130
Number of constraints         : 232
Reversible stack (bytes)      : 20124
Solver heap (bytes)           : 112584
Solver global heap (bytes)    : 4044
And stack (bytes)             : 4044
Or stack (bytes)              : 4044
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 11152
Total memory used (bytes)     : 160036
```

```
Elapsed time since creation   : 0.09
```

C H A P T E R

8

Combining Constraints: Bin Packing

In this lesson, you will learn how to:

- combine constraints using logical operators
- modify the model during the solve phase
- use `IloIfThen`, `IloRange`, and `IloRangeArray`

Describe

The bin packing problem is a simplified example of a problem found in many industrial settings, such as configuration of telecommunications switching equipment. In this lesson, you will learn more about modeling techniques, including combining constraints. You will also learn how to modify a model during the solve phase—you will add constraints to the model during search.

In this problem you have components of different types and bins of various types. There are various constraints on which type of component can go into which type of bin. You must find the minimum total number of bins required to contain the components.

In this example there are five types of components: glass, plastic, steel, wood, copper. There are three types of bins: red, blue, green.

Each type of bin has a different capacity:

- red can contain 3 components
- blue can contain 1 component
- green can contain 4 components

Each type of bin can only contain certain types of components:

- red can contain glass, wood, copper
- blue can contain glass, steel, copper
- green can contain plastic, wood, copper

There are also some special requirements on the mixing of component types:

- wood requires plastic (a bin that contains wood must also contain plastic, but a bin that contains plastic does not necessarily also contain wood)
- glass excludes copper (a bin that contains glass cannot also contain copper)
- copper excludes plastic (a bin that contains copper cannot also contain plastic)

There are some limits on certain types of components in certain types of bins:

- red can contain at most 1 component of type wood
- green can contain at most 2 components of type wood

The demand for each type of component is:

- 2 components of type glass
- 4 components of type plastic
- 3 components of type steel
- 6 components of type wood
- 4 components of type copper

> ***Note:*** *This bin packing example is based on a problem designed by André Vellino. Vellino created this bin packing problem while working on a problem dealing with the configuration of telecom switching equipment. In this problem, different shelves can contain line cards subject to capacity constraints. The shelves have requirement and exclusion constraints. The problem is how to distribute components to minimize the number of shelves.*

Step 1 Describe the problem

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

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

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

Discussion

What is the known information in this problem?

- There are three types of bins: red, blue, green.
- There are five types of components: glass, plastic, steel, wood, copper.
- There is the following demand for each type of component: 2 glass components, 4 plastic components, 3 steel components, 6 wood components, and 4 copper components.

What are the decision variables in this problem?

- What is the type of each bin?
- How many components of each type are in each bin?

What are the constraints in this problem?

- There are a variety of constraints in this problem. Most of them are summarized in the following table:

Table 8.1 *Bin packing constraints*

Bin	Bin Capacity	Component				
		glass	plastic	steel	wood	copper
red	3	yes			at most 1	yes
blue	1	yes		yes		yes
green	4		yes		at most 2	yes

There are a few other constraints relating to the mixing of different types of components in the bins:

- wood requires plastic (a bin that contains wood must also contain plastic, but a bin that contains plastic does not necessarily also contain wood)
- glass excludes copper (a bin that contains glass cannot also contain copper)
- copper excludes plastic (a bin that contains copper cannot also contain plastic)

There are constraints relating to the total number of bins:

- the sum over all the bins of the quantities of a given component must be equal to the demand for that component
- the total capacity must be equal to or exceed the total demand

What is the objective?

- To minimize the number of bins used

Model

Once you have written a description of your problem, you can use Concert Technology classes to model it. You are going to model the bin packing problem in a different way from the way that you have modeled problems in previous lessons. In this lesson, you use classes to represent some of the model. You would normally want to create most industrial applications in this way, using object orientation.

You are going to create a class `Bin`. Instances of the class `Bin` represent the different bins used. Each instance of the class `Bin` will be described by three properties: type, capacity, and an array representing the number of each type of component in the bin. Each instance of the class `Bin` also adds the constraints to the model that affect each bin: constraints on bin capacity, bin contents, and the mixing of types of components in bins.

Step 2 Open the example file

Open the example file `YourSolverHome/examples/src/tutorial/binpacking_partial.cpp` in your development environment.

First, you create an environment and a model. This code is provided for you:

```
int main(){
  IloEnv env;
  try {
    IloModel model(env);
    IloInt i;
```

In this lesson, you first define the class `Bin` with data members, a constructor, and a method to display the solution. The data members are the decision variables of the problem: the type

of bin, the capacity of the bin, and an array representing the number of each type of component in the bin. This code is provided for you:

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

> ***Note:*** *For the sake of clarity in this example, the constrained integer variables are declared as* `public` *data members. However, good software engineering practices recommend* `private` *data members in similar cases for large programs.*

Step 3 Declare the constructor

Add the following code after the comment `//Declare the constructor`

```
Bin::Bin (IloModel model,
          IloIntArray Capacity,
          IloInt nTypes,
          IloInt nComponents) {
```

The constructor for an instance of `Bin` takes four parameters. The first parameter is the model. The second parameter is an array `Capacity` that contains the capacities of the different types of bins. The third parameter is `nTypes`, the number of types of bins. The fourth parameter is `nComponents`, the number of types of components.

In the constructor, you also declare the decision variables for each instance of `Bin`.

Step 4 Declare the decision variables in the constructor

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

```
  IloEnv env = model.getEnv();
  _type = IloIntVar(env, 0, nTypes-1);
  _capacity = IloIntVar(env, 0, IloMax(Capacity));
  _contents = IloIntVarArray(env, nComponents, 0, 4);
```

The first variable _type is an IloIntVar with a lower bound of 0 and an upper bound of nTypes-1. The second variable _capacity is an IloIntVar with a lower bound of 0. The upper bound is the maximum value in the array Capacity. The array of variables _contents is an IloIntVarArray representing the number of components of each type in the bin. The size of the array is nComponents, the lower bound is 0, and the upper bound is 4.

The decision variables _type and _contents will contain the solution to the problem, once it is solved. Declaring a decision variable for _capacity may seem redundant to you, since the capacity of a bin can be easily determined from its type. However, the decision variable _capacity is used in creating constraints and finding a solution. Since constraints are placed on this variable, it needs to be a decision variable.

Next, you create the constraints that affect each bin: constraints on bin capacity, bin contents, and the mixing of types of components in bins. All of these constraints have to be true for each bin and are created in the constructor of the class. These constraints are called *class constraints* since they will be true for all the instances of Bin. Each constraint must also be added to the model using the member function IloModel::add.

Step 5 Create the capacity constraints in the constructor

Add the following code after the comment
//Create the capacity constraints in the constructor

```
model.add(_capacity == Capacity(_type));
model.add(_capacity >= IloSum(_contents));
model.add(0. < IloSum(_contents));
```

The first constraint states that the variable _capacity must be equal to the value in the array Capacity corresponding to the type of bin, which is given by the variable _type. The second constraint states that the variable _capacity must be greater than or equal to the sum of the values in the array _contents, which is the sum of the number of components of each type in the bin. The third constraint states that bin must not be empty; 0 must be less than the sum of the values in the array _contents.

Next, you create the constraints that state what type of components can go in what type of bin. To do this, you use metaconstraints and the predefined constraint IloIfThen. The constructor for IloIfThen creates a condition constraint. The first parameter is the environment. The parameter left indicates the IF clause. The parameter right indicates the THEN clause. The fourth optional parameter is a name used for debug and trace purposes. Here is a constructor for IloIfThen:

```
IloIfThen (const IloEnv env,
           const IloConstraint left,
           const IloConstraint right,
           const char* name = 0);
```

For example, to create the constraints relating specifically to red bins you use `IloIfThen`. This is a metaconstraint because the constraint `IloIfThen` is a constraint placed on other constraints—the constraints representing the IF and THEN clauses. The IF part of the condition is represented by a constraint that uses the logical operator `==` to evaluate if the variable `_type` is equal to `red`. The THEN clause then performs an action based on this evaluation. The THEN part of the constraint is a metaconstraint itself consisting of three constraints joined by the logical AND operator `&&`. The first constraint uses the logical operator `==` to set the number of components of type `plastic` equal to `0`. The second constraint uses the logical operator `==` to set the number of components of type `steel` equal to `0`. The third constraint uses the logical operator `<=` to set the number of components of type `wood` to less than or equal to `1`. In other words, IF the type of bin is equal to red, THEN the bin is constrained to have no plastic components, no steel components, and no more than 1 wood component.

Step 6 Create the constraints on bin type red in the constructor

Add the following code after the comment
`//Create the constraints on bin type red in the constructor`

```
  model.add(IloIfThen(env, _type == red,
                      ((_contents[plastic] == 0)
                       && (_contents[steel] == 0)
                       && (_contents[wood] <= 1))));
```

You use the same technique to create the constraints on bins of type `blue` and `green` in the constructor. This code is provided for you:

```
  model.add(IloIfThen(env, _type == blue,
                      ((_contents[plastic] == 0) && (_contents[wood] == 0))));
  model.add(IloIfThen(env, _type == green,
                      ((_contents[glass] == 0)
                       && (_contents[steel] == 0)
                       && (_contents[wood] <= 2))));
```

You also use metaconstraints, logical operators, and the constraint `IloIfThen` to state the constraints relating to the mixing of different types of components in the bins. This code is provided for you:

```
  model.add(IloIfThen(env, _contents[wood] != 0, _contents[plastic] != 0));
  model.add(((_contents[glass] == 0)  || (_contents[copper]  == 0)));
  model.add(((_contents[copper] == 0) || (_contents[plastic] == 0)));
}
```

You create the function `display` to display the solution, including bin type, capacity, and the number of each type of component contained in the bin. This code is provided for you:

```
void Bin::display(const IloSolver solver) {
  _type.getEnv().out() << "Type :\t"
                       << Types[(IloInt)solver.getValue(_type)] << endl;
  _type.getEnv().out() << "Capacity:\t"
                       << solver.getValue(_capacity) << endl;
  IloInt nComponents = _contents.getSize();
  for (IloInt i = 0; i < nComponents; i++)
    if (solver.getValue(_contents[i]) > 0)
      _type.getEnv().out() << Components[i] << " :  \t"
                           << solver.getValue(_contents[i]) << endl;
  _type.getEnv().out() << endl;
}
```

After you have completed the declaration of the `Bin` class and its constructor and member function, the next step is to create an environment and a model in the `main` function. The variable `i` will be used in a “for” loop. Since you already know how to do this, it is provided for you in the exercise code:

```
int main(){
  IloEnv env;
  try {
    IloModel model(env);
    IloInt i;
```

You create and initialize `nComponents`, representing the number of types of components, and `nTypes`, representing the number of types of bins. This code is provided for you:

```
    const IloInt nComponents = 5;
    const IloInt nTypes = 3;
```

You create an array `Demand`, with `nComponents` members. This array contains the demand for each type of component. There is the following demand for each type of component: 2 glass components, 4 plastic components, 3 steel components, 6 wood components, and 4 copper components. This code is provided for you:

```
    IloIntArray Demand(env, nComponents, 2, 4, 3, 6, 4);
```

You create an array `Capacity`, with `nTypes` members. This array contains the capacity of each type of bin: red bins have a capacity of 3, blue bins have a capacity of 1, and green bins have a capacity of 4. This code is provided for you:

```
    IloIntArray Capacity(env, nTypes, 3, 1, 4);
```

You create two variables, `totalDemand` and `maxBin`. The variable `totalDemand` is equal to the sum of the demands for each type of component. In this example, `totalDemand`

equals 19. This is the sum of the demand for each type of component $(2 + 4 + 3 + 6 + 4)$. You then set maxBin equal to totalDemand. This means that, at the most, you would need one bin for each component. This code is provided for you:

```
IloInt totalDemand = (IloInt)IloSum(Demand);
const IloInt maxBin = (IloInt)totalDemand;
```

Next, you use range constraints to implement the following constraints:

- the total capacity of all the bins must be at least equal to the total demand for all components
- the amount of each type of component in all the bins must equal the demand for each type of component

You use the predefined constraints IloRange and IloRangeArray. The constructor for IloRange creates an empty range. The first parameter is the environment. The second and third parameters are the lower and upper bounds of the range. The fourth optional parameter is a name used for debug and trace purposes. Here is a constructor:

```
IloRange(const IloEnv env,
         IloNum lb,
         IloNum ub,
         const char* name = 0);
```

The constructor for IloRangeArray creates an array of empty ranges. The first parameter is the environment. The second and third parameters are arrays. The number of elements in the range array will equal the number of elements in the arrays lbs or ubs (which must both contain the same number of elements). The lower bound of each element in the range array will be equal to the corresponding element in the array lbs. The upper bound of each element in the range array will be equal to the corresponding element in the array ubs. Here is a constructor:

```
IloRangeArray(const IloEnv env,
              const IloNumArray lbs,
              const IloNumArray ubs);
```

Step 7 Create the range constraints

Add the following code after the comment //Create the range constraints

```
IloRange totalCapacityCon(env, totalDemand, IloIntMax);
model.add(totalCapacityCon);
IloRangeArray componentsDemandCon(env, Demand, Demand);
model.add(componentsDemandCon);
```

The constraint `totalCapacityCon` creates an empty range with a lower bound equal to `totalDemand` (or 19 in this example) and an upper bound of `IloIntMax`. This constraint states that the total capacity of all the bins must be at least equal to the total demand for components. The constraint `componentsDemandCon` creates an empty array of ranges. Each range in the array has the same lower and upper bound: the values of the array `Demand`. This constraint states that the amount of each type of component in all the bins must equal the demand for each type of component. For example, the demand for glass is 2 components. Therefore, for this constraint to be satisfied the contents of all the bins must sum to exactly 2 components of type glass, and likewise for the other types of components.

Solve

You are going to solve this problem in a different way than you solved other problems. In previous lessons, you created a model, declaring variables and adding constraints, and then solved the problem.The two stages of modeling and solving were clearly separated. In this problem, you solve in an incremental fashion. First you create one bin and fill it, making sure that all class constraints are satisfied. If all of the demand for each type of component has been met, then you have a solution. If not, Solver creates a second bin and fills it, making sure all class constraints are satisfied. When the problem is solved, you can be sure that the minimum number of bins has been used because of this incremental process. The objective, which is to minimize the number of bins used, is treated implicitly by the incremental solving method. It is never formally added to the model.

This example will show you how to add constraints to the model during the solving phase. In this problem, some of the constraints—the range constraints and the constraints to reduce symmetry—are defined with respect to the total number of bins used. For example, the constraint that the total capacity must exceed the demand depends on the number of bins. As a consequence, these constraints have to be *redefined* at each step since the total number of bins used so far changes at each step. This redefinition is carried out inside a “do-while” loop. Inside that loop, Solver detects whether a solution has been found yet. If there is no solution, Solver increments the number of bins used by one.

The following chart shows the process you will use to solve this problem:

```
solution = false
DO
     create a new bin
     add the bin to the range constraints
     IF there are at least 2 bins
          add the symmetry reduction constraints
     IF no solution is found
          the new bin becomes the previous bin
          increment the number of bins
     ELSE
          solution = true
WHILE solution = false and the number of bins
      is less than maxBin, continue DO
IF solution = true
      display the solution
```

First, you create pointers to the instances of the `Bin` class. The pointer `newBin` points to the instance of the class `Bin` just created inside the "do-while" loop. The pointer `prevBin` points to the instance of the class `Bin` created in the previous "do-while" loop and is initialized at `0`. The pointer `theBins` points to an array of all the instances of class `Bin` created so far. The maximum number of possible instances of `Bin` is `maxBin`, or 19 in this example.

Step 8 Create the pointers to the bins

Add the following code after the comment `//Create the pointers to the bins`

```
Bin* prevBin  =0;
Bin* newBin;
Bin** theBins = new Bin*[maxBin];
IloInt nBins = 1;
```

A Boolean variable `solution` is created and initialized as false, which means that when you start there is no solution. This code is provided for you:

```
IloBool solution = IloFalse;
```

Now you create an instance of the class `IloSolver` to solve the problem expressed in the model.

Step 9 Create an instance of IloSolver

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

```
IloSolver solver(model);
```

You now create the "do-while" loop in the solving phase. The first step in the loop is to create a new instance of the class Bin. Solver creates a new instance of class Bin and adds it to the array of bins theBins.

Step 10 Create an instance of Bin

Add the following code after the comment //Create an instance of Bin

```
do{
  newBin = new Bin(model, Capacity, nTypes, nComponents);
  theBins[nBins-1] = newBin;
```

Next, you add the bins to the range constraints you created in Step 7 on page 147.

Step 11 Add the bin to the range constraints

Add the following code after the comment //Add the bin to the range constraints

```
totalCapacityCon.setCoef(newBin->_capacity, 1.);
for (i = 0; i < nComponents; i++)
  componentsDemandCon[i].setCoef(newBin->_contents[i], 1.);
```

The capacity of the bin is added to the range totalCapacityCon. The capacity of the first bin will not equal totalDemand, 19 in this example, so the constraint is not satisfied. In the next "do-while" loop, the capacity of the second bin is added to the capacity of the first bin, and the sum is tested to see if it is equal to totalDemand. When the sum of the capacities of all the bins equals totalDemand, the constraint totalCapacityCon is satisfied.

A "for" loop is used to populate the range array componentsDemandCon. The contents of the first bin are added to the range array by component type. For example, if the first bin contained 2 components of type glass, the range array componentsDemandCon would be [2,0,0,0,0]. In order for this constraint to be satisfied, the range array componentsDemandCon needs to equal the array Demand [2,4,3,6,4]. So in this case, the constraint would not be satisfied. In the next "do-while" loop, the contents of the second bin is added to contents of the first bin, and the sums of each type of component is tested to see if they are equal to the elements of the array Demand. When the sums are equal to the elements of the array Demand, the constraint componentsDemandCon is satisfied.

After adding the range constraints to the "do-while" solving loop, you now add constraints to reduce symmetry if there are at least 2 bins. One way to reduce symmetry is to group

variables by type. Another way to reduce symmetry is to introduce order among variables, which is what you will do in this lesson.

Introduce order among variables

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

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

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

You add constraints that state that the bins must be added in increasing order by type. This constraint removes only symmetric solutions; it does not eliminate any "real" solutions.

For example, if your solution in this problem requires 3 bins, one of each color, all of the following solutions are correct:

- 1 red bin, 1 blue bin, 1 green bin
- 1 red bin, 1 green bin, 1 blue bin
- 1 blue bin, 1 green bin, 1 red bin
- 1 blue bin, 1 red bin, 1 green bin
- 1 green bin, 1 blue bin, 1 red bin
- 1 green bin, 1 red bin, 1 blue bin

These are six different solutions. However, in a physical sense, all of these solutions are identical; they all consist of 1 red bin, 1 blue bin, and 1 green bin.

> ***Note:*** *In problems such as car sequencing, the order of the variables is important. In those types of problems, you would not want to introduce an artificial order.*

You introduce order by creating two more constraints, which are added to the model during the solving phase. The first constraint states that the type of each new bin should be greater than or equal to the type of the previous bin. This constraint orders the variables, first selecting bins of type red, then bins of type blue, and then bins of type green. After adding the constraint to reduce symmetry based on type of bin, you also need a second constraint that deals with the fact that there may be more than one bin of any given type in your solution. For example, if your solution includes one bin of type red with 1 component of type glass, 1 bin of type blue with 1 component of type glass, and 1 bin of type blue with 1 component of type steel, both of the following results are correct:

- Bin 1: red bin—1 glass, Bin 2: blue bin—1 glass, Bin 3: blue bin—1 steel
- Bin 1: red bin—1 glass, Bin 2: blue bin—1 steel, Bin 3: blue bin—1 glass

However, in a physical sense these two solutions are identical. One way to order these solutions is to assign a different weight to each type of component. You can use the function `IloScalProd` and the predefined constraint `IloIfThen` to do this. (The function `IloScalProd` was introduced in Chapter 3, *Using Arrays and Basic Search: Changing Money*.) First you create an array of coefficients. This is provided for you in the code:

```
IloIntArray coef(env, nComponents, 625, 125, 25, 5, 1);
```

Then you use `IloScalProd` to assign a different coefficient to each type of component. For example, components of type glass have a coefficient of 625. Components of type steel have a coefficient of 25. You add a constraint that states that, if the new bin and the previous bin are both the same type of bin, then the `IloScalProd` of the new bin must be greater than or equal to the `IloScalProd` of the previous bin. In the example just described, the two blue bins are the same type. However, they do not have the same `IloScalProd`. The blue bin with 1 component of type glass has a scalar product of $1 * 625 = 625$. The blue bin with 1 component of type steel has a scalar product of $1 * 25 = 25$. Therefore, after removing symmetric solutions, only this solution will be found:

- Bin 1: red bin—1 glass, Bin 2: blue bin—1 steel, Bin 3: blue bin—1 glass

Step 12 Add the constraints to reduce symmetry

Add the following code after the comments `//Add the constraints to reduce symmetry`

```
if (prevBin != 0){
  model.add(newBin->_type >= prevBin->_type);
  model.add(IloIfThen(env, prevBin->_type == newBin->_type,
                      IloScalProd(newBin->_contents, coef)
                      >= IloScalProd(prevBin->_contents, coef)  ));
}
```

You now search for a solution with the member function `IloSolver::solve`. The search is placed in an “if-else” statement. If the search does not find a solution, the current bin becomes the previous bin and the number of bins increments by 1. While there is no solution and while the number of bins is less than or equal to `maxBin`, which is 19 in this example, the program returns to the “do” portion of the loop. This loop creates a new bin, adds the new bin to the range and symmetry constraints, and searches for a solution. If search finds a solution, the variable `solution` becomes true. The “do-while” loop is broken and Solver displays the solution.

Step 13 Search for a solution

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

```
      solver.out() << "Search with " << nBins << endl;
      if (!solver.solve()){
        prevBin = newBin;
        nBins++;
      }
      else
        solution = IloTrue;
    }
    while (!solution && nBins <= maxBin);
```

You use the `display` function to display the solution: each bin is listed with its type, capacity, and contents. This code is provided for you:

```
    if (solution) {
      for(i = 0; i < nBins; i++){
        theBins[i]->display(solver);
      }
    }
```

Finally, you clean up the memory. This code is provided for you:

```
    for  (i = 0; i < nBins; ++i)
      delete theBins[i];
    delete [] theBins;
```

Step 14 Compile and run the program

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

```
Search with 1
Search with 2
Search with 3
Search with 4
Search with 5
Search with 6
Search with 7
Search with 8
Type :  Red
Capacity:       3
Glass :         2

Type :  Blue
Capacity:       1
Steel :         1

Type :  Blue
Capacity:       1
Steel :         1

Type :  Blue
Capacity:       1
Steel :         1

Type :  Green
Capacity:       4
Copper :        4

Type :  Green
Capacity:       4
Plastic :       1
Wood :          2

Type :  Green
Capacity:       4
Plastic :       1
Wood :          2

Type :  Green
Capacity:       4
Plastic :       2
Wood :          2
```

As you can see, the solution requires 8 bins. Each bin is displayed, including its type, capacity, and contents. The complete program is listed in “Complete program” on page 157. You can also view it online in the file `YourSolverHome/examples/src/binpacking.cpp`.

Review exercises

For answers, see “Suggested answers” on page 155.

1. What are the parameters of the constructor of the predefined constraint `IloIfThen`?
2. What are two ways of reducing symmetry in a model?
3. In the example you have just completed, you reduced symmetry in the model by ordering variables. In this exercise, change the order of the variables. Select bins of type green first, then bins of type blue, and then bins of type red. The scalar product of the new bin should still be greater than or equal to the scalar product of the previous bin. However, change the coefficients of types of components so that they are weighted in the following descending order: copper, wood, steel, plastic, and glass.

Suggested answers

Exercise 1

What are the parameters of the constructor of the predefined constraint `IloIfThen`?

Suggested Answer

The constructor for `IloIfThen` creates a condition constraint. The first parameter is the environment. The parameter `left` indicates the IF clause. The parameter `right` indicates the THEN clause. The fourth optional parameter is a name used for debug and trace purposes.

Exercise 2

What are two ways of reducing symmetry in a model?

Suggested Answer

1. Introduce order among variables.
2. Group variables by type.

Exercise 3

In the example you have just completed, you reduced symmetry in the model by ordering variables. In this exercise, change the order of the variables. Select bins of type green first, then bins of type blue, and then bins of type red. The scalar product of the new bin should still be greater than or equal to the scalar product of the previous bin. However, change the

coefficients of types of components so that they are weighted in the following descending order: copper, wood, steel, plastic, and glass.

Suggested Answer

The code that has changed from `binpacking.cpp` follows. You can view the complete program online in the file `YourSolverHome/examples/src/binpacking_ex3.cpp`.

You change the array of coefficients:

```
    IloIntArray coef     (env, nComponents, 1, 5, 25, 125, 625);
```

You change how the coefficients are weighted:

```
      if (prevBin != 0){
        model.add(newBin->_type <= prevBin->_type);
        model.add(IloIfThen(env, prevBin->_type == newBin->_type,
                            IloScalProd(newBin->_contents, coef)
                            >= IloScalProd(prevBin->_contents, coef)  ));
      }
```

You should get the following results:

```
Search with 1
Search with 2
Search with 3
Search with 4
Search with 5
Search with 6
Search with 7
Search with 8
Type :  Green
Capacity:       4
Plastic :       1
Wood :          2

Type :  Green
Capacity:       4
Plastic :       1
Wood :          2

Type :  Green
Capacity:       4
Plastic :       2
Wood :          2

Type :  Green
Capacity:       4
Copper :        4

Type :  Blue
Capacity:       1
Steel :         1

Type :  Blue
```

```
Capacity:       1
Steel :         1

Type :  Blue
Capacity:       1
Steel :         1

Type :  Red
Capacity:       3
Glass :         2
```

Complete program

The complete bin packing program follows. You can also view it online in the file `YourSolverHome/examples/src/binpacking.cpp`.

```
#include <ilsolver/ilosolverint.h>
ILOSTLBEGIN
enum Component {glass, plastic, steel, wood, copper};
const char* Components[5] = {"Glass", "Plastic", "Steel", "Wood", "Copper"};

enum Type {red, blue, green};
const char* Types[3] = {"Red", "Blue", "Green"};
class Bin {
public:
  IloIntVar       _type;
  IloIntVar       _capacity;
  IloIntVarArray  _contents;
  Bin (IloModel    mod,
       IloIntArray Capacity,
       IloInt      nTypes,
       IloInt      nComponents);
  void display(const IloSolver sol);
};
Bin::Bin (IloModel model,
          IloIntArray Capacity,
          IloInt nTypes,
          IloInt nComponents) {
  IloEnv env = model.getEnv();
  _type = IloIntVar(env, 0, nTypes-1);
  _capacity = IloIntVar(env, 0, IloMax(Capacity));
  _contents = IloIntVarArray(env, nComponents, 0, 4);
  model.add(_capacity == Capacity(_type));
  model.add(_capacity >= IloSum(_contents));
  model.add(0. < IloSum(_contents));
  model.add(IloIfThen(env, _type == red,
                      ((_contents[plastic] == 0)
                       && (_contents[steel] == 0)
                       && (_contents[wood] <= 1))));
  model.add(IloIfThen(env, _type == blue,
                      ((_contents[plastic] == 0) && (_contents[wood] == 0))));
  model.add(IloIfThen(env, _type == green,
```

```
                        ((_contents[glass] == 0)
                         && (_contents[steel] == 0)
                         && (_contents[wood] <= 2))));
  model.add(IloIfThen(env, _contents[wood] != 0, _contents[plastic] != 0));
  model.add(((_contents[glass] == 0)  || (_contents[copper]  == 0)));
  model.add(((_contents[copper] == 0) || (_contents[plastic] == 0)));
}
void Bin::display(const IloSolver solver) {
  _type.getEnv().out() << "Type :\t"
                       << Types[(IloInt)solver.getValue(_type)] << endl;
  _type.getEnv().out() << "Capacity:\t"
                       << solver.getValue(_capacity) << endl;
  IloInt nComponents = _contents.getSize();
  for (IloInt i = 0; i < nComponents; i++)
    if (solver.getValue(_contents[i]) > 0)
      _type.getEnv().out() << Components[i] << " :  \t"
                           << solver.getValue(_contents[i]) << endl;
  _type.getEnv().out() << endl;
}
int main(){
  IloEnv env;
  try {
    IloModel model(env);
    IloInt i;
    const IloInt nComponents = 5;
    const IloInt nTypes = 3;
    IloIntArray Demand(env, nComponents, 2, 4, 3, 6, 4);
    IloIntArray Capacity(env, nTypes, 3, 1, 4);
    IloIntArray coef(env, nComponents, 625, 125, 25, 5, 1);
    IloInt totalDemand = (IloInt)IloSum(Demand);
    const IloInt maxBin = (IloInt)totalDemand;
    Bin* prevBin  =0;
    Bin* newBin;
    Bin** theBins = new Bin*[maxBin];
    IloInt nBins = 1;
    IloBool solution = IloFalse;
    IloRange totalCapacityCon(env, totalDemand, IloIntMax);
    model.add(totalCapacityCon);
    IloRangeArray componentsDemandCon(env, Demand, Demand);
    model.add(componentsDemandCon);
    IloSolver solver(model);
    do{
      newBin = new Bin(model, Capacity, nTypes, nComponents);
      theBins[nBins-1] = newBin;
      totalCapacityCon.setCoef(newBin->_capacity, 1.);
      for (i = 0; i < nComponents; i++)
        componentsDemandCon[i].setCoef(newBin->_contents[i], 1.);
      if (prevBin != 0){
        model.add(newBin->_type >= prevBin->_type);
        model.add(IloIfThen(env, prevBin->_type == newBin->_type,
                            IloScalProd(newBin->_contents, coef)
                            >= IloScalProd(prevBin->_contents, coef)  ));
      }
      solver.out() << "Search with " << nBins << endl;
      if (!solver.solve()){
        prevBin = newBin;
        nBins++;
      }
```

```
      else
        solution = IloTrue;
    }
    while (!solution && nBins <= maxBin);
    if (solution) {
      for(i = 0; i < nBins; i++){
        theBins[i]->display(solver);
      }
    }
    for  (i = 0; i < nBins; ++i)
      delete theBins[i];
    delete [] theBins;
  }
  catch (IloException& ex) {
    cerr << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Results

```
Search with 1
Search with 2
Search with 3
Search with 4
Search with 5
Search with 6
Search with 7
Search with 8
Type :  Red
Capacity:       3
Glass :         2

Type :  Blue
Capacity:       1
Steel :         1

Type :  Blue
Capacity:       1
Steel :         1

Type :  Blue
Capacity:       1
Steel :         1

Type :  Green
Capacity:       4
Copper :        4

Type :  Green
Capacity:       4
Plastic :       1
Wood :          2

Type :  Green
Capacity:       4
```

```
Plastic :       1
Wood :          2

Type :  Green
Capacity:       4
Plastic :       2
Wood :          2
```

C H A P T E R

9

Using Table Constraints: Scheduling Teams

In this lesson, you will learn how to:

- use table constraints
- use constraints defined from allowed tuples
- use `IloTableConstraint` and `IloNumTupleSet`

Describe

In this lesson, you will search for a schedule for a group of sports teams. There are 8 teams that play over 7 weeks. In each week there are 4 time periods. (For example, these time periods could be Tuesday night, Thursday night, Friday night, and Saturday afternoon.) Teams can play at home (in the stadium in their home town) or away (at a stadium in another team's town). Each team must play each other team exactly once. Each team plays only once each week. No team plays in the same time period more than twice.

Step 1 Describe the problem

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

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

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

Discussion

What is the known information in this problem?

- There are 8 teams. These teams are known as Team 0, Team 1, Team 2, Team 3, Team 4, Team 5, Team 6, Team 7. (Solver arrays start at `[0]`.) Teams can play at home (in the stadium in their home town) or away (at a stadium in another team's town).
- There are 4 time periods in each week and 7 weeks.

Here is another way of looking at the information about periods and weeks. The intersection of each period and week forms a slot. This slot can be identified by the coordinates: the period, the week. There are 28 slots available for matches between teams. The matches can be identified by a single integer.

Table 9.1 *Slots for Scheduling*

	Week 0	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6
Period 0	Match 0 Slot (0,0)	Match 1 Slot (0,1)	Match 2 Slot (0,2)	Match 3 Slot (0,3)	Match 4 Slot (0,4)	Match 5 Slot (0,5)	Match 6 Slot (0,6)
Period 1	Match 7 Slot (1,0)	Match 8 Slot (1,1)	Match 9 Slot (1,2)	Match 10 Slot (1,3)	Match 11 Slot (1,4)	Match 12 Slot (1,5)	Match 13 Slot (1,6)
Period 2	Match 14 Slot (2,0)	Match 15 Slot (2,1)	Match 16 Slot (2,2)	Match 17 Slot (2,3)	Match 18 Slot (2,4)	Match 19 Slot (2,5)	Match 20 Slot (2,6)
Period 3	Match 21 Slot (3,0)	Match 22 Slot (3,1)	Match 23 Slot (3,2)	Match 24 Slot (3,3)	Match 25 Slot (3,4)	Match 26 Slot (3,5)	Match 27 Slot (3,6)

What are the variables or unknowns in this problem?

Basically, the unknown information is which teams play in which slots. However, this information can be broken down into the following variables:

- Which team is playing at home in each slot?

- Which team is playing away in each slot?
- Which slot? This is a variable represented by the coordinates: the period, the week.
- Which match is played in each slot? This variable represents roughly the same information as which slot, but is represented by a single integer.

What are the constraints on these variables?

- All matches are different. (Each team must play each other team exactly once.)
- Each team must play every week, but only once.
- No team plays in the same time period more than twice.
- The variables (home team, away team, slot, and match) must be linked to each other.

Model

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

Step 2 Open the example file

Open the example file `YourSolverHome/examples/src/tutorial/sports_basic_partial.cpp` in your development environment.

In this exercise, you use constrained integer variables and therefore you use the include file `<ilsolver/ilosolverint.h>`.

First, you represent the data of the program. The number of team, `nbTeams`, is set to 8 in this example, but can be easily modified. (For this problem, the number of teams must always be even. If the number of teams is not even, it is increased by 1.) The number of weeks, `nbWeeks`, is `nbTeams-1`, or 7 in this example. The number of periods, `nbPeriods`, is `nbTeams/2`, or 4 in this example. The number of slots, `nbSlots`, is equal to the number of weeks times the number of periods, or 28 in this example. This code is provided for you:

```
int main(int argc, char** argv){
  IloInt i;
  IloInt j;
  IloInt m;
  IloInt p;
  IloInt w;

  nbTeams = (argc>1? atol(argv[1] ) : 8 );
  if ( nbTeams % 2 ) ++nbTeams;
  nbWeeks = nbTeams-1;
  nbPeriods = nbTeams /2;
  nbSlots = nbWeeks * nbPeriods;
```

Next, you create an environment and a model. This code is provided for you:

```
IloEnv env;
  try {
    IloModel model(env);
```

First, you create a matrix, an array of arrays, to represent the `slotVars`, the slot variables. You use a `for` loop to create an array with a size of 4 (the number of periods). Then, for each period, you create an array of size 7 (the number of weeks). Each variable is represented by the coordinates: the period, the week.

Step 3 Create the matrix of slots

Add the following code after the comment `//Create the matrix of slots`

```
IloArray<IloIntVarArray> slotVars(env, nbPeriods);

for(i = 0; i < nbPeriods; i++)
  slotVars[i] = IloIntVarArray(env, nbWeeks, 0, nbSlots-1);
```

You create an array to represent the `matchVars`, the match variables. The match variables are identified by a single integer. The array `matchVars` has `nbSlots` elements, or 28 in this example. You use two nested `for` loops. You create one element of the array `matchVars` for each element of the matrix `slotVars`. For example, to represent the first period of the first week, you declare `matchVars[0]` which corresponds to `slotVars[0][0]`. (Remember that Solver arrays start at `[0]`.) You then loop to the second week in the first period, and declare `matchVars[1]` which corresponds to `slotVars[0][1]`. The loop continues until all `matchVars` elements are declared.

Step 4 Declare the match variables

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

```
IloIntVarArray matchVars(env, nbSlots);
IloInt s = 0;
for (p = 0; p < nbPeriods; ++p) {
  for (w = 0; w < nbWeeks; ++w) {
    matchVars[s] = slotVars[p][w];
    ++s;
  }
}
```

Next, you declare the home team and away team variables. One matrix of variables is created for `homeTeamVars` and a second matrix is created for `awayTeamVars`. The lower bound of possible values for each variable is 0 and the upper bound is `nbTeams-1` or 7. These values represent the teams: Team 0, Team 1, Team 2, Team 3, Team 4, Team 5, Team

6, Team 7. Each variable can also be represented by its matrix coordinates: the period, the week.

Step 5 Declare the home team and away team variables

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

```
IloArray<IloIntVarArray> homeTeamVars(env, nbPeriods);
IloArray<IloIntVarArray> awayTeamVars(env, nbPeriods);

for (p = 0; p < nbPeriods; ++p) {
  homeTeamVars[p]  = IloIntVarArray(env, nbWeeks, 0, nbTeams-1);
  awayTeamVars[p] = IloIntVarArray(env, nbWeeks, 0, nbTeams-1);
}
```

Now that you have declared all the variables, you need a way to link them together. Since the `matchVars` are defined based on `slotVars`, you can focus on linking `slotVars`, `homeTeamVars`, and `awayTeamVars`. Concert Technology offers a way to do this—table constraints. Table constraints provide a way to add a constraint using a table, a single store of related information. The information can be in the form of combinations of values that are allowed or in the form of combinations of values that are forbidden.The information used by a table constraint can also be in the form of a predicate. A predicate is a logical formula that contains variables. This formula can be evaluated as either true or false.

To demonstrate how table constraints work, imagine that the teams in this example are split into leagues. If Teams 0, 1, 2, and 3 are in League 0 and Teams 4, 5, 6, and 7 are in League 1, you might want to constrain that only teams in the same league can play each other. You could create a table, a set of allowed combinations of values (for example, Team 0 can play Team 2), and use the table constraint to state that only these combinations of values are allowed.

In this lesson, you use the table constraint to constrain that only certain configurations of home team, away team, and slots are allowed. To do this, you first create a tuple set. An integer tuple is an ordered set of values. For example, if you state that Team 1 must play Team 2 in Match 7, this is a tuple. You would write it as (1, 2, 7). The number of values in a tuple is known as the arity of the tuple. The tuple (1, 2, 7) has an arity of 3.

Sets of tuples are represented in Concert Technology by an instance of the class `IloIntTupleSet`. The first parameter of the constructor is the environment and the second parameter is the arity. You use the member function `IloIntTupleSet::add` to add tuples to the set. Here is a constructor for `IloIntTupleSet`:

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

Now, you create the tuple set.

Figure 9.1

Step 6 Create the tuple set

Add the following code after the comment `//Create the tuple set`

```
IloIntTupleSet tuple(env, 3);
IloInt m = 0;
  for (i=0; i < nbTeams; ++i) {
    for ( j=i+1; j < nbTeams; ++j) {
      tuple.add(IloIntArray(env, 3, i, j, m));
      ++m;
    }
  }
assert( m == nbSlots );
```

The set of tuples is created using a matrix (like much else in this lesson). The matrix is created using two nested `for` loops. The variable `i` represents the home team, the variable `j` represents the away team, and the variable `m` represents the match. The tuples are derived from this matrix. For example, the first tuple (`i`, `j`, `m`) is (0, 1, 0). This means that Team 0 (home) must play Team 1 (away) in Match 0. You can derive the rest of the tuples in the set from this matrix.

Table 9.2 *Tuple Sets*

	Team 0 Home	**Team 1 Home**	**Team 2 Home**	**Team 3 Home**	**Team 4 Home**	**Team 5 Home**	**Team 6 Home**	**Team 7 Home**
Team 0 Away								
Team 1 Away	Match 0							
Team 2 Away	Match 1	Match 7						
Team 3 Away	Match 2	Match 8	Match 13					
Team 4 Away	Match 3	Match 9	Match 14	Match 18				
Team 5 Away	Match 4	Match 10	Match 15	Match 19	Match 22			
Team 6 Away	Match 5	Match 11	Match 16	Match 20	Match 23	Match 25		
Team 7 Away	Match 6	Match 12	Match 17	Match 21	Match 24	Match 26	Match 27	

After creating the tuple set, you declare a table constraint to state that these tuples are the only allowed combinations of values for the constrained variables.

Table constraints are represented in Concert Technology by an instance of the class `IloTableConstraint`. The first parameter of the constructor is the environment. The

second parameter is an array of variables. The third parameter is a set of tuples that designates combinations of values. The fourth constraint is a Boolean. If this parameter is `IloTrue`, the tuples represent the allowed values for the array of variables. If this parameter is `IloFalse`, the tuples represent the forbidden values for the array of variables. Here is a constructor for `IloTableConstraint`:

```
IloConstraint IloTableConstraint(const IloEnv env,
                                 const IloNumVarArray vars,
                             const IloNumTupleSet set,
                                 IloBool compatible);
```

Now, you add the table constraint. This makes sure that home teams, away teams, and slots are properly linked. It also forbids impossible schedule combinations. For example, a team playing itself would not form part of the allowed set of tuples.

Step 7 Add the table constraint

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

```
for (p = 0; p < nbPeriods; ++p) {
  for (w = 0; w < nbWeeks; ++w) {
    IloIntVarArray slotConsistencyTest(env, 3);
    slotConsistencyTest[0] = homeTeamVars[p][w];
    slotConsistencyTest[1] = awayTeamVars[p][w];
    slotConsistencyTest[2] = slotVars[p][w];
    model.add(IloTableConstraint(env,
                                 slotConsistencyTest,
                                 tuple,
                                 IloTrue));
  }
}
```

Again, the code creates a matrix based on periods and weeks. At the place in the matrix designated by the coordinates Period 0 and Week 0, you create an array of variables `slotConsistencyTest`. This array has three elements. The first element is `homeTeamVars[0][0]` or the home team playing in the slot Period 0 of Week 0. The second element is `awayTeamVars[0][0]` or the away team playing in the slot Period 0 of Week 0. The third element is `slotVars[0][0]` or the slot representing Period 0 of Week 0 (and also `matchVars[0]` which is created from `slotVars`). Then you add the table constraint. It constrains that the three variables in `slotConsistencyTest` must be assigned values that are represented in one of the tuple sets declared in `tuple`. For example, an accepted combination of values is (0, 1, 0). This means that Team 0 must play Team 1 in Match 0.

Now that you have linked the variables, you can add the other constraints.

You use the predefined constraint `IloAllDiff` to add the constraint that all matches are different. (The constraint `IloAllDiff` is introduced in Chapter 4, *Searching with*

Predefined Goals: Magic Square.) In other words, each team must play each other exactly once. In this constraint, you use the array of variables `matchVars`. Each variable in this array has a value of an integer and `IloAllDiff` constrains that all these values must be different. (There are three representations of matches in this model: `matchVars`, `slotVars`, and the value `m` representing the match in each tuple. They are linked to each other through the table constraint.)

Step 8 Add the constraint that all matches are different

Add the following code after the comment
`//Add the constraint that all matches are different`

```
model.add( IloAllDiff(env, matchVars));
```

You also use `IloAllDiff` to add the constraint that every team must play each week, but only once. For each week, you create an array of variables `weekTeams`. For each period in this week, you create two variables, one to represent the home team playing in that slot and one to represent the away team playing in that slot. Then you use `IloAllDiff` to constrain that the 8 variables (one home team and one away team in each of 4 periods) in the array `weekTeams` must all be different. You use a `for` loop to add this constraint for each week.

Step 9 Add the constraint that every team must play each week, but only once

Add the following code after the comment
`//Add the constraint that every team must play each week, but only once`

```
for (w = 0; w < nbWeeks; ++w) {
  IloIntVarArray weekTeams(env, nbTeams);
  for (p = 0; p < nbPeriods; ++p) {
    weekTeams[2*p]   = homeTeamVars [p][w];
    weekTeams[2*p+1] = awayTeamVars[p][w];
  }
  model.add( IloAllDiff(env, weekTeams) );
}
```

You use the predefined constraint `IloDistribute` to add the constraint that each team plays at least once, but no more than twice in each period. As you remember from Chapter 6, *Using the Distribute Constraint: Car Sequencing*, the constraint `IloDistribute` takes three arrays as parameters: `cards`, `values`, and `vars`. The variables in the array `cards` are equal to the number of occurrences in the array `vars` of the values in the array `values`.

First, you create the array of possible values `teamValues`. The values in this array represent the teams: Team 0, Team 1, Team 2, Team 3, Team 4, Team 5, Team 6, Team 7.

Step 10 Declare the teamValues array

Add the following code after the comment `//Declare the teamValues array`

```
IloIntArray teams(env, nbTeams);
  for (m = 0; m < nbTeams; ++m)
    teams[m] = m;
```

Then, you create the array of variables. For each period, you create an array of variables `periodTeamVars`. For each week in this period, you create two variables, one to represent the home team playing in that slot and one to represent the away team playing in that slot. These variables can take the values declared in the array `teamValues` array.

Step 11 Declare the periodTeamVars array

Add the following code after the comment `//Declare the periodTeamVars array`

```
for (p = 0; p < nbPeriods; ++p) {
  IloIntVarArray periodTeams(env, 2*nbWeeks);
  for ( w = 0; w < nbWeeks; ++w) {
    periodTeams[ 2*w ]   = homeTeamVars [p][w];
    periodTeams[ 2*w+1 ] = awayTeamVars[p][w];
  };
```

Since you know that teams must play in at least one game, but no more than two games, the array `cards` is easy to create. Each variable in this array can have a value of either 1 or 2.

Step 12 Declare the cards array

Add the following code after the comment `//Declare the cards array`

```
  IloIntVarArray cardVars(env, nbTeams, 1, 2);
```

Now, you can add the distribute constraint to the model.

Step 13 Add the distribute constraint

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

```
  model.add( IloDistribute(env, cardVars, teams, periodTeams) );
}
```

This constraint states that the array of variables `periodTeamVars` can take any value in the array `TeamValues`. However, the values must all be used at least once, but not more than two times. In other words, each team must play at least once, but not more than twice in each period.

Solve

Now you create an instance of the class `IloSolver` to solve the problem expressed in the model.

Step 14 Create an instance of IloSolver

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

```
IloSolver solver(model);
```

Then you search for a solution using the member function `IloSolver::solve`. The solution is displayed using two nested `for` loops. For each period and week, the home team and the away team are displayed.

Step 15 Search for a solution

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

```
if (solver.solve()) {
  cout << endl << "SOLUTION" << endl;
  for (p=0; p < nbPeriods; ++p) {
       cout << "period " << p << " : ";
         for (w=0; w < nbWeeks; ++w) {
         cout << solver.getValue(homeTeamVars[p][w]) << " vs "
             << solver.getValue(awayTeamVars[p][w]) << " - " ;
       }
   cout << endl;
  }
  solver.printInformation();
    }
else
  cout << "**** NO SOLUTION ****" << endl;
```

Step 16 Compile and run the program

Compile and run the program. You should get the following results, though the information displayed by `IloSolver::printInformation` will vary depending on platform, machine, configuration, and so on:

```
%
* sports scheduling teams: 8 weeks: 7 periods: 4

SOLUTION
period 0 : 0 vs 1 - 0 vs 2 - 1 vs 2 - 3 vs 4 - 3 vs 5 - 4 vs 5 - 6 vs 7 -
period 1 : 2 vs 3 - 3 vs 6 - 4 vs 6 - 0 vs 7 - 0 vs 4 - 2 vs 7 - 1 vs 5 -
period 2 : 4 vs 7 - 5 vs 7 - 0 vs 5 - 1 vs 6 - 2 vs 6 - 1 vs 3 - 0 vs 3 -
period 3 : 5 vs 6 - 1 vs 4 - 3 vs 7 - 2 vs 5 - 1 vs 7 - 0 vs 6 - 2 vs 4 -
%
```

The first period (`period 0`) has the following schedule. In Week 1, Team 0 is home and Team 1 is away. In Week 2, Team 0 is home and Team 2 is away. In Week 3, Team 1 is home and Team 2 is away. In Week 4, Team 3 is home and Team 4 is away. In Week 5, Team 3 is home and Team 5 is away. In Week 6, Team 4 is home and Team 5 is away. In Week 7, Team 6 is home and Team 7 is away. You can follow the schedule for the other three periods.

The complete program is listed in "Complete program" on page 172. You can also view it online in the file `YourSolverHome/examples/src/sports_basic.cpp`.

Review exercises

For answers, see "Suggested answers" on page 171.

1. What is a table constraint?
2. What is a tuple?
3. How would adding a "fake" week 8 to the model affect the problem solution?

Suggested answers

Exercise 1

What is a table constraint?

Suggested Answer

Table constraints provide a way to add a constraint using a table, a single store of related information. The information can be in the form of combinations of values that are allowed

or in the form of combinations of values that are forbidden.The information used by a table constraint can also come in the form of a predicate. A predicate is a logical formula that contains variables. This formula can be evaluated as either true or false.

Exercise 2

What is a tuple?

Suggested Answer

An tuple is an ordered set of values. For example, if you state that Team 1 must play Team 2 in Match 7, this is a tuple. You would write it as (1, 2, 7).

Exercise 3

How would adding a “fake” week 8 to the model affect the problem solution?

Suggested Answer

By introducing a “fake” week 8 to the model, you can prove that each team plays exactly twice in each period and each team plays exactly once in the “fake” week 8. The solver is helped by these additional constraints. It can deduce inconsistencies more quickly and should be able to find a solution more quickly.

Complete program

The complete table constraint program follows. You can also view it online in the file `YourSolverHome/examples/src/sports_basic.cpp`.

```
#include <ilsolver/ilosolver.h>
ILOSTLBEGIN

IloInt nbTeams;
IloInt nbWeeks;
IloInt nbPeriods;
IloInt nbSlots;

int main(int argc, char** argv){
  IloInt i;
  IloInt j;
  IloInt m;
  IloInt p;
  IloInt w;

  nbTeams = (argc>1? atol(argv[1] ) : 8 );
  if ( nbTeams % 2 ) ++nbTeams;
  nbWeeks = nbTeams-1;
  nbPeriods = nbTeams /2;
  nbSlots = nbWeeks * nbPeriods;
  IloEnv env;
```

```
    try {
      IloModel model(env);

      IloArray<IloIntVarArray> slotVars(env, nbPeriods);

      for(i = 0; i < nbPeriods; i++)
        slotVars[i] = IloIntVarArray(env, nbWeeks, 0, nbSlots-1);
// ie : NumVarMatrix slotVars = IloArray<IloNumVarArray>(env, nbPeriods);
// build copy vector of match variables matrix
      IloIntVarArray matchVars(env, nbSlots);
      IloInt s = 0;
      for (p = 0; p < nbPeriods; ++p) {
        for (w = 0; w < nbWeeks; ++w) {
          matchVars[s] = slotVars[p][w];
          ++s;
        }
      }
      IloArray<IloIntVarArray> homeTeamVars(env, nbPeriods);
      IloArray<IloIntVarArray> awayTeamVars(env, nbPeriods);

      for (p = 0; p < nbPeriods; ++p) {
        homeTeamVars[p]  = IloIntVarArray(env, nbWeeks, 0, nbTeams-1);
        awayTeamVars[p] = IloIntVarArray(env, nbWeeks, 0, nbTeams-1);
      }
// --------------------------------
// Constraint 1 : each team plays each other exactly once,
// i.e. a match (T1, T2) occurs exactly once.
// --------------------------------
      model.add( IloAllDiff(env, matchVars));
// --------------------------------
// Constraint 2 : each week, all teams do play.
// remember 2*nbPeriods == nbTeams.
// --------------------------------
      for (w = 0; w < nbWeeks; ++w) {
        IloIntVarArray weekTeams(env, nbTeams);
        for (p = 0; p < nbPeriods; ++p) {
          weekTeams[2*p]   = homeTeamVars [p][w];
          weekTeams[2*p+1] = awayTeamVars[p][w];
        }
        model.add( IloAllDiff(env, weekTeams) );
      }
// --------------------------------
// constraint 3 : in each period, each team plays no more than twice.
// one can demonstrate that the number of occurences of each team in
// one period is at least 1, so we constrain it to be in [1..2]
// --------------------------------
      IloIntArray teams(env, nbTeams);
        for (m = 0; m < nbTeams; ++m)
          teams[m] = m;
          for (p = 0; p < nbPeriods; ++p) {
            IloIntVarArray periodTeams(env, 2*nbWeeks);
            for ( w = 0; w < nbWeeks; ++w) {
              periodTeams[ 2*w ]   = homeTeamVars [p][w];
              periodTeams[ 2*w+1 ] = awayTeamVars[p][w];
            };
            IloIntVarArray cardVars(env, nbTeams, 1, 2);
            model.add( IloDistribute(env, cardVars, teams, periodTeams) );
          }
```

```
// --------------------------------
// Constraint 4: link team and match vars with table constraint
// --------------------------------
      IloIntTupleSet tuple(env, 3);
      IloInt m = 0;
        for (i=0; i < nbTeams; ++i) {
          for ( j=i+1; j < nbTeams; ++j) {
            tuple.add(IloIntArray(env, 3, i, j, m));
            ++m;
          }
        }
      assert( m == nbSlots );
      for (p = 0; p < nbPeriods; ++p) {
        for (w = 0; w < nbWeeks; ++w) {
          IloIntVarArray slotConsistencyTest(env, 3);
          slotConsistencyTest[0] = homeTeamVars[p][w];
          slotConsistencyTest[1] = awayTeamVars[p][w];
          slotConsistencyTest[2] = slotVars[p][w];
          model.add(IloTableConstraint(env,
                                       slotConsistencyTest,
                                       tuple,
                                       IloTrue));
        }
      }
      IloSolver solver(model);
      if (solver.solve()) {
        cout << endl << "SOLUTION" << endl;
        for (p=0; p < nbPeriods; ++p) {
             cout << "period " << p << " : ";
               for (w=0; w < nbWeeks; ++w) {
               cout << solver.getValue(homeTeamVars[p][w]) << " vs "
                   << solver.getValue(awayTeamVars[p][w]) << " - " ;
             }
         cout << endl;
        }
        solver.printInformation();
          }
      else
        cout << "**** NO SOLUTION ****" << endl;
      }
    catch (IloException& ex) {
       cerr << "Error: " << ex << endl;
    }
  env.end();
  return 0;
}
```

Results

The information displayed by `IloSolver::printInformation` will vary depending on platform, machine, configuration, and so on.

```
%
* sports scheduling teams: 8 weeks: 7 periods: 4
```

```
SOLUTION
period 0 : 0 vs 1 - 0 vs 2 - 1 vs 2 - 3 vs 4 - 3 vs 5 - 4 vs 5 - 6 vs 7 -
period 1 : 2 vs 3 - 3 vs 6 - 4 vs 6 - 0 vs 7 - 0 vs 4 - 2 vs 7 - 1 vs 5 -
period 2 : 4 vs 7 - 5 vs 7 - 0 vs 5 - 1 vs 6 - 2 vs 6 - 1 vs 3 - 0 vs 3 -
period 3 : 5 vs 6 - 1 vs 4 - 3 vs 7 - 2 vs 5 - 1 vs 7 - 0 vs 6 - 2 vs 4 -
%
```

C H A P T E R

10

Reducing Symmetry: Configuring Racks

In this lesson, you will learn how to:

- reduce symmetries in a model
- create a class to represent the racks
- use the constraints `IloScalProd` and `IloSum`

This chapter solves a configuration problem where electronic cards must be plugged into a number of racks at minimal cost.

> ***Note:*** *The subject matter of this chapter lends itself more naturally to a conceptual introduction rather than procedural-based learning. It is not presented using steps, as in previous chapters in this part.*

Describe

A set of electronic cards must be plugged into racks with electric connectors. Different types of racks can hold different sets of cards, incurring different costs. Given a set of cards and a maximal number of racks to use, you must find a set of (at most) the given number of racks to fit in all the cards, while you hold the cost to a minimum.

A rack has the following features:

- a maximal power it can provide
- a number of connection slots—each slot can receive one card
- its price—the price of a rack depends on its number of connection slots and on the maximal power it can provide

For cards, consider only the power they use.

Recognizing the constraints

The constraints of this example can be summarized this way:

- plug all given cards into racks
- use at most a given number of racks
- each rack has a given number of connectors and can receive at most one card per connector
- the sum of power used by the cards plugged into a rack must be less than or equal to the maximal power of this rack

Determining the optimization criterion

As the maximal number of racks to use is fixed, the objective is to find two points about each rack used:

- the cards it receives
- its correct type according to the number of plugged cards and the power it must provide

You will search for a solution that minimizes the total cost (that is, the sum of rack prices).

Choosing types of racks

For example, assume that there are two possible types of racks and four types of cards that use the following power 20, 40, 50, and 75. The features of racks would be given by the following chart.

Table 10.1 *Rack Type*

Rack Type	Power	Connectors	Price
1	150	8	150
2	200	16	200

You want to plug in the following cards, using at most 5 racks:

Table 10.2 *Card Type*

Card Type	Power	Number
1	20	10
2	40	4
3	50	2
4	75	1

Model

You use Concert Technology classes to model this problem.

This program reads the description of racks and cards in the file `rack.dat`. First, the data are represented like this in the code:

```
const IloInt nbRackTypes = 2;
const IloInt maxSlots = 16;
enum {power, price, nbSlot};

IloInt Racks[3][nbRackTypes+1] = {
    { 0, 150, 200 },
    { 0, 150, 200 },
    { 0, 8,    16 }
};


const IloInt nbCardTypes = 4;

const IloInt nbRack = 5;
IloInt nbCards[nbCardTypes] = {10, 4, 2, 1};
```

One way to model this problem would be to define a class of objects representing the cards. Unfortunately, this model is highly symmetrical. With this model, each permutation of two cards of the same type gives a new “solution.” Those symmetries increase the size of the search space without really offering any more authentic solutions.

A better way to model the problem is to focus on modeling the racks and reducing symmetry. Since the features of racks and the maximal number of racks to use are fixed, you can assume that you have *n* racks, and you want to find for each one:

- whether it is used or not
- the cards it receives

- its correct type according to the number of plugged cards and the power it must provide

Rack types are identified by integers. The type zero indicates the unused racks. This convention avoids introducing one constrained variable per rack to indicate whether the rack is used or not.

You define the class `Rack`. Its instances have the following characteristics:

- `type`: integer, the type of the rack. It is an unknown so it is represented by a constrained integer variable.
- `price`: integer, the price of the rack. This value depends on the type of the rack. It is an unknown so it is represented by a constrained integer variable.

To avoid the symmetries introduced by the first model, you manipulate the number of cards of a given *type* plugged into a rack (rather than the cards plugged into a rack). This model suppresses the symmetries introduced by the first one and thus reduces the search space.

In this model, each rack has an array of counters. Each counter is associated with a card type. It counts the number of cards of this type plugged into this rack. The values of the counters are unknowns, so they are represented by constrained integer variables.

To represent those counters, you add a new data member `_counters` to the instances of the class `Rack`. For a given rack, this data member contains its array of card counters. It is defined like this:

```
class Rack {
  public:
    IloIntVar      _type;
    IloIntVar      _price;
    IloIntVarArray _counters;
```

The arrays of card counters associated with racks all have the same size. This size is equal to the number of card types, given by `nbCardTypes`.

Searching for a solution using this model consists of finding, for each rack, the values of its associated counters.

Representing the constraints

The type and the power of a given rack are related to each other. In fact, the maximal power provided by a rack depends on its type, so its type must be chosen according to the power this rack must provide. Solver allows you to easily express this dependence by writing the expression `rackPower(_type)`.

This interdependence is implemented through the use of an *element* constraint. The dependence between type and price, and type and number of slots is also implemented with such a constraint.

Implementing the constraints

This section presents the implementation of the constraints of this problem according to the second model. It uses the following notation:

- *C(a,ct)* is the number of cards of type *ct* plugged into rack *a*.
- *power(a)* is the maximal power provided by rack *a*.
- *numberOfSlots(a)* is the number of connection slots provided by rack *a*.
- *use(ct)* is the power used by a card of type *ct*.
- *Nct* is the number of card types.
- *Nrt* is the number of rack types.
- *Nr* is the maximal number of racks to use.
- *cards(ct)* is the given number of cards of type *ct* to plug in.

Constraint: Number of cards

The first constraint states that the number of cards plugged into a rack is less than or equal to its number of connection slots.

For each rack *a*, $\sum_{ct = 0}^{ct = Nct - 1} C(a,ct) \le numberOfSlots(a)$

Constraint: Power used

The power used by the cards plugged into rack *a* must be less than or equal to the maximal power it provides.

$$\sum_{ct = 0}^{ct = Nct - 1} C(a,ct)use\langle ct\rangle \le power(a)$$

Generic constraints and class constraints

Those two constraints—number of cards and power used—are implemented directly by the functions `IloSum` and `IloScalProd`. These are both *generic* constraints predefined in the Solver library. By generic, it is meant that a single constraint applies to a number of constrained variables simultaneously. Those two constraints are added by the constructor of

the class `Rack`. In other words, you are using those two generic constraints as *class constraints*. The constructor uses the following code:

```
Rack::Rack(IloModel model, IloIntArray cardPower) {
  IloEnv env = model.getEnv();
  _type = IloIntVar(env, 0, nbRackTypes);
  _counters = IloIntVarArray(env, nbCardTypes, 0, maxSlots);
  _price = IloIntVar(env, 0, 10000);
  IloIntArray prices(env, nbRackTypes + 1);
  IloIntArray rackPower (env, nbRackTypes + 1);
  IloIntArray rackSlot (env, nbRackTypes + 1);
  for (IloInt i=0; i<nbRackTypes + 1; i++) {
    prices[i] = Racks[price][i];
    rackPower[i] = Racks[power][i];
    rackSlot[i] = Racks[nbSlot][i];
  }
  model.add(_price ==  prices(_type));
  model.add(IloScalProd(_counters, cardPower) <= rackPower(_type));
  model.add(IloSum(_counters) <= rackSlot(_type));
}
```

In that code,

- `nbRackTypes` is the number of rack types. The types of racks are represented by integers between zero and this number. The type zero indicates the unused racks.
- `Racks` is a two-dimensional array of integers, with the first index accessing the price, power, and number of slots of each rack.

The construction `T[i] = v` (where `T` is an instance of `IloNumVarArray`, and `i` and `v` are instances of `IloNumVar`) constrains the `i`th element of `T` to be equal to `v`. The code of the `Rack` constructor also shows other ways to use this *element* constraint.

Constraint: Demand

To make sure that all cards of a given type are used, you state that the sum of all counters of this card type associated with the racks is equal to the given number of those cards:

For each card type *ct*, $\sum_{a=0}^{a=Nr-1} C(a,ct) = cards(ct)$

This constraint is implemented by the function `IloSum`, a predefined generic constraint that applies simultaneously to a number of constrained variables. It is added in the `main` function.

```
    // all cards must be plugged
    for(j = 0; j < nbCardTypes; j++){
      IloIntVarArray vars(env,nbRack);
      for(i = 0; i < nbRack; i++)
        vars[i] = racks[i]->_counters[j];
      model.add(IloSum(vars) == nbCards[j]);
    }
```

Redundant constraint: Eliminating more symmetry

This model still has a lot of symmetries: any permutation between racks gives a new "solution." You remove these symmetries by defining an *order* among the configurations of each rack. The idea is to state that the type of rack *i-1* is greater than the type of rack *i*. This order constraint removes some symmetries.

That redundant constraint is stated like this in the `main` function:

```
    // remove symmetries
    for(i = 1; i < nbRack; i++){
      model.add(racks[i]->_type >= racks[i-1]->_type);
      model.add(IloIfThen(env, racks[i-1]->_type == racks[i]->_type,
                               racks[i-1]->_counters[0] <= racks[i]-
>_counters[0]));
    }
```

Cost function

As mentioned earlier, the goal is to minimize the total cost of plugging cards, so the *cost* variable is constrained in the `main` function like this:

```
    // cost constraints
    IloIntVarArray prices(env, nbRack);
    for(i=0;i<nbRack;i++)
      prices[i] = racks[i]->_price;

    IloObjective objective = IloMinimize(env, IloSum(prices));
    model.add(objective);
```

Solve

To search for a solution, you instantiate the type of each rack using `IloInstantiate` and generate the counters of each rack using `IloGenerate`. Counters with the smallest domain are bound first. The function `Generate` searches for a solution.

```
    IloGoal goal = IloGoalTrue(env);
    for(i=0;i<nbRack;i++){
      goal = goal && IloInstantiate(env, racks[i]->_type);
      goal = goal && IloGenerate(env, racks[i]->_counters);
    }
```

This goal is used in the `main` function like this:

```
IloSolver solver(model);

if (solver.solve(goal)) {
  solver.out() << endl
               << "Found an " << solver.getStatus()<< " solution at cost "
               << solver.getValue(objective) << endl << endl;
  for(i = 0; i < nbRack; i++) {
    if (solver.getValue(racks[i]->_type) !=0) {
      solver.out() << "Rack " << i << endl
                   << "Type : " << solver.getValue(racks[i]->_type)
                   << endl << "Price : "
                   << solver.getValue(racks[i]->_price) << endl
                   << "Counters : ";
      for (j = 0; j < nbCardTypes; j++)
        solver.out() << solver.getValue(racks[i]->_counters[j]) << " ";
      solver.out() << endl << endl;
    }
  }
}
else
  solver.out() << "No solution" << endl;
```

Complete program

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

```
#include <ilsolver/ilosolverint.h>

ILOSTLBEGIN

const IloInt nbRackTypes = 2;
const IloInt maxSlots = 16;
enum {power, price, nbSlot};

IloInt Racks[3][nbRackTypes+1] = {
    { 0, 150, 200 },
    { 0, 150, 200 },
    { 0, 8,    16 }
};


const IloInt nbCardTypes = 4;

const IloInt nbRack = 5;
IloInt nbCards[nbCardTypes] = {10, 4, 2, 1};

class Rack {
  public:
    IloIntVar      _type;
    IloIntVar      _price;
```

```
    IloIntVarArray _counters;
  Rack(IloModel model, IloIntArray cardPower);
};

Rack::Rack(IloModel model, IloIntArray cardPower) {
  IloEnv env = model.getEnv();
  _type = IloIntVar(env, 0, nbRackTypes);
  _counters = IloIntVarArray(env, nbCardTypes, 0, maxSlots);
  _price = IloIntVar(env, 0, 10000);
  IloIntArray prices(env, nbRackTypes + 1);
  IloIntArray rackPower (env, nbRackTypes + 1);
  IloIntArray rackSlot (env, nbRackTypes + 1);
  for (IloInt i=0; i<nbRackTypes + 1; i++) {
    prices[i] = Racks[price][i];
    rackPower[i] = Racks[power][i];
    rackSlot[i] = Racks[nbSlot][i];
  }
  model.add(_price ==  prices(_type));
  model.add(IloScalProd(_counters, cardPower) <= rackPower(_type));
  model.add(IloSum(_counters) <= rackSlot(_type));
}

int main () {
  IloEnv env;
  try {
    IloModel model(env);

    IloIntArray cardPower(env, (int)nbCardTypes, 20, 40, 50, 75);
    Rack* racks[nbRack];
    IloInt i,j;
    for (i = 0; i < nbRack; i++)
      racks[i] = new Rack(model, cardPower);
    // all cards must be plugged
    for(j = 0; j < nbCardTypes; j++){
      IloIntVarArray vars(env,nbRack);
      for(i = 0; i < nbRack; i++)
        vars[i] = racks[i]->_counters[j];
      model.add(IloSum(vars) == nbCards[j]);
    }
    // remove symmetries
    for(i = 1; i < nbRack; i++){
      model.add(racks[i]->_type >= racks[i-1]->_type);
      model.add(IloIfThen(env, racks[i-1]->_type == racks[i]->_type,
                               racks[i-1]->_counters[0] <= racks[i]-
>_counters[0]));
    }
    // cost constraints
    IloIntVarArray prices(env, nbRack);
    for(i=0;i<nbRack;i++)
      prices[i] = racks[i]->_price;

    IloObjective objective = IloMinimize(env, IloSum(prices));
    model.add(objective);

    IloGoal goal = IloGoalTrue(env);
    for(i=0;i<nbRack;i++){
      goal = goal && IloInstantiate(env, racks[i]->_type);
      goal = goal && IloGenerate(env, racks[i]->_counters);
```

```
    }
    IloSolver solver(model);

    if (solver.solve(goal)) {
      solver.out() << endl
                   << "Found an " << solver.getStatus()<< " solution at cost "
                   << solver.getValue(objective) << endl << endl;
      for(i = 0; i < nbRack; i++) {
        if (solver.getValue(racks[i]->_type) !=0) {
          solver.out() << "Rack " << i << endl
                       << "Type : " << solver.getValue(racks[i]->_type)
                       << endl << "Price : "
                       << solver.getValue(racks[i]->_price) << endl
                       << "Counters : ";
          for (j = 0; j < nbCardTypes; j++)
            solver.out() << solver.getValue(racks[i]->_counters[j]) << " ";
          solver.out() << endl << endl;
        }
      }
    }
    else
      solver.out() << "No solution" << endl;
    solver.printInformation();
    for (i = 0; i < nbRack; ++i)
      delete racks[i];
  }
  catch (IloException& ex) {
    cout << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Results

Executing this program with those data produces the following output. The optimal solution is found very rapidly, but the proof of optimality takes 100 times longer.

```
Found an OPTIMAL solution at cost 550

Rack 2
Type : 1
Price : 150
Counters : 0 1 2 0

Rack 3
Type : 2
Price : 200
Counters : 0 3 0 1

Rack 4
Type : 2
Price : 200
Counters : 10 0 0 0

Number of fails              : 279
Number of choice points      : 285
Number of variables          : 31
Number of constraints        : 30
Reversible stack (bytes)     : 4044
Solver heap (bytes)          : 20124
Solver global heap (bytes)   : 4044
And stack (bytes)            : 4044
Or stack (bytes)             : 4044
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 11144
Total memory used (bytes)    : 51488
Running time since creation  : 0.19
```

C H A P T E R

11

Using Constrained Floating-Point Variables: Modeling Equations

In this lesson, you will learn how to:

- use constrained floating-point variables in expressions
- use predefined functions such as `IloExponent`, `IloPower`, and `IloLog`
- use the goals `IloDichotomize` and `IloGenerateBounds`
- use arrays of constrained floating-point variables

Solver allows you to use constrained floating-point *variables* using the class `IloNumVar`. Instances of the class `IloNumVar` have a domain represented by an *interval* of floating-point real numbers. The bounds of the domain are not represented by integers but by doubles defined by C++. Consequently, computations with constrained floating-point variables occur in double precision.

> ***Note:*** *This chapter is not presented using the three-stage method, as in previous chapters in this part. This subject matter lends itself more naturally to a conceptual introduction rather than procedural-based learning. However, when confronted with a problem using constrained floating-point variables, you should still use the Describe, Model, and Solve stages.*

Declaring floating-point variables

You must *declare* a constrained floating-point variable before you use it, just as you do for other constrained variables in Solver. To create a floating-point variable, you create an `IloNumVar` of type `Float`.

Here is a constructor for `IloNumVar`:

```
IloNumVar(const IloEnv env,
          IloNum lowerBound = 0.0,
          IloNum upperBound = IloInfinity,
          Type type = Float,
          const char* name = 0);
```

An instance of this class represents a numeric variable in a model. A numeric variable may be an integer variable, a Boolean variable, or a floating-point variable; that is, a numeric variable has a type, a value of the nested enumeration `IloNumVar::Type`, which can be `Bool`, `Int`, or `Float`. By default, its type is `Float`. It also has a lower and upper bound. A numeric variable cannot assume values less than its lower bound, nor greater than its upper bound.

> ***Note:*** *If you are looking for a class of variables that can assume only constrained integer values, consider the class* `IloIntVar`*. If you are looking for a class of binary decision variables that can assume only the values 0 (zero) or 1 (one), then consider the class* `IloBoolVar`*.*

Using floating-point variables in expressions

Concert Technology allows you to define expressions involving floating-point variables. For example, you can formulate constraints on floating-point expressions by using the operators:

- *addition +*
- *subtraction -*
- *multiplication **
- *division /*

> ***Note:*** *In the context of floating-point expressions, Concert Technology and Solver know how to rewrite certain expressions more simply. For example,* `x+x+y` *is treated as* `2*x+y`*. However, Solver is not a general purpose rewrite system, so an expression like* `x+y+x` *remains unchanged.*

You can then use these expression to formulate constraints:

- *equality* `==`
- *less than or equal to* `<=`
- *greater than or equal to* `>=`

The following example model and solves a simple equation by mixing constrained integer variables and constrained floating-point variables. The floating-point variables are modeled using the class `IloNumVar`. The member function `IloSolver::getFloatVar` returns the algorithmically constrained floating-point variable corresponding to the variable `x`. Here is the example:

```
#include <ilsolver/ilosolverfloat.h>

ILOSTLBEGIN

int main() {

  IloEnv env;
  try {
  IloModel model(env);

  IloNumVar x(env, -2, 2), y(env, -100, 100);
  model.add((x - 1)*(x + 2) == y );
  IloSolver solver(model);
  solver.propagate();
  cout << "y = " << solver.getFloatVar(y) << endl;
}
  catch (IloException& ex) {
          cerr << "Error:" << ex << endl;
  }
  env.end();
  return 0;
}
```

The output is:

```
y = [-12, 4]
```

Using predefined functions with floating-point variables

Concert Technology provides many predefined functions to use with floating-point variables, such as `IloPower` and `IloExponent`. To demonstrate their use, let's code the equation: $x + x^3 + e^x = 10$.

```
#include <ilsolver/ilosolverfloat.h>

ILOSTLBEGIN

int main(){
  IloEnv env;
  try {
    IloModel model(env);

    IloNumVar x (env, 1, 10);  // Declaring the variable

    // Adding the constraint to the model
    model.add(x + IloPower(x, 3L) + IloExponent(x) == 10.);

    IloSolver solver(model);

    if (solver.solve())
      solver.out() << "x = " << solver.getValue(x) << endl;
    solver.printInformation();
  }
  catch (IloException& ex) {
    cout << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

When executing this program, you should get the following output.

```
x = 1.55113
```

The complete program is available online in the `YourSolverHome/examples/src/equation.cpp` file.

Using IloDichotomize

The function `IloDichotomize` creates and returns a goal in a Concert Technology model. The goal tries to instantiate constrained floating-point variables. To do so, it recursively searches half the domain of each variable at a time. This function works on `IloNumVar` variables or `IloNumVarArray` arrays of variables. (The type can be either `Float` or `Int`.)

The following example computes the intersection between the folium of Descartes and an exponentially decaying function. It uses the goal returned by `IloDichotomize` to search for a solution.

For the model of the problem, you use two variables related by the following equations:

$x*x/y + y*y/x = 2$

$y = e^{-x}$

Here is the code:.

```
#include <ilsolver/ilosolverfloat.h>

ILOSTLBEGIN

int main() {
  IloEnv env;
  try {
    IloModel model(env);

    IloNumVar x(env, -1e30, 1e30);   // Declaring the variables
    IloNumVar y(env, -1e30, 1e30);

    model.add(x*x/y + y*y/x == 2.);  // Posting the constraints
    model.add(y == IloExponent(-x));

    IloSolver solver(model);
    solver.startNewSearch(IloDichotomize(env, x));

    while (solver.next()) {
      solver.out() << "x = " << solver.getFloatVar(x) << endl;
      solver.out() << "y = " << solver.getFloatVar(y) << endl << endl;
    }
    solver.printInformation();
    solver.endSearch();
  }
  catch (IloException& ex) {
    cout << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

The program finds two solutions:

```
x = [0.868418..0.868418]
y = [0.419615..0.419615]

x = [0.294563..0.294563]
y = [0.744857..0.744857]
```

The complete program is available online in the `YourSolverHome/examples/src/folium.cpp` file.

Using IloGenerateBounds

The function `IloGenerateBounds` creates and returns a goal. The goal efficiently reduces the domain of a floating-point variable by propagating any constraints on that variable more than usual. It checks whether the boundaries of the domain of the variable are consistent with all the constraints posted on the variable. If that is not the case, then it reduces an interval around the variable until the boundaries become consistent up to the precision indicated by a precision parameter. If the precision is small, the new domain computed by `IloGenerateBounds` will be smaller. However, the smaller the precision, the longer the computation will take. This function works on `IloNumVar` variables or `IloNumVarArray` arrays of variables. (The type can be either `Float` or `Int`.)

The following example uses the goal returned by `IloGenerateBounds` to search for a solution.

The problem is to find the roots of the following polynomial:

$(x + 1)(x + 2)\ldots(x + 20) + 2^{-23}x^{19}$

That example can be coded like this:

```
#include <ilsolver/ilosolverfloat.h>

ILOSTLBEGIN

int main () {
  IloEnv env;
  try {
    IloModel model(env);

    IloNumVar x(env, -1e10, 1e10);

    IloExpr y = x + 1;
    for(IloInt i = 2; i <= 20; i++)
      y = y * (x + i);

    model.add(y + IloPower(2, -23) * IloPower(x, 19) == 0);
    y.end();

    IloSolver solver(env);
    solver.setDefaultPrecision(1e-8);

    solver.out().precision(8);

    solver.extract(model);

    solver.startNewSearch(IloGenerateBounds(env, x, 1e-5) &&
                          IloDichotomize(env, x));

    while (solver.next())
      solver.out() << "x = " << solver.getFloatVar(x) << endl;
    solver.printInformation();
    solver.endSearch();
  }
  catch (IloException& ex) {
    cout << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

That program prints all the roots of the equation.

```
x = [-1..-1]
x = [-2..-2]
x = [-3..-3]
x = [-4..-4]
x = [-4.9999999..-4.9999999]
x = [-6.0000069..-6.0000069]
x = [-6.9996972..-6.9996972]
x = [-8.0072676..-8.0072676]
x = [-8.9172503..-8.9172502]
x = [-20.846908..-20.846908]
```

The complete program is available online in the `YourSolverHome/examples/src/wilkins.cpp` file.

Using arrays of floating-point variables

You may often find it useful to design a model of your problem that organizes the unknowns into arrays of variables. Concert Technology provides a class of arrays for numerical variables, `IloNumVarArray`.

Let's assume two constrained floating-point variables, x and y, and one constrained integer variable, k. These three variables use the following intervals as their domains:

$x \in [-5, 10^8]$

$y \in [0, 10^8]$

$k \in [-1000, 1000]$

Let's also assume the following system of equations must be solved:

$x^3 + 10x = y^x - 2^k$

$kx + 7.7y = 2.4$

$(k - 1)^{y+1} \leq 10$

$(log(y + 2x + 12.) \leq k + 5. \vee y \geq k^2) \Rightarrow (x \leq 0. \wedge y \leq 1)$

$x \leq 0 \Rightarrow k > 3$

The solution is:

$x = -1.285057857952$

$y = 0.9792508352872$

$k = 4$

The complete program follows:

```
#include <ilsolver/ilosolverfloat.h>

ILOSTLBEGIN

int main(){
  IloEnv env;
  try {
    IloModel model(env);

    IloNumVar x(env, -5, 1e8);
    IloNumVar y(env, 0, 1e8);
    IloIntVar k(env, -1000, 1000);

    model.add(IloPower(x,3) + 10*x == IloPower(y, x) - IloPower(2, k));
```

```
    model.add(k*x + 7.7*y == 2.4);
    model.add(IloPower(k-1, y+1) <= 10);
    model.add(IloIfThen(env, IloLog(y + 2*x + 12) <= k + 5 || y >= k*k, x <= 0
&& y <= 1));
    model.add(IloIfThen(env, x <= 0, k > 3));

    IloNumVarArray vars(env, 2, x, y);
    IloSolver solver(env);

    solver.out().precision(16);

    solver.extract(model);

    if (solver.solve(IloGenerateBounds(env, vars, .1))) {
      solver.out() << "x = " <<  solver.getFloatVar(x) << endl;
      solver.out() << "y = " <<  solver.getFloatVar(y) << endl;
      solver.out() << "k = " <<  solver.getIntVar(k) << endl;
    }
    solver.printInformation();
  }
  catch (IloException& ex) {
    cout << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Here's the output of the program.

```
x = [-1.285057857942743..-1.285057857928474]
y = [0.9792508352918879..0.979250835302613]
k = [4]
```

The complete program is available online in the `YourSolverHome/examples/src/narin.cpp` file.

Factored and canonical forms of expressions

There is no exact method to determine whether it is better to write a constrained expression using a factored or canonical form in order for the domains to be maximally reduced. Consequently, if you face a problem where this issue is important, you should experiment with both factored and canonical forms of constrained expressions to see which gives better domain reduction for your particular problem. The following sections demonstrate this concept.

Factored and canonical forms: Example 1

Suppose x, y, and z are three variables, and you want to express the following relation:

$z = (x - 1)*(y + 2)$

There are various ways to express the constraint. You can, for example, write the expression in a factored form like this:

```
#include <ilsolver/ilosolverfloat.h>

ILOSTLBEGIN

int main() {

  IloEnv env;
  try {
  IloModel model(env);

  IloNumVar x(env, -2, 2), y(env, 0, 10), z(env, -100, 100);
  model.add((x - 1)*(y + 2) == z);
  IloSolver solver(model);
  solver.propagate();
  cout <<  "z = " << solver.getFloatVar(z) << endl;

}
  catch (IloException& ex) {
          cerr << "Error:" << ex << endl;
  }
  env.end();
  return 0;
}
```

The output is:

```
z = [-36, 12]
```

You can write the same expression in a canonical algebraic form, like this:

```
#include <ilsolver/ilosolverfloat.h>

ILOSTLBEGIN

int main() {

  IloEnv env;
  try {
  IloModel model(env);

  IloNumVar x(env, -2, 2), y(env, 0, 10), z(env, -100, 100);
  model.add(z == x*y - y + 2*x - 2);
  IloSolver solver(model);
  solver.propagate();
  solver.out() << "z = [" << solver.getMin(z)
                          << ", " << solver.getMax(z)
                          << "]" << endl;
}
  catch (IloException& ex) {
          cerr << "Error:" << ex << endl;
  }
  env.end();
  return 0;
}
```

After propagation, the result is:

```
z = [-36, 22]
```

With the factored form, you can see that propagation has more severely reduced the domain of z, and in terms of performance, that is a good thing.

Because of that observation, you might think that it is always better to write constraints in a factored form. However, you will see a counter-example to that hasty generalization.

Factored and canonical forms: Example 2

Now suppose that *x* and *y* are two constrained variables, and you want to express the following relation: $y = (x - 1)*(x + 2)$. If you write it in a factored form, like this:

```
#include <ilsolver/ilosolverfloat.h>

ILOSTLBEGIN

int main() {

  IloEnv env;
  try {
  IloModel model(env);

  IloNumVar x(env, -2, 2), y(env, -100, 100);
  model.add((x - 1)*(x + 2) == y );
  IloSolver solver(model);
  solver.propagate();
  cout << "y = " << solver.getFloatVar(y) << endl;
}
  catch (IloException& ex) {
          cerr << "Error:" << ex << endl;
  }
  env.end();
  return 0;
}
```

The propagation reduces the domain of *y*:

```
y = [-12, 4]
```

However, if you write it this way:

```
#include <ilsolver/ilosolverfloat.h>

ILOSTLBEGIN

int main() {

  IloEnv env;
  try {
  IloModel model(env);

  IloNumVar x(env, -2, 2), y(env, -100, 100);
  model.add(x*x + x - 2 == y);
  IloSolver solver(model);
  solver.propagate();
  cout << "y = " << solver.getFloatVar(y) << endl;
}
  catch (IloException& ex) {
          cerr << "Error:" << ex << endl;
  }
  env.end();
  return 0;
}
```

Then, the domain of *y* is:

```
y = [-4, 4]
```

In short, there are situations where the canonical form of an arithmetic expression leads to greater domain reduction than does the factored form of the same expression.

Factored and canonical forms: Using IloGenerateBounds

`IloGenerateBounds` is a goal that performs efficient domain reduction, regardless of whether its arguments are factored or canonical form.

Let's look at an example, where the constraint is not written in a factored form but in canonical form, like this:

```
#include <ilsolver/ilosolverfloat.h>

ILOSTLBEGIN

int main() {

  IloEnv env;
  try {
  IloModel model(env);

  IloNumVar x(env, -2, 2), y(env, 0, 10), z(env, -100, 100);
  model.add(z == x*y - y + 2*x - 2);
  IloSolver solver(model);
  solver.propagate();
  solver.out() << "z = [" << solver.getMin(z)
                          << ", " << solver.getMax(z)
                          << "]" << endl;
}
  catch (IloException& ex) {
          cerr << "Error:" << ex << endl;
  }
  env.end();
  return 0;
}
```

In this case, the domain of z is:

```
z = [-36, 22]
```

If you call the function `IloGenerateBounds` on the variable z, the reduction is efficient anyway. In other words, in situations where the form of a floating-point expression (whether canonical or factored) is an issue, it's a good idea to exploit `IloGenerateBounds`.

Here's the example:

```
#include <ilsolver/ilosolverfloat.h>

ILOSTLBEGIN

int main() {

  IloEnv env;
  try {
  IloModel model(env);

  IloNumVar x(env, -2, 2), y(env, 0, 10), z(env, -100, 100);
  model.add(z == x*y - y + 2*x - 2);
  IloSolver solver(model);
  solver.solve(IloGenerateBounds(env, z, 1e-5));
  solver.out() << "z = [" << solver.getMin(z)
                          << ", " << solver.getMax(z)
                          << "]" << endl;
}
  catch (IloException& ex) {
          cerr << "Error:" << ex << endl;
  }
  env.end();
  return 0;
}
```

The output is:

```
z = [-36, 15.04]
```

IEEE 754 and Solver

There are some special considerations about floating-point arithmetic due to the fact that certain floating-point numbers cannot be represented exactly on any given hardware and software platform. The Institute of Electronics and Electrical Engineers (IEEE) has proposed a widely adopted standard (IEEE 754) for dealing with these variations in a consistent and reliable way. Solver conforms to the IEEE 754 floating-point representation.

Part III

More on Solving

This part consists of the following lessons:

- Chapter 12, *Setting Filter Levels: Coloring Graphs*
- Chapter 13, *Controlling the Search: Locating Warehouses*
- Chapter 14, *Limits and Problem Decomposition: Locating Warehouses*
- Chapter 15, *Searching for Optimal Solutions: Replanning Warehouses*
- Chapter 16, *Using Parallel Solver: Multithreaded Warehouse Location*

C H A P T E R

12

Setting Filter Levels: Coloring Graphs

In this lesson, you will learn how to:

- set filter levels on constraints
- time the search
- use `IloBasicLevel`, `IloMediumLevel`, `IloExtendedLevel`, and `IloSolver::setDefaultFilterLevel`

Describe

In this lesson, you will solve a graph coloring problem. Graph coloring problems may seem like puzzles, but they have many applications in industry, including scheduling, testing circuit boards, and assigning frequencies for radio and phone communications.

A graph is composed of *nodes*. Nodes that must be different colors are connected by lines, called *edges*. A *clique* is a set of nodes where every node is linked to every other node by an edge.

These concepts will become clearer if you re-examine the map coloring problem you solved in Chapter 2, *Modeling and Solving a Simple Problem: Map Coloring*. Map coloring problems can be represented as graph coloring problems. Each country on the map is a node.

Nodes (countries) that must be colored different colors are linked by lines or edges. See Figure 12.1.

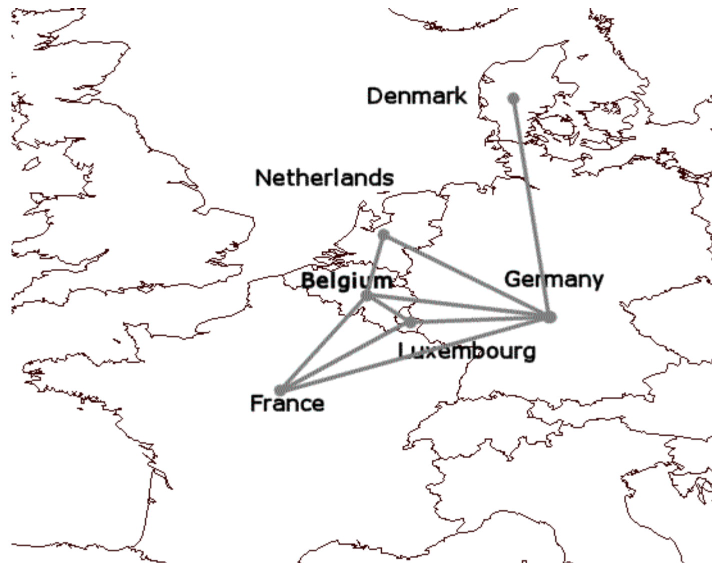

Figure 12.1 *Map coloring problem represented as a graph*

The nodes are Belgium, Denmark, France, Germany, Luxembourg, and the Netherlands. Each pair of neighboring countries is connected by an edge, indicating that these two countries cannot share the same color. There are many cliques. For example, France, Luxembourg, and Germany are all linked to each other by edges. They form a clique; Luxembourg is linked to Germany, Germany is linked to France, and France is linked to Luxembourg. However, France, Belgium, and the Netherlands do not represent a clique; France is linked to Belgium and Belgium is linked to the Netherlands, but France and the Netherlands are not linked to each other.

Two types of cliques, maximal cliques and the largest clique, are interesting for this lesson. A *maximal clique* is a clique such that no other clique contains it. For example, Germany and Denmark form a maximal clique, because no other clique contains Denmark. Another maximal clique is Germany, France, Luxembourg, and Belgium. Another is Germany, Belgium, and the Netherlands. The clique Germany, Luxembourg, and France is NOT a maximal clique because the clique Germany, Luxembourg, France, and Belgium contains it.

The *largest clique* is simply the clique containing the largest number of nodes. In this example, the largest clique is Germany, Luxembourg, France, and Belgium.

To convert this map coloring problem into a graph coloring problem, you can extract the nodes and edges from the map as shown in Figure 12.2. The problem is now to color the nodes in such a way that no pair of nodes connected by an edge are the same color.

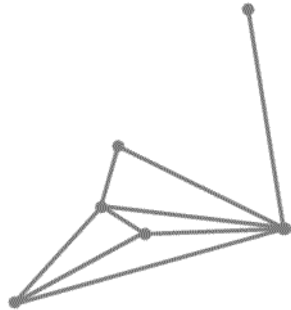

Figure 12.2 *Map coloring problem converted to a graph coloring problem*

You can solve this graph coloring problem by using the answers you found for the map coloring problem in Chapter 2, *Modeling and Solving a Simple Problem: Map Coloring*. Belgium is blue; Denmark is blue; France is white; Germany is yellow; Luxembourg is green; and the Netherlands is white.

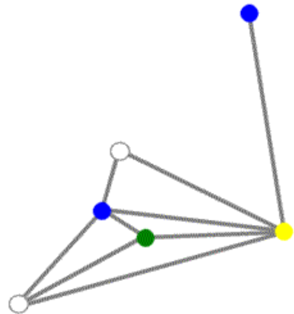

Figure 12.2 *Map coloring problem solved as graph coloring problem*

Many real world problems can be represented as graph coloring problems. In general, the graph represents items and the edges represent incompatibilities between items. The problem is to find a way to color the graph so that each incompatible pair is colored different colors, ideally using the fewest possible colors.

For example, the problem of assigning frequencies is essentially a graph coloring problem. In its simplest form, you have a set of frequencies and customers who use these frequencies, for example, in a cell phone. Customers who are located within a certain distance of each

other cannot share the same frequency—cannot be the same “color”—or there will be interference. Customers who are farther apart in distance can share the same frequency—can be the same “color.” The problem is to find a solution using the fewest number of frequencies—“colors”—while avoiding any interference.

Now that you understand the basic concepts in graph coloring problems, here is the problem description for this lesson. You will solve a graph coloring problem for a special kind of graph. This graph has $n*(n - 1)/2$ nodes, where n is an even number. Every node belongs to exactly two maximal cliques of size $n - 1$.

There is a theorem that states that the minimum number of colors needed to color a graph so that two nodes that are linked by an edge are different colors is greater than or equal to the size of the largest clique. In this problem, the largest clique is the same size as the maximal cliques of size $n - 1$. Therefore, the minimum number of colors needed to color this graph is greater than or equal to $n - 1$. You will try to color the graph with $n - 1$ colors and see if it is possible.

For example, if you have a graph with $n = 4$, there are 6 nodes or $4 * (4 - 1)/2$. Every node belongs to exactly two maximal cliques of size 3 or $4 - 1$.

Here are the maximal cliques:

- clique 0 = [0,1,2]
- clique 1 = [0,3,4]
- clique 2 = [1,3,5]
- clique 3 = [2,4,5]

Here is a diagram with the nodes and maximal cliques. The nodes are node 0, node 1, node 2, node 3, node 4, and node 5. The edges of clique 0 are blue; the edges of clique 1 are black; the edges of clique 2 are white; and the edges of clique 3 are green.

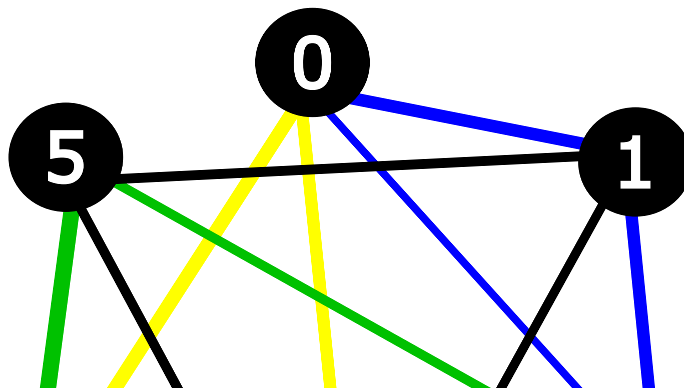

Figure 12.3 *Graph with nodes and maximal cliques*

As you can see in the diagram, the largest clique is also size 3 or $n - 1$. Therefore, the minimum number of colors needed to color this graph is greater than or equal to 3. You can, in fact, color this graph with 3 colors, as shown in the following diagram. None of the nodes connected by an edge share a color.

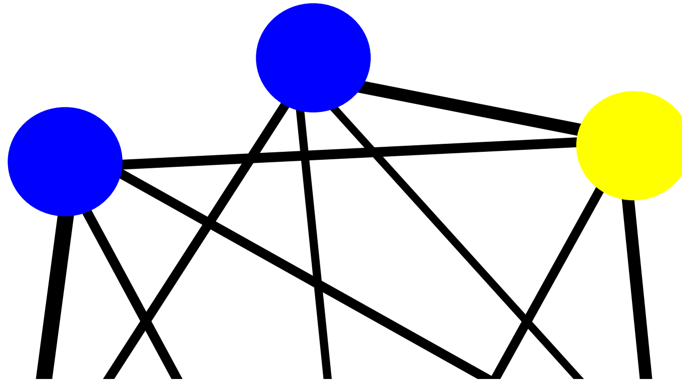

Figure 12.4 *Graph with colored nodes*

Of course, figuring out how to color a simple graph like this one is easy to do without using Solver. However, graphs representing scheduling or frequency assignment problems can have hundreds or thousands of nodes.

Step 1 Describe the problem

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

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

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

Discussion

What is the known information in this problem?

- the number of nodes in the graph
- which nodes belong to which maximal cliques

What are the decision variables or unknowns in this problem?

- what are the colors of the nodes

What are the constraints on these variables?

- in each maximal clique, all the nodes must be a different color

Since you know all the maximal cliques, you know all the nodes that are linked to each other. Therefore, if all the nodes in each maximal clique are different colors, in the graph as a whole, every pair of linked nodes will be different colors.

There is no objective in this problem. You will simply search for the first solution.

Model

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

Step 2 Open the example file

Open the example file `YourSolverHome/examples/src/tutorial/graph_partial.cpp` in your development environment.

You want to be able to run this program and test graphs of different sizes, so you create a function `testGraph` to run the test. It has one parameter, `nTest`, which is the size of the maximal cliques of the graph.

Step 3 Declare the testGraph function

Add the following code after the comment `//Declare the testGraph function`

```
void testGraph(IloInt nTest)
```

Next, you create the environment and the model. The parameter `nTest` is the size of the maximal clique, which is `n-1` in this problem. Since `n` must be an even number for this problem, you check if `nTest` is an even number. If it is not, you increment `nTest` by 1 to get the size `n` of the graph. This code is provided for you:

```
{
  IloEnv env;
  try {
    IloModel model(env);
    IloInt n = (nTest%2)?nTest+1:nTest;
```

Then you set the number of nodes, nbNodes, equal to n*(n-1)/2 and the number of colors, nbColors, equal to n-1. You declare i and j for use in loops. This code is provided for you:

```
IloInt nbNodes = n*(n-1)/2;
IloInt nbColors = n-1;
IloInt i, j;
```

Now you declare the variables, using an instance of IloIntVarArray. The array contains nbNodes variables. The variables in the array represent the unknown information—the color of each node. Each variable takes a value in the domain of integers from 0 to nbColors-1. These values represent the possible colors.

Step 4 Declare the decision variables

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

```
IloIntVarArray vars(env,nbNodes,0,nbColors-1);
for(i = 0; i < n-2; i++){
  model.add(vars[i] < vars[i+1]);
}
```

Now that you have created the variables, you create the maximal cliques using the function createClique. The function createClique is declared before the function testGraph. It is used to build the cliques. The following code is provided for you:

```
IloInt createClique(IloInt i, IloInt j, IloInt n){
  if (j >= i) return (i*n-i*(i+1)/2+j-i);
  else return createClique(j,i-1,n);
}
```

For each clique, you add the predefined constraint IloAllDiff. This constraint states that every node in that maximal clique must be a different color.

Step 5 Create the cliques and add the IloAllDiff constraint

Add the following code after the comment
//Create the cliques and add the IloAllDiff constraint

```
for(i = 0; i < n; i++){
  IloIntVarArray clique(env,n-1);
  for(j = 0; j < n-1; j++){
        clique[j] = vars[createClique(i,j, n)];
  }
  model.add(IloAllDiff(env,clique));
}
```

To demonstrate how the `createClique` function works, consider a graph with $n = 4$. There are 6 or $4 * (4 - 1)/2$ nodes. Every node belongs to exactly two maximal cliques of size 3 or $4 - 1$.

The "for" loops and the function `createClique` create these maximal cliques. The `createClique` function takes three parameters: `i`, `j`, and `n`. For `i=0` and `j=0` and `n=4`, the `createClique` function returns `(i*n - i*(i+1)/2 + j - i)` or $(0*4 - 0*(0 + 1)/2 + 0 - 0) = 0$. This corresponds to element `[0]` in the array `vars`, or node 0. In the second loop using `j`, `i=0` and `j=1` and `n=4` and the `createClique` function returns 1. This corresponds to element `[1]` in the array `vars`, or node 1. In the third loop using `j`, `i=0` and `j=2` and `n=4` and the `createClique` function returns 2. This corresponds to element `[2]` in the array `vars`, or node 2. After the loop using `j` has run three times, the code has created an array of variables `clique`. This array contains three elements: the variables representing nodes 0, 1, and 2. You use `IloAllDiff` to add the constraint to the model that these three variables must all be different from each other. This procedure is then followed for `i=1`, and so on. The following table shows the results of the "for" loops for a $n = 4$ graph:

	`j = 0`	`j = 1`	`j = 2`	
`i = 0`	0	1	2	clique 0
`i = 1`	0	3	4	clique 1
`i = 2`	1	3	5	clique 2
`i = 3`	2	4	5	clique 3

In other words, there are four maximal cliques:

- clique 0 = [0,1,2]
- clique 1 = [0,3,4]
- clique 2 = [1,3,5]
- clique 3 = [2,4,5]

For each clique, you use the predefined constraint `IloAllDiff` to state that every node in that clique must be a different color.

Solve

After you have declared the variables and added the constraints to the model, you are ready to search for a solution. In this lesson, you will not actually display the solution to the graph

coloring problem, but instead search to see if a solution is possible. You will learn how to time a search and how to use different filter levels for constraints.

First you create an instance of the class `IloSolver` to solve the problem expressed in the model. This code is already provided for you:

```
IloSolver solver(model);
```

Now, you create an instance of the class `IloTimer`. This class works like a stop watch and you will use it to time how long it takes to find a solution to the graph coloring problem using different filter levels.

Step 6 Create the timer

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

```
IloTimer timer(env);
```

Now, you set filter levels for constraints. Constraint propagation is, of course, more sophisticated than discussed in the introductory section “Solve” on page 32 in Chapter 1, *Constraint Programming with IBM ILOG Solver*. Each constraint is associated with a filtering algorithm. This filtering algorithm performs domain reductions based on the associated constraint—it will remove values from the current domains of variables that do not belong to a solution. Constraint propagation is the mechanism used to communicate the effects of these domain reductions.

For some types of constraints, you can set a filter level that specifies the type of filtering algorithm. So far, you have looked at two types of constraints. These are:

- binary constraints, such as the inequality constraint `x != y`, that affect two variables
- global constraints, such as `IloAllDiff` and `IloDistribute`, that can be defined as a set of non-global constraints

For all constraints except global constraints, the filtering algorithm is always the same. For global constraints, the filtering algorithm varies depending on the filter level. There are three filter levels for global constraints: `IloBasicLevel`, `IloMediumLevel`, and `IloExtendedLevel`.

To understand the difference between these filter levels, consider the `IloAllDiff` global constraint. You can view this constraint in two ways. It can be seen as a set of inequality `!=` constraints or as one global constraint. For example, consider a graph coloring problem with three nodes: *x*, *y*, and *z*. All the nodes must be colored a different color. Node *x* can be colored red or blue; node *y* can be colored red or blue; and node *z* can be colored red, blue, or yellow. If you set the filter level of `IloAllDiff` to `IloBasicLevel`, the filtering algorithm treats this global constraint as a set of inequality constraints. Looking at each set

of binary constraints individually, the filtering algorithm is not able to reduce the domains. The domains are not reduced after constraint propagation and appear as in Figure 12.5.

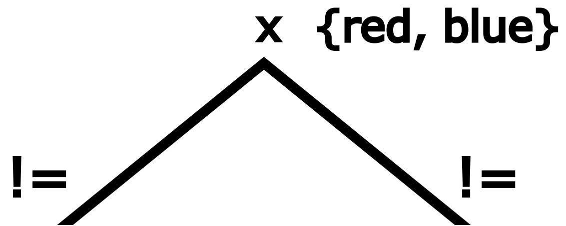

Figure 12.5 *IloBasicLevel filter level*

If you set the filter level of `IloAllDiff` to `IloExtendedLevel`, the filtering algorithm treats this global constraint as a truly global constraint. The filtering algorithm is not able to reduce the domains of *x* and *y*. However, the filtering algorithm can “realize” that between them, variables *x* and *y* must use both of the values red and blue. This leaves only the value of yellow available for variable *z*. The domain of *z* is reduced after constraint propagation and appears as in Figure 12.6.

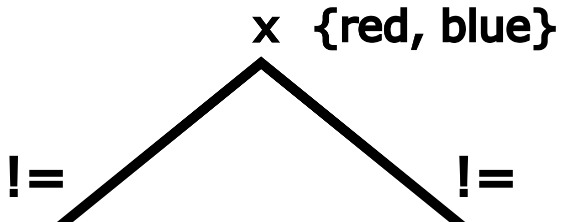

Figure 12.6 *IloExtendedLevel filter level*

Given that `IloExtendedLevel` is the most thorough filter level, why would you use any other filter level? There is a tradeoff in using `IloExtendedLevel`. In general, `IloExtendedLevel` takes longer. The filter level `IloBasicLevel` is less thorough, but faster. `IloMediumLevel` is a compromise between the two levels—faster than `IloExtendedLevel` and more thorough than `IloBasicLevel`. However, these are general rules and are not true for every situation. Depending on your application, different filter

levels may be appropriate. In this lesson, you run each test of the graph with the three filter levels and include performance output. You can analyze the different results.

> ***Note:*** *Filtering algorithms vary depending on the type of global constraint. In general, however, the filter levels provide a similar tradeoff between thoroughness and speed.* `IloExtendedLevel` *is the most thorough,* `IloBasicLevel` *the fastest, and* `IloMediumLevel` *is a compromise between the two.*

You use the member function `IloSolver::setDefaultFilterLevel` to set the filter levels for all global constraints. This function takes two parameters: the first specifies the type of global constraint, the second specifies the filter level.

Step 7 Set the default filter level

Add the following code after the comment `//Set the default filter level`

```
solver.setDefaultFilterLevel(IloAllDiffCt,IloExtendedLevel);
```

Now, you use the member function `IloTimer::start` to start the timer.

Step 8 Start the timer

Add the following code after the comment `//Start the timer`

```
timer.start();
```

You search for a solution using the predefined goal `IloGenerate`. Since you have chosen the parameter `IloChooseMinSizeInt`, Solver will first choose the unbound variable with the smallest domain and propagate the effects of this move, and so on. If a search move leads to a situation where a constraint is not satisfied, it is undone and Solver backtracks.

Step 9 Search for a solution

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

```
if (!solver.solve(IloGenerate(env,vars,IloChooseMinSizeInt)))
  solver.out() << "No solution" << endl;
```

When a solution is found, you display the size of the maximal cliques, the number of failures, and the elapsed time. This code is provided for you:

```
solver.out() << "IloExtendedLevel \t clique size:" << nTest;
solver.out() << "\t#fails:\t" << solver.getNumberOfFails() << "\t";
solver.out() << "cpu time: " << timer.getTime() << " s" << endl;
```

Now, you add the code to run the tests using the filter levels `IloMediumLevel` and `IloBasicLevel`. You will not have the thorough domain reductions that you had using `IloExtendedLevel`. Therefore, it is a good idea to add a redundant constraint to reduce the search space.

As you remember from Chapter 6, *Using the Distribute Constraint: Car Sequencing*, the `IloDistribute` constraint takes three arrays as parameters: `cards`, `values`, and `vars`. The constrained variables in the array `cards` are equal to the number of occurrences in the array `vars` of the values in the array `values`. If no `values` array is specified, this array is assumed to be an array of consecutive integers starting at 0. More precisely, for each `i`, `cards[i]` is equal to the number of occurrences of `values[i]` in the array `vars`.

Step 10 Add the redundant IloDistribute constraint

Add the following code after the comment
`//Add the redundant IloDistribute constraint`

```
    IloIntVarArray cards(env,nbColors,nbNodes/nbColors,
                         nbNodes/nbColors);
    IloConstraint distribute=IloDistribute(env,cards,vars);
    model.add(distribute);
```

Here is an example of how the distribute constraint would work for a graph of size $n = 3$, with 6 nodes and 3 colors. The array `cards` consists of 3 elements. The elements all have the same value—the upper and lower bounds are both `nbNodes/nbColors`, or $6/3 = 2$. The array `values` must be the same size as the array `cards`, or 3 elements. These are the first three consecutive integers starting at 0 and represent the possible colors. There are 6 variables that represent the color of each node.

In a graph of size $n = 3$, the array `cards` is `[2, 2, 2]`. The array `values` is `[0,1,2]`. Therefore, the value 0 (element `[0]` in the array `values`) must occur 2 (element `[0]` in the array `cards`) times in the array `vars`. Likewise, the value 1 must occur 2 times and the value 2 must occur 2 times. One possible set of values for the array `vars` that would satisfy this distribute constraint is `[0, 2, 1, 1, 2, 0]`. The value 0 occurs 2 times, the value 1 occurs 2 times, and the value 2 occurs 2 times.

> ***Note:** This is just an example of a set of values that would satisfy the distribute constraint. It is not necessarily a set of values that would represent a solution to the problem.*

Now you add the default filter level `IloMediumLevel`. This code is provided for you:

```
solver.setDefaultFilterLevel(IloAllDiffCt,IloMediumLevel);
timer.restart();
if (! solver.solve(IloGenerate(env,vars,IloChooseMinSizeInt)))
   solver.out() << "No solution" << endl;
solver.out() << "IloMediumLevel \t \t clique size:" << nTest;
solver.out() << "\t#fails:\t" << solver.getNumberOfFails() << "\t";
solver.out() << "cpu time: " <<  timer.getTime() << " s" << endl;
```

Finally, you run the tests with the default filter level `IloBasicLevel`. This code is provided for you:

```
solver.setDefaultFilterLevel(IloAllDiffCt,IloBasicLevel);
timer.restart();
if (! solver.solve(IloGenerate(env,vars,IloChooseMinSizeInt)))
  solver.out() << "No solution" << endl;
solver.out() << "IloBasicLevel \t \t clique size:" << nTest;
solver.out() << "\t#fails:\t" << solver.getNumberOfFails() << "\t";
solver.out() << "cpu time: " << timer.getTime() << " s" << endl;
```

In this example, you model and solve the problem in the function `testGraph`. The `main` function only serves as a way to input the clique size of the graph and to call the function `testGraph`. You run the first test on a graph with a maximal clique size of 27 (378 nodes). The second test is run on a graph with a maximal clique size of 29 (435 nodes). The third test is run on a graph with a maximal clique size of 31 (496 nodes). This code is provided for you:

```
int main(int argc, char** argv){
  IloInt b1=(argc>1)?atoi(argv[1]):27;
  IloInt b2=(argc>2)?atoi(argv[2]):b1+4;
  for(IloInt nTest = b1; nTest < b2+1; nTest+=2){
    testGraph(nTest);
  }
  return 0;
}
```

Step 11 Compile and run the program

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

```
--------------------------------------
IloExtendedLevel      clique size:27 #fails: 0       cpu time: 0.651 s
IloMediumLevel        clique size:27 #fails: 1       cpu time: 0.09 s
IloBasicLevel         clique size:27 #fails: 1       cpu time: 0.05 s

--------------------------------------
IloExtendedLevel      clique size:29 #fails: 4       cpu time: 0.541 s
IloMediumLevel        clique size:29 #fails: 7       cpu time: 0.11 s
IloBasicLevel         clique size:29 #fails: 23      cpu time: 0.07 s

--------------------------------------
IloExtendedLevel      clique size:31 #fails: 4       cpu time: 0.751 s
IloMediumLevel        clique size:31 #fails: 49      cpu time: 0.16 s
IloBasicLevel         clique size:31 #fails: 65      cpu time: 0.11 s
```

As you can see, Solver found a solution for the graph coloring problem with test runs of maximal clique size 27, 29, and 31. The number of failures and the time to solve varies depending on the filter level. At these clique sizes, the `IloExtendedLevel` filter level has fewer failures, but takes longer.

You can experiment by trying different clique sizes. Here is a test run with maximal cliques of size 51, 53, and 55. You can observe that the test run with a clique of size 55 follows the general pattern. There are fewer fails for `IloExtendedLevel`, but it takes the longest time to solve. However, the test runs of clique size 51 and 53 do not follow the normal pattern. This is because this graph coloring problem does not get more difficult in a purely linear fashion. Certain clique sizes, regardless of relative size, are very difficult to solve. Often these more difficult instances of the problem will solve faster with a filter level of `IloExtendedLevel` or `IloMediumLevel`. Here are the results:

```
--------------------------------------
IloExtendedLevel      clique size:51 #fails: 501     cpu time: 9.754 s
IloMediumLevel        clique size:51 #fails: 10906   cpu time: 13.299 s
IloBasicLevel         clique size:51 #fails: 24512   cpu time: 19.578 s

--------------------------------------
IloExtendedLevel      clique size:53 #fails: 2       cpu time: 10.896 s
IloMediumLevel        clique size:53 #fails: 368     cpu time: 1.121 s
IloBasicLevel         clique size:53 #fails: 13159   cpu time: 9.063 s

--------------------------------------
IloExtendedLevel      clique size:55 #fails: 2       cpu time: 13.35 s
IloMediumLevel        clique size:55 #fails: 27      cpu time: 1.221 s
IloBasicLevel         clique size:55 #fails: 94      cpu time: 0.892 s
```

The following test run on a clique of size 61—a particularly difficult instance of the problem—demonstrates the importance of trying different filter levels in your problem. You

will get the following solutions for the filter levels of `IloExtendedLevel` and `IloMediumLevel`. However, you will not have any response about the test using filter level `IloBasicLevel` for many hours:

```
--------------------------------------
IloExtendedLevel      clique size:61 #fails: 5       cpu time: 22.502 s
IloMediumLevel        clique size:61 #fails: 120408491       cpu time: 172926 s
```

The following test run on a clique size of 71 again demonstrates the general case. The filter level `IloBasicLevel` is fastest and the filter level `IloExtendedLevel` is the most thorough, with only 5 fails:

```
--------------------------------------
IloExtendedLevel      clique size:71 #fails: 5       cpu time: 49.471 s
IloMediumLevel        clique size:71 #fails: 586     cpu time: 5.217 s
IloBasicLevel         clique size:71 #fails: 1167    cpu time: 4.977 s
```

The complete program is listed in "Complete program" on page 223. You can also view it online in the file `YourSolverHome/examples/src/graph.cpp`.

Review exercises

For answers, see "Suggested answers" on page 221.

1. What are some real world examples of graph coloring problems?
2. What is the difference between the filter levels `IloExtendedLevel`, `IloMediumLevel`, and `IloBasicLevel`?
3. Remove the redundant `IloDistribute` constraint from the graph coloring program by commenting out the code you added in Step 10 on page 218. Run the tests on cliques of size 27, 29, and 31 and compare the results to those you obtained using the redundant constraint. How are they affected?

Suggested answers

Exercise 1

What are some real world examples of graph coloring problems?

Suggested Answer

These include scheduling problems, testing circuit boards, and assigning frequencies.

Exercise 2

What is the difference between the filter levels `IloExtendedLevel`, `IloMediumLevel`, and `IloBasicLevel`?

Suggested Answer

In general, the filter level `IloExtendedLevel` is the most thorough, but takes the longest. The filter level `IloBasicLevel` is less thorough, but faster. `IloMediumLevel` is a compromise between the two levels—faster than `IloExtendedLevel` and more thorough than `IloBasicLevel`. However, these are general rules and are not true for every situation.

Exercise 3

Remove the redundant `IloDistribute` constraint from the graph coloring program by commenting out the code you added in Step 10 on page 218. Run the tests on cliques of size 27, 29, and 31 and compare your results to those you obtained using the redundant constraint. How are they affected?

Suggested Answer

The results will vary depending on platform, machine, configuration, and so on.

Results

Results using the `IloDistribute` constraint:

```
--------------------------------------
IloExtendedLevel      clique size:27 #fails: 0       cpu time: 0.651 s
IloMediumLevel        clique size:27 #fails: 1       cpu time: 0.09 s
IloBasicLevel         clique size:27 #fails: 1       cpu time: 0.05 s

--------------------------------------
IloExtendedLevel      clique size:29 #fails: 4       cpu time: 0.541 s
IloMediumLevel        clique size:29 #fails: 7       cpu time: 0.11 s
IloBasicLevel         clique size:29 #fails: 23      cpu time: 0.07 s

--------------------------------------
IloExtendedLevel      clique size:31 #fails: 4       cpu time: 0.751 s
IloMediumLevel        clique size:31 #fails: 49      cpu time: 0.16 s
IloBasicLevel         clique size:31 #fails: 65      cpu time: 0.11 s
```

Results

When you do not use the `IloDistribute` constraint, there is no effect on the number of fails and little effect on the solution time for the `IloExtendedLevel` filter level for clique size 27. However, you will not have any response using filter levels `IloMediumLevel` and `IloBasicLevel` for many hours. As you can see, the `IloDistribute` constraint has a large impact on performance for these filter levels.

Results without the `IloDistribute` constraint:

```
-------------------------------------
IloExtendedLevel      clique size:27 #fails: 0       cpu time: 0.54 s
```

Complete program

The complete graph coloring program follows. The results will vary depending on platform, machine, configuration, and so on. You can also view it online in the file `YourSolverHome/examples/src/graph.cpp`.

```
#include <ilsolver/ilosolverint.h>
ILOSTLBEGIN

IloInt createClique(IloInt i, IloInt j, IloInt n){
  if (j >= i) return (i*n-i*(i+1)/2+j-i);
  else return createClique(j,i-1,n);
}

void testGraph(IloInt nTest)
{
  IloEnv env;
  try {
    IloModel model(env);
    IloInt n = (nTest%2)?nTest+1:nTest;
    env.out() << "-------------------------------------" << endl;
    IloInt nbNodes = n*(n-1)/2;
    IloInt nbColors = n-1;
    IloInt i, j;
    IloIntVarArray vars(env,nbNodes,0,nbColors-1);
    for(i = 0; i < n-2; i++){
      model.add(vars[i] < vars[i+1]);
    }
    for(i = 0; i < n; i++){
      IloIntVarArray clique(env,n-1);
      for(j = 0; j < n-1; j++){
            clique[j] = vars[createClique(i,j, n)];
      }
      model.add(IloAllDiff(env,clique));
    }
    IloSolver solver(model);
    IloTimer timer(env);
    // IloExtendedLevel
    solver.setDefaultFilterLevel(IloAllDiffCt,IloExtendedLevel);
    timer.start();
    if (!solver.solve(IloGenerate(env,vars,IloChooseMinSizeInt)))
      solver.out() << "No solution" << endl;
    solver.out() << "IloExtendedLevel \t clique size:" << nTest;
    solver.out() << "\t#fails:\t" << solver.getNumberOfFails() << "\t";
    solver.out() << "cpu time: " << timer.getTime() << " s" << endl;
    // Redundant constraint for other levels
```

```
    IloIntVarArray cards(env,nbColors,nbNodes/nbColors,
                        nbNodes/nbColors);
    IloConstraint distribute=IloDistribute(env,cards,vars);
    model.add(distribute);
    // IloMediumLevel
    solver.setDefaultFilterLevel(IloAllDiffCt,IloMediumLevel);
    timer.restart();
    if (! solver.solve(IloGenerate(env,vars,IloChooseMinSizeInt)))
       solver.out() << "No solution" << endl;
    solver.out() << "IloMediumLevel \t \t clique size:" << nTest;
    solver.out() << "\t#fails:\t" << solver.getNumberOfFails() << "\t";
    solver.out() << "cpu time: " <<  timer.getTime() << " s" << endl;
    // IloBasicLevel
    solver.setDefaultFilterLevel(IloAllDiffCt,IloBasicLevel);
    timer.restart();
    if (! solver.solve(IloGenerate(env,vars,IloChooseMinSizeInt)))
      solver.out() << "No solution" << endl;
    solver.out() << "IloBasicLevel \t \t clique size:" << nTest;
    solver.out() << "\t#fails:\t" << solver.getNumberOfFails() << "\t";
    solver.out() << "cpu time: " << timer.getTime() << " s" << endl;
    solver.out() << endl;
  }
  catch (IloException& ex) {
    cerr << "Error: " << ex << endl;
  }
  env.end();
}
int main(int argc, char** argv){
  IloInt b1=(argc>1)?atoi(argv[1]):27;
  IloInt b2=(argc>2)?atoi(argv[2]):b1+4;
  for(IloInt nTest = b1; nTest < b2+1; nTest+=2){
    testGraph(nTest);
  }
  return 0;
}
```

Results

Results: cliques of size 27, 29, and 31

```
--------------------------------------
IloExtendedLevel      clique size:27 #fails: 0       cpu time: 0.651 s
IloMediumLevel        clique size:27 #fails: 1       cpu time: 0.09 s
IloBasicLevel         clique size:27 #fails: 1       cpu time: 0.05 s

--------------------------------------
IloExtendedLevel      clique size:29 #fails: 4       cpu time: 0.541 s
IloMediumLevel        clique size:29 #fails: 7       cpu time: 0.11 s
IloBasicLevel         clique size:29 #fails: 23      cpu time: 0.07 s

--------------------------------------
IloExtendedLevel      clique size:31 #fails: 4       cpu time: 0.751 s
IloMediumLevel        clique size:31 #fails: 49      cpu time: 0.16 s
IloBasicLevel         clique size:31 #fails: 65      cpu time: 0.11 s
```

Results

Results: cliques of size 51, 53, and 55

```
--------------------------------------
IloExtendedLevel      clique size:51 #fails: 501     cpu time: 9.754 s
IloMediumLevel        clique size:51 #fails: 10906   cpu time: 13.299 s
IloBasicLevel         clique size:51 #fails: 24512   cpu time: 19.578 s

--------------------------------------
IloExtendedLevel      clique size:53 #fails: 2       cpu time: 10.896 s
IloMediumLevel        clique size:53 #fails: 368     cpu time: 1.121 s
IloBasicLevel         clique size:53 #fails: 13159   cpu time: 9.063 s

--------------------------------------
IloExtendedLevel      clique size:55 #fails: 2       cpu time: 13.35 s
IloMediumLevel        clique size:55 #fails: 27      cpu time: 1.221 s
IloBasicLevel         clique size:55 #fails: 94      cpu time: 0.892 s
```

Results

Results: cliques of size 61

```
--------------------------------------
IloExtendedLevel      clique size:61 #fails: 5       cpu time: 22.502 s
IloMediumLevel        clique size:61 #fails: 120408491       cpu time: 172926 s
```

Results

Results: cliques of size 71

```
--------------------------------------
IloExtendedLevel      clique size:71 #fails: 5       cpu time: 49.471 s
IloMediumLevel        clique size:71 #fails: 586     cpu time: 5.217 s
IloBasicLevel         clique size:71 #fails: 1167    cpu time: 4.977 s
```

C H A P T E R

13

Controlling the Search: Locating Warehouses

In this lesson, you will learn how to:

- use node evaluators
- use search selectors
- use Depth-First Search (DFS) and Slice-Based Search (SBS)
- use `IloInstantiate`, `IloNodeEvaluator`, `IloSBSEvaluator`, `IloApply`, `IloSearchSelector`, `IloMinimizeVar`, and `IloSelectSearch`

Describe

In this lesson, you will solve a logistics problem. A company has 10 stores. Each store must be supplied by one supplier warehouse. The company has five possible locations where it has property and can build a supplier warehouse: Bonn, Bordeaux, London, Paris, and Rome. The warehouse locations have different capacities. A warehouse built in Bonn or Paris could supply only one store. A warehouse built in London could supply two stores; a warehouse built in Rome could supply three stores; and a warehouse built in Bordeaux could supply four stores. The supply costs vary for each store, depending on which warehouse is the supplier. For example, a store that is located in Paris would have low supply costs if it

were supplied by a warehouse also in Paris. That same store would have much higher supply costs if it were supplied by the other warehouses.

In a real world problem, the cost of building a warehouse would vary depending on warehouse location. To keep this problem simple in order to focus on search strategies, the cost of building any warehouse is set to 30.

The problem is to find the most cost-effective solution to this problem, while making sure that each store is supplied by a warehouse. Table 13.1 gives the relative cost of supplying each existing store from each of the potential supplier warehouse sites.

Table 13.1 *Relative supply costs for stores*

	Bonn	Bordeaux	London	Paris	Rome
store 0	20	24	11	25	30
store 1	28	27	82	83	74
store 2	74	97	71	96	70
store 3	02	55	73	69	61
store 4	46	96	59	83	04
store 5	42	22	29	67	59
store 6	01	05	73	59	56
store 7	10	73	13	43	96
store 8	93	35	63	85	46
store 9	47	65	55	71	95

Step 1 Describe the problem

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

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

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

Discussion

What is the known information in this problem?

- there are 10 stores
- there are five potential supplier warehouse sites
- the cost of building the warehouses
- the relative cost of supplying each existing store from each of the potential warehouse site

What are the decision variables in this problem?

- which supplier warehouse should supply each store

What are the constraints on these variables?

- each warehouse can only supply a certain number of stores

What is the objective?

- to find the most cost-effective solution, one that takes into account both the costs of supplying individual stores and the costs of building warehouses

Model

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

Step 2 Open the example file

Open the example file `YourSolverHome/examples/src/tutorial/storesbs_partial.cpp` in your development environment.

First, you represent the warehouse locations using C++ enumerations to associate meaningful names with values. This code is provided for you:

```
const char* Suppliers[] = {"Bonn", "Bordeaux", "London", "Paris", "Rome"};
```

As usual, you declare an environment and a model. You declare `i` and `j` for use in loops. This code is provided for you:

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

In this lesson, you will input data from the file `YourSolverHome/examples/data/store.dat`. The following code is provided for you:

```
    const char* fileName;
    if ( argc != 2 ) {
      env.warning() << "usage: " << argv[0] << " <filename>" << endl;
      env.warning() << "Using default file" << endl;
      fileName = "../../../examples/data/store.dat";
    } else
      fileName = argv[1];

    ifstream file(fileName);
    if ( !file )
      throw FileError();
```

Here is the data from the file `YourSolverHome/examples/data/store.dat`:

```
30
[[20, 24, 11, 25, 30],
 [28, 27, 82, 83, 74],
 [74, 97, 71, 96, 70],
 [2,  55, 73, 69, 61],
 [46, 96, 59, 83, 4],
 [42, 22, 29, 67, 59],
 [1,  5,  73, 59, 56],
 [10, 73, 13, 43, 96],
 [93, 35, 63, 85, 46],
 [47, 65, 55, 71, 95]]
[1, 4, 2, 1, 3]
```

The first line of data, 30, is the cost associated with building a warehouse. The second part of the data is a matrix representing the relative cost of supplying each existing store from each of the potential warehouse sites. This is the information presented in Table 13.1 on page 228. The last line of data is the capacity of each potential supplier warehouse: Bonn 1, Bordeaux 4, London 2, Paris 1, and Rome 3.

You model the cost associated with building a supplier warehouse as an integer `buildingCost`. You model the matrix of relative supply costs as an instance of `IloIntArray2`, an array of arrays. You model the capacities of the potential warehouses as an instance of `IloIntArray`.

Step 3 Model the data

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

```
IloInt       buildingCost;
IloIntArray2 costMatrix(env);
IloIntArray  capacity(env);
```

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

Step 4 Input the data

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

```
file >> buildingCost >> costMatrix >> capacity;
```

You create variables to represent the number of stores, `nStores`, and the number of supplier warehouses, `nSuppliers`. The number of stores can be deduced from the size of the `costMatrix` array of arrays. The number of supplier warehouses can be deduced from the size of the `capacity` array. This allows you to easily extend the example by using another data file. This code is provided for you:

```
IloInt nStores    = costMatrix.getSize();
IloInt nSuppliers = capacity.getSize();
```

Now you declare the decision variables. The first array of decision variables represents the unknown information in this problem—which supplier warehouse should supply each store. To represent this information, you declare an array of variables `supplier`. The array `supplier` represents the supplier warehouse selected for each store. This array has `nStores` elements or, in this example, 10. The possible values for these variables represent the supplier warehouses. In this example, a value of 0 represents Bonn, a value of 1 represents Bordeaux, a value of 2 represents London, a value of 3 represents Paris, and a value of 4 represents Rome. The array of decision variables `supplier` will contain the solution to the problem, once it is solved.

Step 5 Declare the supplier decision variables

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

```
IloIntVarArray supplier(env, nStores, 0, nSuppliers-1);
```

You also declare three other arrays of decision variables that will be used in creating the constraints. The array of variables `cost` represents the relative cost of supplying each existing store from each of the potential warehouse site. It has `nStores` elements or, in this example, 10.

Step 6 Declare the cost decision variables

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

```
IloIntVarArray cost(env, nStores, 0, 99999);
```

The array of decision variables `open` represents whether a potential supplier warehouse site is open or not—whether the warehouse should be built. It has `nSuppliers` elements or, in this example, 5. There are two possible values for each variable. The value is 1 if the warehouse is open and 0 if the warehouse is not open.

Step 7 Declare the warehouse open decision variables

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

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

You create a decision variable to represent the total cost in the problem. This total cost is defined as a constraint later in this section. It reflects a balance between finding the lowest relative supply costs for each store and the cost of building more warehouses.

Step 8 Declare the totalCost decision variable

Add the following code after the comment
`//Declare the totalCost decision variable`

```
IloIntVar totalCost(env, 0, 999999);
```

Now, you add the constraints. The first constraint states that the cost associated with each store is equal to the relative cost given in the matrix for this store and the supplier that has been selected for it. Using a “for” loop, this constraint is added for each store. For example, if store 0 is supplied by the Rome warehouse, then `cost[0]` is equal to the value in `costMatrix` represented by store 0 and the supplier for store 0. The element `[i]` denotes the row and the element `(supplier[i])` denotes the column. In this case, the value at `[0]`— or row 0—and `supplier[0]`—or Rome, column 4—is 30. See Table 13.1 on page 228.

Step 9 Add the constraints on relative cost

Add the following code after the comment
`//Add the constraints on relative cost`

```
    for (i = 0; i < nStores; i++){
      model.add(cost[i] == costMatrix[i](supplier[i]));
```

Next, you add the constraint that states that the supplier warehouse used in the previous constraint must be open—in other words, the warehouse must be built if a store is going to be supplied by it. For example, if store 0 is supplied by the Rome warehouse, this constraint states that the value of the open variable associated with the Rome warehouse must be equal to 1. The Rome warehouse must be built.

Step 10 Add the constraints on open warehouses

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

```
      model.add(open(supplier[i])==1 );
    }
```

Next, you add the constraints relating to supplier warehouse capacity. To do this, you need to find a way to add the number of stores being supplied by a warehouse and make sure that this number does not exceed the capacity of the warehouse. You can use the Concert Technology function `IloSum` to do this. This function returns a numeric value representing the sum of the variables in an array. Here is a constructor for `IloSum`:

```
IloExpr IloSum(const IloNumVarArray vars);
```

To add the constraints on warehouse capacity, you use two "for" loops. For each supplier warehouse `j`, you create a temporary array, `temp`, representing whether a store is supplied by that supplier. The array `temp` has `nStores` elements or 10 in this example. For each store `k`, add a constraint that if the supplier of that store equals `j`, then `temp [k]` equals 1. If the store is not supplied by supplier `j`, then `temp[k]` equals 0. Then, use the function `IloSum` to sum up the variables in the array `temp`. This sum will equal the number of stores supplied by the supplier warehouse `j`. This sum is constrained to be less than or equal to the capacity of supplier warehouse `j`. You add these constraints for all the supplier warehouses, 5 in this example.

Step 11 Add the constraints on warehouse capacity

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

```
for (j = 0; j < nSuppliers; j++ ) {
  IloIntVarArray temp(env, nStores, 0, 1);
  for (IloInt k = 0; k < nStores; k++)
    model.add(temp[k] == (supplier[k] == j));
  model.add(IloSum(temp) <= capacity[j]);
}
```

Finally, you use the function `IloSum` to add the constraints on total cost to the model. There are two cost issues in this problem. First, you want to supply each store with the warehouse that is the most cost-efficient. However, you also want to minimize the number of open warehouses since it costs a certain amount, 30 in this example, to open an warehouse. Therefore, the total cost is constrained to be equal to the sum of the relative costs of supplying each store from its supplier warehouse plus the number of open warehouses multiplied by the cost to build each warehouse. No explicit objective is added to the model in the problem, since total cost is minimized during search using a search selector. For more information on using search selectors, see “Solve” on page 234.

Step 12 Add the constraint on total cost

Add the following code after the comment `//Add the constraint on total cost`

```
model.add(totalCost == IloSum(cost) + IloSum(open) * buildingCost);
```

Solve

In this lesson, you will learn how to control the search using node evaluators and search selectors. Goals are the mechanism by which Solver implements search strategies. Goals are a bit like building blocks—goals can be created by combining other goals and a function that returns a goal can take other goals as parameters. You will perform five steps to create a final goal:

- ◆ You use the functions `IloGenerate` and `IloInstantiate` to return three goals, one goal for each of the following decision variables: `cost`, `supplier`, and `totalCost`. Then, you create `combinedGoal` by joining these three goals with logical AND operators.
- ◆ Next, you create a node evaluator, `SBSNodeEvaluator`.

◆ Then, you use the function `IloApply`. This function takes `combinedGoal` and `SBSNodeEvaluator` as parameters and returns `SBSNodeEvaluatorGoal`.

◆ Then, you create a search selector, `minimizeSearchSelector`, using the `totalCost` decision variable. The objective, which is to minimize the total cost, is treated implicitly by the search selector.

◆ Finally, you use the function `IloSelectSearch`. This function takes `SBSNodeEvaluatorGoal` and `minimizeSearchSelector` as parameters and returns `finalGoal`.

The member function `IloSolver::solve` takes `finalGoal` as a parameter and searches for a solution. Figure 13.1 demonstrates this procedure.

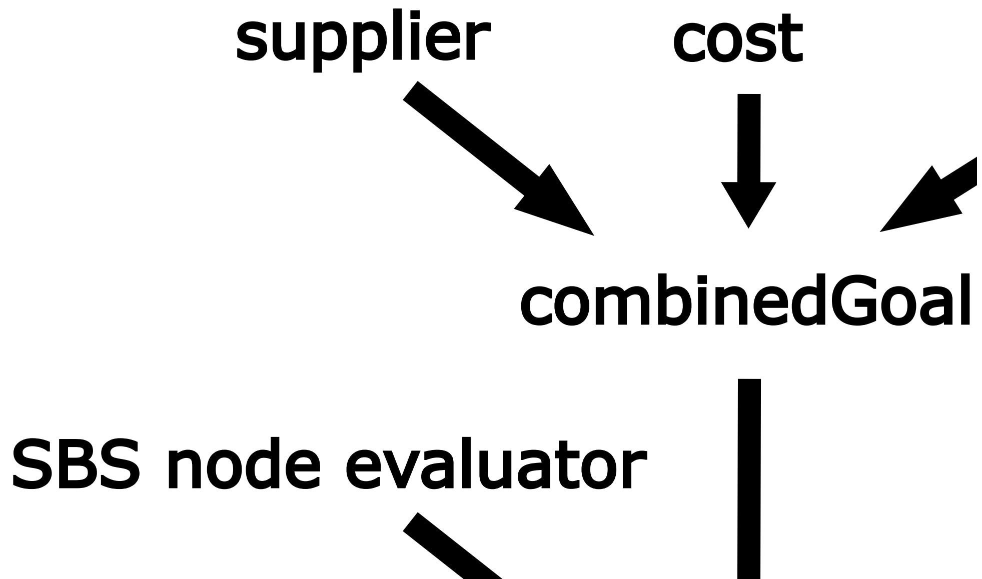

Figure 13.1 Creating a final goal

First, you create `combinedGoal` using logical AND operators. You are already familiar with the predefined goal `IloGenerate`. The goal `IloGenerate` takes an array of decision variables as a parameter and calls the function `IloInstantiate` on each variable in the array. The function `IloInstantiate` takes a decision variable and binds a value to it.

The first goal binds each decision variable in the array of variables `cost` using the principle of least regret, or minimizing regret. The `IloGenerate` parameter `IloChooseMaxRegretMin` is explained after the following step. The second goal binds each decision variable in the array of variables `supplier` in the default way, choosing the

first unbound variable. The third goal uses the function IloInstantiate to bind the decision variable totalCost.

Step 13 Create the combinedGoal

Add the following code after the comment //Create the combinedGoal

```
IloGoal combinedGoal = IloGenerate(env, cost, IloChooseMaxRegretMin) &&
                       IloGenerate(env, supplier) &&
                       IloInstantiate(env, totalCost);
```

As you may remember from Chapter 4, *Searching with Predefined Goals: Magic Square*, the predefined goal IloGenerate allows you to control a choice in the solution search by a parameter which determines which constrained variable Solver tries first. In this lesson, you use the parameter IloChooseMaxRegretMin for the array of variables cost. This parameter uses the principle of *least regret* to guide the search strategy. Regret is the difference between what would have been the best decision in a scenario and what was the actual decision. In the case of choosing a variable, Solver will select the value for the variable where the difference between the smallest possible value and the next smallest value is maximal.

The following table shows the relative supply costs for the stores. The best decision for each individual store is the warehouse with the lowest relative cost. The second best decision is the warehouse with the second lowest relative cost. The lowest and second lowest costs are in bold.

Table 13.2 *Relative supply costs for stores showing regret*

	Bonn	**Bordeaux**	**London**	**Paris**	**Rome**	**Difference between lowest cost and second lowest cost**
store0	**20**	24	**11**	25	30	9
store1	**28**	**27**	82	83	74	1
store2	74	97	**71**	96	**70**	1
store3	**02**	**55**	73	69	61	53
store4	**46**	96	59	83	**04**	42
store5	42	**22**	**29**	67	59	7
store6	**01**	**05**	73	59	56	4
store7	**10**	73	**13**	43	96	3

Table 13.2 Relative supply costs for stores showing regret

	Bonn	Bordeaux	London	Paris	Rome	Difference between lowest cost and second lowest cost
store8	93	**35**	63	85	**46**	11
store9	**47**	65	**55**	71	95	8

To illustrate the principle of least regret, examine one small decision. The supplier warehouse located in Bonn has a capacity of one store. Which store should be supplied by the warehouse in Bonn? Two likely candidates are store 3, with a relative cost of 2, and store 6, with a relative cost of 1. If you just looked at this information, you would likely make the decision to use the Bonn warehouse to supply store 6, since the relative cost is lower. However, choosing the Bonn warehouse to supply store 6 will force store 3 to be supplied by the Bordeaux warehouse, with the next lowest relative cost for store 3. You can also imagine what would happen if you choose the Bonn warehouse to supply store 3 and forced store 6 to be supplied by the Bordeaux warehouse, with the next lowest relative cost for store 6. Take a closer look at the two scenarios.

Table 13.3 Scenario A

Scenario A	Warehouse	Relative Cost
store6	Bonn	1
store3	Bordeaux	55
Total cost Scenario A		**56**

Table 13.4 Scenario B

Scenario B	Warehouse	Relative Cost
store6	Bordeaux	2
store3	Bonn	5
Total cost Scenario B		**7**

You can see that Scenario B is a much better choice for the total cost, even though the original choice to supply store 3 from the Bonn warehouse rather than store 6 is a slightly more costly decision than its inverse. You can use the principle of least regret to make just this kind of decision. If you had examined the difference between the lowest relative cost and the next lowest relative cost for store 3, you would see that it is 53. The difference for

store 6 is only 4. In deciding which store to supply from Bonn, choose the store with the largest difference. This decision will give you the lowest total cost and cause the least regret.

Now that you have created `combinedGoal`, you create a node evaluator and a search selector. Before you can do this, you need to learn more about search trees, nodes, and selectors. As you remember from Chapter 1, *Constraint Programming with IBM ILOG Solver*, the search tree is the part of the search space remaining after initial constraint propagation. In the search tree, Solver will use a search strategy to search for a solution. Chapter 1, *Constraint Programming with IBM ILOG Solver* contained many diagrams of the search tree to help you visualize the search process. These diagrams were simplified versions of the search tree. Solver actually performs a binarization of the search tree, representing every choice as a binary decision. Figure 13.2 is a binary representation of the search tree originally shown in Figure 1.2 on page 35. As you can see in Figure 13.2, every decision is a binary one. For example, if Solver selects the value 10 for variable *x*, there are now two choices. Solver can either try the value 5 for variable *y* or choose not to try value 5 for variable *y*. If Solver chooses not to try value 5 for variable *y*, Solver is again confronted with two choices. Solver can now either try the value 6 for variable *y* or the value 7 for variable *y*.

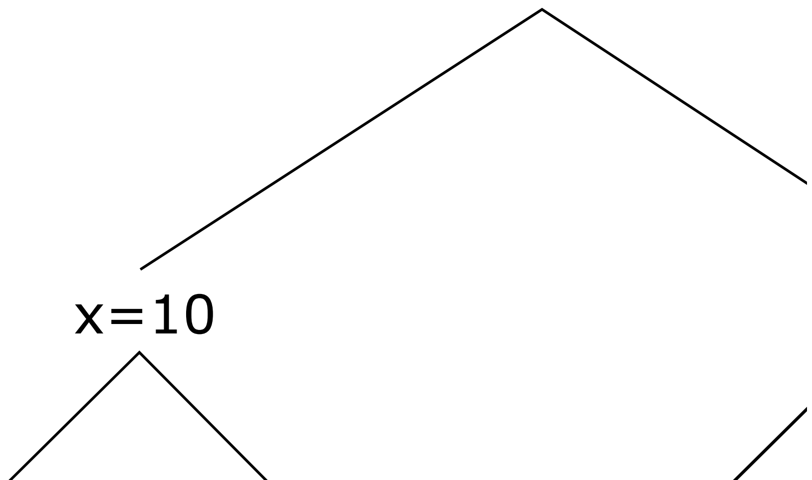

Figure 13.2 *Binarization of the search tree*

A search tree is composed of nodes connected by branches. The root of the tree is the starting point in the search for a solution; each branch descending from the root represents a decision in the search. At the beginning of the search, all the nodes in the tree are unexplored nodes. Solver starts the search at the root node. At this point, the root node is called an open node, since no decision has yet been made. There are two possible decisions to make at this open node—these two possible decisions are called leaves. Solver selects a leaf. The root node is now a closed node, since a decision has been made. The selected leaf is now a branch

and the node at the other end of this branch is now the open node. Two leaves come off this open node, representing two possible decisions. Tree traversal is the technique used to explore the search tree. As Solver traverses a search tree, it builds an active path. Each node, except for the terminal ones, has one or more child nodes and is called the parent of its child nodes. Removing branches from the search tree is pruning. *Node evaluators* guide how Solver traverses the search tree—which nodes are evaluated and in what order. *Search selectors* determine how Solver makes decisions at open nodes—a search selector guides the selection of a leaf.

These concepts are shown in Figure 13.3.

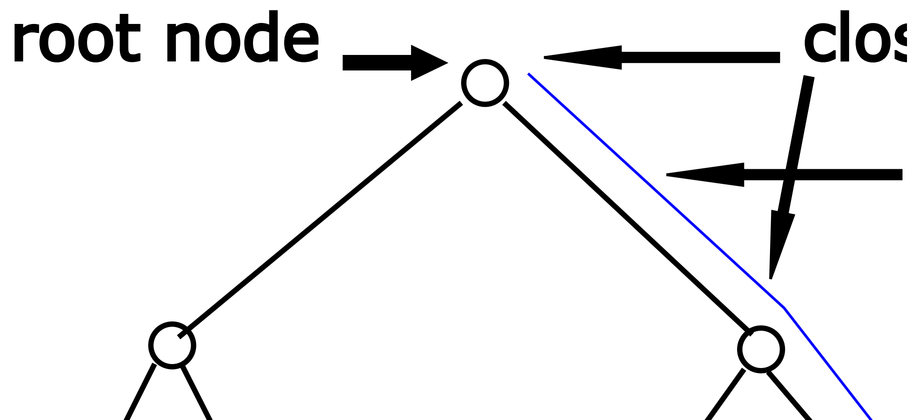

Figure 13.3 *Parts of the search tree*

As a default, Solver uses Depth-First Search (DFS) to traverse the search tree and control the order of node evaluation—Depth-First Search is the default node evaluator. Depth-First Search explores the search tree in a “left-to-right” fashion. It searches by selecting the left leaf at each open node until the end of the search space is reached. If it does not find a solution, it backtracks and tries again. Figure 13.4 shows tree traversal using Depth-First Search (DFS). The numbers at the bottom represent the order in which paths are created by the search strategy.

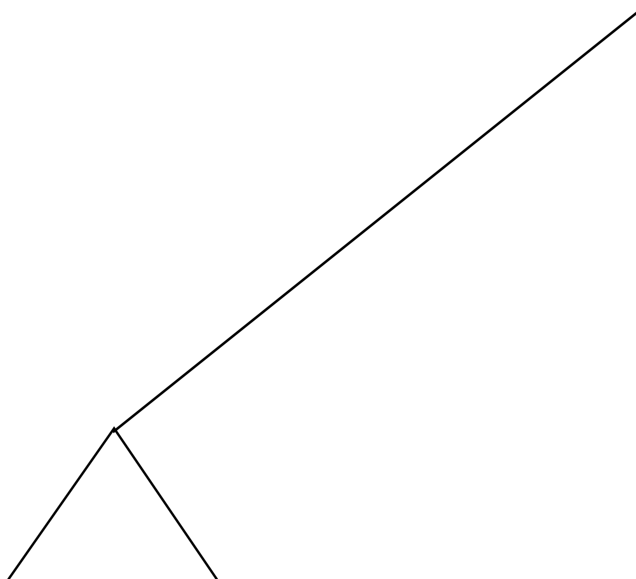

Figure 13.4 *Search tree using the Depth-First Search (DFS) strategy*

Depth-First Search is a straightforward approach to searching for a solution. However, sometimes you may have some knowledge about the search tree—you may know that certain parts of the search tree are more likely to contain a solution. In this case, you may want to change the order of node evaluation, since you think that certain nodes are more likely to lead to a solution. To do this, you can use a different node evaluator. There are several predefined functions in Solver that create and return node evaluators. To learn more about them, see Chapter 12. In this lesson, you will use the predefined function `IloSBSEvaluator` to create and return a node evaluator based on the principle of Slice-Based Search (SBS).

Usually, you have some kind of strategy to solve your problem. Therefore, you hope that Solver will not make many mistakes as it searches for a solution. In search trees, right moves are considered mistakes and left moves are considered correct decisions. So, if you think you have a good search strategy, it would make sense to look for a solution first among the nodes in paths that have fewer right moves. By searching first among the paths with fewer right moves you are saying you believe in your search strategy and you are trying to stick to it.

Slice-Based Search (SBS) does this. It looks first in the paths of the tree with fewer right moves. A right move in the path from the root of the search tree to the current node is known as a *discrepancy*. By looking first in paths with fewer right moves, Slice-Based Search cuts the search tree in slices.

The function `IloSBSEvaluator` creates and returns an instance of `IloNodeEvaluator` that implements Slice-Based Search. The first parameter is the environment. The second parameter is `step`. Solver will look first at paths that have a number of discrepancies, or right moves, less than or equal to `step`. The third parameter is `maxDiscrepancy`. If this parameter is used, Solver will discard nodes with a number of discrepancies greater than `maxDiscrepancy`. The default value for `maxDiscrepancy` is `IloIntMax`, which means that Solver will not discard any nodes based on this parameter. Here is a constructor for `IloSBSEvaluator`

```
IloNodeEvaluator IloSBSEvaluator(IloEnv env,
                                 IloInt step = 4,
                                 IloInt maxDiscrepancy = IloIntMax);
```

Figure 13.5 shows a search tree that has been traversed with a node evaluator using Slice-Based Search with a discrepancy of 1. The numbers in row 1 represent the order in which paths are evaluated. As you can see, Solver searches in slices as opposed to the “left-to-right” strategy of Depth-First Search (DFS). Solver first evaluates paths 1, 2, and 3. It then jumps to path 4 and then jumps again to path 5. It jumps back to path 6 and so on. The numbers in row 2 represent the number of discrepancies, or right moves, in each path. As you can see, Solver first search the paths with 0 or 1 discrepancy, then the paths with 2 discrepancies, then the paths with 3 discrepancies, and finally the path with 4 discrepancies.

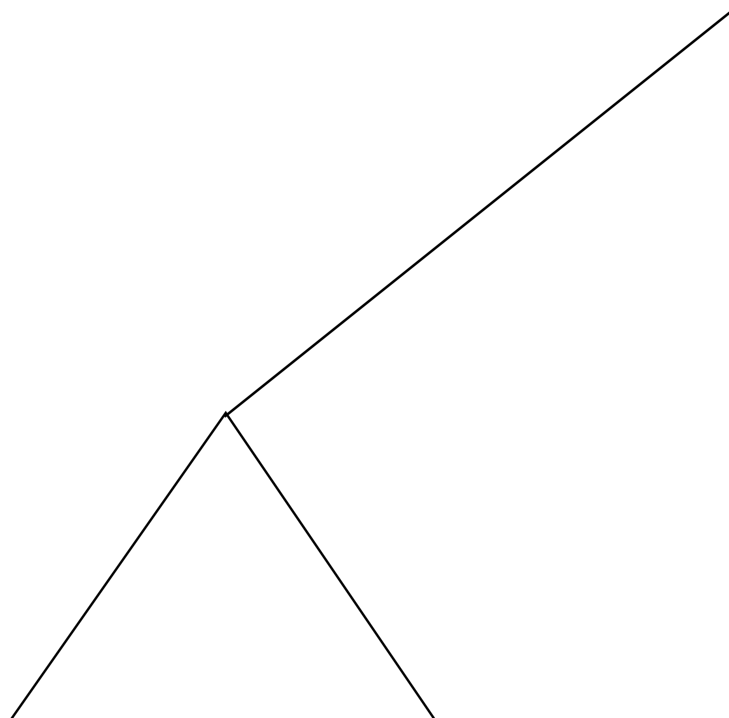

Figure 13.5 Search tree using the Slice-Based Search (SBS) strategy with a discrepancy of 1

Now that you understand the concepts behind node evaluators, you can create one. The node evaluator `SBSNodeEvaluator` will first search paths in the search tree with 4 or less discrepancies.

Step 14 Create the node evaluator

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

```
IloNodeEvaluator SBSNodeEvaluator = IloSBSEvaluator(env, 4);
```

Now that you have created the node evaluator, you need to apply it to `combinedGoal`, the goal you created in Step 13 on page 236. To do this you use the function `IloApply`. This function takes `combinedGoal` and `SBSNodeEvaluator` as parameters and returns

`SBSNodeEvaluatorGoal`. It applies the node evaluator `SBSNodeEvaluator` to the goal `combinedGoal`.

Step 15 Create the SBSNodeEvaluatorGoal

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

```
IloGoal SBSNodeEvaluatorGoal = IloApply(env, combinedGoal,
                                        SBSNodeEvaluator);
```

Now that you have created and applied a node evaluator, you create a search selector. This search selector will guide the search at each open node. In this problem, you want to select the leaf at each open node that will minimize the total cost of the solution. To do this, you use the function `IloMinimizeVar`. This function returns an instance of the class `IloSearchSelector`. It takes three parameters: the environment, the variable to be minimized, and a step. Here is a constructor:

```
IloSearchSelector IloMinimizeVar(IloEnv env,
                                 IloNumVar var,
                                 IloNum step = 0);
```

The search selector `IloMinimizeVar` does several things:

- It stores the leaf of the search tree that represents a solution with the optimal value of the variable `var`—in this example, the solution with the lowest `totalCost`. After the search tree has been completely explored, it reactivates this value. In other words, it stores the leaf representing the current best solution and gives the search the ability to go back to that leaf if it does not find a better solution.
- When a solution is found, it adds a constraint that `var` must be less than or equal to the solution minus `step`. For example, if the search selector (with `step = 0.1`) finds a solution with `totalCost` equal to 300, it adds a constraint that `totalCost` must be less than or equal to $300 - 0.1$ or 299.9 for the remainder of the search.

Step 16 Create the search selector

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

```
IloSearchSelector minimizeSearchSelector = IloMinimizeVar(env,
                                                          totalCost,
                                                          0.1);
```

Now that you have created the search selector `minimizeSearchSelector`, you need to apply it to `SBSNodeEvaluatorGoal`, the goal you created in Step 15 on page 243. To do this, you use the function `IloSelectSearch`. This function takes `SBSNodeEvaluatorGoal` and `minimizeSearchSelector` as parameters and returns

finalGoal. It applies the search selector minimizeSearchSelector to the goal SBSNodeEvaluatorGoal.

Step 17 Create the finalGoal

Add the following code after the comment //Create the finalGoal

```
IloGoal finalGoal = IloSelectSearch(env, SBSNodeEvaluatorGoal,
                                    minimizeSearchSelector);
```

Now, you create an instance of IloSolver to solve the problem expressed in the model. This code is provided for you:

```
IloSolver solver(model);
```

You now use the member function IloSolver::solve to search for a solution. In this lesson, you use finalGoal as a parameter of IloSolver::solve.

Step 18 Search for a solution

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

```
if (solver.solve(finalGoal))
```

The member functions and streams IloAlgorithm::out and IloSolver::getValue are used to display the solution. When you print the solution, you associate the value for the supplier warehouse variable supplier with the warehouse location using the array

Suppliers[], which is provided for you in the exercise code. The following code is provided for you:

```
      {
      solver.out() << "------------------------------" << endl;
      solver.out() << "Solution:  " << endl;
      for (i = 0; i < nStores; i ++)
        solver.out() << "Store  " << i << " " << "Warehouse:  "
                     << Suppliers[(IloInt)solver.getValue(supplier[i])]
                     << " " << endl;
        solver.out() << endl;
        for (j = 0; j < nStores ; j ++)
          solver.out() << "Store  " << j << " " << "Cost:  "
                       << solver.getValue(cost[j]) << " " << endl;
          solver.out() << endl;
          solver.out() << "Total cost:  " << solver.getValue(totalCost)
                                         << endl;
          solver.out() << "------------------------------" << endl;

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

Step 19 Compile and run the program

Compile and run the program. You should get the following results, though the information displayed by `IloSolver::printInformation` will vary depending on platform, machine, configuration, and so on:

```
Using default file
Cost to build a warehouse:
30
Relative costs for stores:
[[20, 24, 11, 25, 30], [28, 27, 82, 83, 74], [74, 97, 71, 96, 70], [2, 55, 73,
6
9, 61], [46, 96, 59, 83, 4], [42, 22, 29, 67, 59], [1, 5, 73, 59, 56], [10, 73,
13, 43, 96], [93, 35, 63, 85, 46], [47, 65, 55, 71, 95]]
Warehouse capacities:
[1, 4, 2, 1, 3]
------------------------------
Solution:
Store  0 Warehouse:  Rome
Store  1 Warehouse:  Bordeaux
Store  2 Warehouse:  Rome
Store  3 Warehouse:  Bonn
Store  4 Warehouse:  Rome
Store  5 Warehouse:  Bordeaux
Store  6 Warehouse:  Bordeaux
Store  7 Warehouse:  London
Store  8 Warehouse:  Bordeaux
Store  9 Warehouse:  London

Store  0 Cost:  30
Store  1 Cost:  27
Store  2 Cost:  70
Store  3 Cost:  2
Store  4 Cost:  4
Store  5 Cost:  22
Store  6 Cost:  5
Store  7 Cost:  13
Store  8 Cost:  35
Store  9 Cost:  55

Total cost:  383
------------------------------
Number of fails               : 25
Number of choice points       : 27
Number of variables           : 76
Number of constraints         : 76
Reversible stack (bytes)      : 12084
Solver heap (bytes)           : 52284
Solver global heap (bytes)    : 20124
And stack (bytes)             : 4044
Or stack (bytes)              : 4044
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 11160
Total memory used (bytes)     : 107784
Elapsed time since creation   : 0.02
```

As you can see, the solution uses four warehouse sites: Bonn, Bordeaux, London, and Rome. All stores are supplied by a supplier warehouse and the total cost is 383.

The complete program is listed in “Complete program” on page 251. You can also view it online in the file `YourSolverHome/examples/src/storesbs.cpp`.

Review exercises

For answers, see “Suggested answers” on page 247.

1. What is node evaluator and a search selector?
2. What is Slice-Based Search (SBS)?
3. Modify the program in this lesson to use the data file `YourSolverHome/examples/data/store_ex3.dat`. This data file contains information for 30 stores and 15 warehouses. The ten additional potential warehouse sites are: Munich, Barcelona, Prague, Dublin, Madrid, Lisbon, Berlin, Amsterdam, Brussels, and Milan.
4. Modify the file `YourSolverHome/examples/src/graph.cpp` to use Slice-Based Search (SBS) instead of the default Depth-First Search (DFS). Comment out the code using the `IloBasicLevel` filter level. Run the file on the clique set starting with size 61. As you remember, this is a particularly difficult instance of the problem to solve. Evaluate the solutions you get and compare them to the solution you found in Chapter 12, *Setting Filter Levels: Coloring Graphs*.

Suggested answers

Exercise 1

What are node evaluators and search selectors?

Suggested Answer

Node evaluators guide how Solver traverses the search tree—which nodes are evaluated and in what order.

Search selectors determine how Solver makes decisions at open nodes—a search selector guides the selection of a leaf.

Exercise 2

What is Slice-Based Search (SBS)?

Suggested Answer

Slice-Based Search is a search strategy that looks first in the paths of the tree with fewer right moves, or discrepancies. By looking first in paths with fewer right moves, Slice-Based Search cuts the search tree in slices.

Exercise 3

Modify the program in this lesson to use the data file `YourSolverHome/examples/data/store_ex3.dat`. This data file contains information for 30 stores and 15 warehouses. The ten additional potential warehouse sites are: Munich, Barcelona, Prague, Dublin, Madrid, Lisbon, Berlin, Amsterdam, Brussels, and Milan.

Suggested Answer

You can view the complete program online in the file `YourSolverHome/examples/src/storesbs_ex3.cpp`. You only need to change the following line of code to add the additional warehouse sites to the enumeration:

```
const char* Suppliers[] = {"Bonn", "Bordeaux", "London", "Paris", "Rome",
                           "Munich", "Barcelona", "Prague", "Dublin", "Madrid",
                           "Lisbon", "Berlin", "Amsterdam", "Brussels",
"Milan"};
```

You change the following line of code to use the new data file:

```
      fileName = "../../../examples/data/store_ex3.dat";
```

Results

The results will vary depending on platform, machine, configuration, and so on.

```
Using default file
Cost to build a warehouse:
30
Relative costs for stores:
[[20, 24, 11, 25, 30, 28, 27, 82, 83, 74,
2, 55, 73, 69, 61], [28, 27, 82, 83, 74, 93, 15, 63, 87, 46,
47, 57, 55, 71, 95], [74, 97, 71, 96, 70, 42, 8, 29, 67, 59,
93, 15, 63, 87, 46], [2, 55, 73, 69, 61, 1, 65, 73, 59, 56,
93, 35, 19, 85, 46], [46, 96, 59, 83, 4, 42, 22, 56, 67, 59,
11, 73, 15, 43, 96], [42, 22, 29, 67, 59, 1, 5, 57, 67, 56,
42, 8, 29, 67, 59], [1, 5, 73, 59, 56, 1, 5, 73, 59, 56,
42, 8, 29, 67, 59], [10, 73, 13, 43, 96, 42, 22, 56, 67, 59,
42, 56, 29, 56, 59], [93, 35, 63, 85, 46, 13, 22, 29, 66, 59,
65, 6, 73, 59, 56], [47, 65, 55, 71, 95, 42, 22, 29, 69, 59,
93, 25, 45, 85, 46],
[17, 19, 67, 2, 15, 93, 35, 19, 85, 46,
47, 57, 55, 71, 95], [26, 5, 78, 43, 19, 10, 15, 13, 9, 96,
68, 35, 62, 85, 46], [11, 73, 15, 43, 96, 42, 56, 29, 56, 59,
47, 57, 55, 71, 95], [93, 15, 63, 87, 46, 93, 55, 63, 92, 46,
47, 65, 55, 92, 95], [47, 65, 37, 71, 95, 47, 65, 45, 71, 95,
10, 73, 58, 43, 96], [42, 22, 29, 67, 49, 11, 45, 13, 43, 96,
```

```
42, 56, 29, 56, 59], [1, 5, 73, 59, 56, 47, 65, 45, 71, 95,
68, 35, 62, 85, 46], [10, 75, 13, 43, 62, 10, 15, 13, 9, 96,
93, 35, 19, 85, 46], [93, 25, 45, 85, 46, 65, 6, 73, 59, 56,
12, 5, 92, 59, 56], [47, 65, 55, 92, 95, 42, 22, 29, 69, 59,
47, 57, 55, 71, 95],
[42, 8, 29, 67, 59, 93, 35, 89, 85, 46,
47, 65, 56, 71, 95], [1, 5, 57, 67, 56, 42, 22, 56, 67, 5,
65, 6, 73, 59, 56], [10, 15, 13, 9, 96, 42, 22, 29, 69, 59,
47, 57, 55, 71, 95], [68, 35, 62, 85, 46, 93, 35, 19, 85, 46,
47, 57, 55, 71, 95], [47, 57, 55, 71, 95, 42, 22, 29, 69, 59,
93, 25, 45, 85, 46], [42, 22, 29, 69, 59, 1, 65, 73, 59, 56,
93, 35, 19, 85, 46], [1, 65, 73, 59, 56, 1, 65, 73, 59, 56,
93, 35, 19, 85, 46], [10, 73, 13, 83, 96, 11, 45, 13, 43, 96,
42, 56, 29, 56, 59], [93, 35, 19, 85, 46, 93, 35, 19, 85, 46,
47, 57, 55, 71, 95], [47, 65, 45, 71, 95, 28, 27, 82, 83, 74,
2, 55, 73, 69, 61]]
Warehouse capacities:
[5, 9, 12, 3, 4, 6, 8, 1, 6, 11,
3, 10, 8, 9, 3]

------------------------------
Solution:
Store  0 Warehouse:  London
Store  1 Warehouse:  Barcelona
Store  2 Warehouse:  Barcelona
Store  3 Warehouse:  Munich
Store  4 Warehouse:  Lisbon
Store  5 Warehouse:  Munich
Store  6 Warehouse:  Bordeaux
Store  7 Warehouse:  London
Store  8 Warehouse:  Munich
Store  9 Warehouse:  Barcelona
Store  10 Warehouse:  Bordeaux
Store  11 Warehouse:  Bordeaux
Store  12 Warehouse:  London
Store  13 Warehouse:  Bordeaux
Store  14 Warehouse:  Lisbon
Store  15 Warehouse:  Munich
Store  16 Warehouse:  Bordeaux
Store  17 Warehouse:  London
Store  18 Warehouse:  Barcelona
Store  19 Warehouse:  Barcelona
Store  20 Warehouse:  Bordeaux
Store  21 Warehouse:  Bordeaux
Store  22 Warehouse:  London
Store  23 Warehouse:  Bordeaux
Store  24 Warehouse:  Barcelona
Store  25 Warehouse:  Munich
Store  26 Warehouse:  Munich
Store  27 Warehouse:  London
Store  28 Warehouse:  London
Store  29 Warehouse:  Lisbon

Store  0 Cost:  11
Store  1 Cost:  15
Store  2 Cost:  8
Store  3 Cost:  1
Store  4 Cost:  11
```

```
Store  5 Cost:  1
Store  6 Cost:  5
Store  7 Cost:  13
Store  8 Cost:  13
Store  9 Cost:  22
Store  10 Cost:  19
Store  11 Cost:  5
Store  12 Cost:  15
Store  13 Cost:  15
Store  14 Cost:  10
Store  15 Cost:  11
Store  16 Cost:  5
Store  17 Cost:  13
Store  18 Cost:  6
Store  19 Cost:  22
Store  20 Cost:  8
Store  21 Cost:  5
Store  22 Cost:  13
Store  23 Cost:  35
Store  24 Cost:  22
Store  25 Cost:  1
Store  26 Cost:  1
Store  27 Cost:  13
Store  28 Cost:  19
Store  29 Cost:  2

Total cost:  490
------------------------------
Number of fails               : 68415
Number of choice points       : 65840
Number of variables           : 526
Number of constraints         : 526
Reversible stack (bytes)      : 104544
Solver heap (bytes)           : 361824
Solver global heap (bytes)    : 4584576
And stack (bytes)             : 4044
Or stack (bytes)              : 4044
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 13164
Total memory used (bytes)     : 5076240
Elapsed time since creation   : 86.835
```

Exercise 4

Modify the file `YourSolverHome/examples/src/graph.cpp` to use Slice-Based Search (SBS) instead of the default Depth-First Search (DFS). Comment out the code using the `IloBasicLevel` filter level. Run the file on the clique set starting with size 61. As you remember, this is a particularly difficult instance of the problem to solve. Evaluate the solutions you get and compare them to the solution you found in Chapter 12, *Setting Filter Levels: Coloring Graphs*.

Suggested Answer

The code that has changed from `graph.cpp` follows. You can also view the complete program online in the file `YourSolverHome/examples/src/storesbs_ex4.cpp`.

You change the goal as follows:

```
IloGoal goal = IloGenerate(env,vars,IloChooseMinSizeInt);
IloNodeEvaluator SBSNodeEvaluator = IloSBSEvaluator(env, 4);
IloGoal finalGoal = IloApply(env, goal, SBSNodeEvaluator);
```

You use this goal to solve the problem:

```
if (!solver.solve(finalGoal))
```

You should comment out the basic filter level run:

```
//IloBasicLevel
//solver.setDefaultFilterLevel(IloAllDiffCt,IloBasicLevel);
//timer.restart();
//if (! solver.solve(finalGoal))
//  solver.out() << "No solution" << endl;
//solver.out() << "IloBasicLevel \t \t clique size:" << nTest;
//solver.out() << "\t#fails:\t" << solver.getNumberOfFails() << "\t";
//solver.out() << "cpu time: " << timer.getTime() << " s" << endl;
//solver.out() << endl;
```

You should get the following results for cliques of size 61 using Slice-Based Search (SBS). The results will vary depending on platform, machine, configuration, and so on:

```
--------------------------------------
IloExtendedLevel      clique size:61 #fails: 5       cpu time: 22.812 s
IloMediumLevel        clique size:61 #fails: 4002    cpu time: 42.26 s
```

You can compare this to results for cliques of size 61 using the default Depth-First Search (DFS). For the `IloMediumLevel` filter level, the differences are dramatic, both in a greatly reduced number of fails and a greatly reduced solution time. The results will vary depending on platform, machine, configuration, and so on.

```
--------------------------------------
IloExtendedLevel      clique size:61 #fails: 5       cpu time: 22.502 s
IloMediumLevel        clique size:61 #fails: 120408491       cpu time: 172926 s
```

Complete program

The complete locating warehouses program follows. You can also view it online in the file `YourSolverHome/examples/src/storesbs.cpp`.

```
#include <ilsolver/ilosolverint.h>
ILOSTLBEGIN

class FileError : public IloException {
public:
  FileError() : IloException("Cannot open data file") {}
};

const char* Suppliers[] = {"Bonn", "Bordeaux", "London", "Paris", "Rome"};
int main(int argc, char** argv){
  IloEnv env;
  try {
    IloModel model(env);
    IloInt i, j;
    const char* fileName;
    if ( argc != 2 ) {
      env.warning() << "usage: " << argv[0] << " <filename>" << endl;
      env.warning() << "Using default file" << endl;
      fileName = "../../../examples/data/store.dat";
    } else
      fileName = argv[1];

    ifstream file(fileName);
    if ( !file )
      throw FileError();
    // model data
    IloInt       buildingCost;
    IloIntArray2 costMatrix(env);
    IloIntArray  capacity(env);

    file >> buildingCost >> costMatrix >> capacity;
    IloInt nStores    = costMatrix.getSize();
    IloInt nSuppliers = capacity.getSize();
    env.out() << "Cost to build a warehouse:  " << endl;
    env.out() << buildingCost << endl;
    env.out() << "Relative costs for stores:  " << endl;
    env.out() << costMatrix        << endl;
    env.out() << "Warehouse capacities:  " << endl;
    env.out() << capacity     << endl;

    // build model
    IloIntVarArray supplier(env, nStores, 0, nSuppliers-1);
    IloIntVarArray cost(env, nStores, 0, 99999);
    IloIntVarArray open(env, nSuppliers, 0,1);
    for (i = 0; i < nStores; i++){
      model.add(cost[i] == costMatrix[i](supplier[i]));
      model.add(open(supplier[i])==1 );
    }
    for (j = 0; j < nSuppliers; j++ ) {
      IloIntVarArray temp(env, nStores, 0, 1);
      for (IloInt k = 0; k < nStores; k++)
        model.add(temp[k] == (supplier[k] == j));
      model.add(IloSum(temp) <= capacity[j]);
    }
    IloIntVar totalCost(env, 0, 999999);
    model.add(totalCost == IloSum(cost) + IloSum(open) * buildingCost);
    IloGoal combinedGoal = IloGenerate(env, cost, IloChooseMaxRegretMin) &&
                           IloGenerate(env, supplier) &&
```

```
                        IloInstantiate(env, totalCost);
    IloNodeEvaluator SBSNodeEvaluator = IloSBSEvaluator(env, 4);
    IloGoal SBSNodeEvaluatorGoal = IloApply(env, combinedGoal,
                                            SBSNodeEvaluator);
    IloSearchSelector minimizeSearchSelector = IloMinimizeVar(env,
                                                              totalCost,
                                                              0.1);
    IloGoal finalGoal = IloSelectSearch(env, SBSNodeEvaluatorGoal,
                                        minimizeSearchSelector);
    // solve model
    IloSolver solver(model);

    if (solver.solve(finalGoal))
      {
      solver.out() << "------------------------------" << endl;
      solver.out() << "Solution:  " << endl;
      for (i = 0; i < nStores; i ++)
        solver.out() << "Store  " << i << " " << "Warehouse:  "
                     << Suppliers[(IloInt)solver.getValue(supplier[i])]
                     << " " << endl;
        solver.out() << endl;
        for (j = 0; j < nStores ; j ++)
          solver.out() << "Store  " << j << " " << "Cost:  "
                       << solver.getValue(cost[j]) << " " << endl;
          solver.out() << endl;
          solver.out() << "Total cost:  " << solver.getValue(totalCost)
                                         << endl;
          solver.out() << "------------------------------" << endl;

    }
    else solver.out() << "No solution" << endl;
    solver.printInformation();
  }
  catch (IloException& ex) {
    cout << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Results

The results will vary depending on platform, machine, configuration, and so on.

```
Using default file
Cost to build a warehouse:
30
Relative costs for stores:
[[20, 24, 11, 25, 30], [28, 27, 82, 83, 74], [74, 97, 71, 96, 70], [2, 55, 73,
6
9, 61], [46, 96, 59, 83, 4], [42, 22, 29, 67, 59], [1, 5, 73, 59, 56], [10, 73,
13, 43, 96], [93, 35, 63, 85, 46], [47, 65, 55, 71, 95]]
Warehouse capacities:
[1, 4, 2, 1, 3]
------------------------------
Solution:
Store  0 Warehouse:  Rome
```

```
Store  1 Warehouse:  Bordeaux
Store  2 Warehouse:  Rome
Store  3 Warehouse:  Bonn
Store  4 Warehouse:  Rome
Store  5 Warehouse:  Bordeaux
Store  6 Warehouse:  Bordeaux
Store  7 Warehouse:  London
Store  8 Warehouse:  Bordeaux
Store  9 Warehouse:  London

Store  0 Cost:  30
Store  1 Cost:  27
Store  2 Cost:  70
Store  3 Cost:  2
Store  4 Cost:  4
Store  5 Cost:  22
Store  6 Cost:  5
Store  7 Cost:  13
Store  8 Cost:  35
Store  9 Cost:  55

Total cost:  383
------------------------------
Number of fails               : 25
Number of choice points       : 27
Number of variables           : 76
Number of constraints         : 76
Reversible stack (bytes)      : 12084
Solver heap (bytes)           : 52284
Solver global heap (bytes)    : 20124
And stack (bytes)             : 4044
Or stack (bytes)              : 4044
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 11160
Total memory used (bytes)     : 107784
Elapsed time since creation   : 0.02
```

C H A P T E R

14

Limits and Problem Decomposition: Locating Warehouses

In this lesson, you will learn how to:

- use search limits
- decompose problems
- use Depth-Bounded Discrepancy Search (DDS)
- use `IloLimitSearch`, `IloDDSEvaluator`, and `IloFailLimit`

Describe

In this lesson, you will solve the same problem you solved in Chapter 13, *Controlling the Search: Locating Warehouses*: a logistics problem involving locating supplier warehouses for the most cost-efficient solution to supplying stores. For the description of this problem, see the section “Describe” on page 227.

Model

Once you have written a description, you use Concert Technology classes to model it.

Step 1 Open the example file

Open the example file `YourSolverHome/examples/src/tutorial/storecpx_partial.cpp` in your development environment.

In this lesson, you model this problem in the same way you modeled the problem in Chapter 13, *Controlling the Search: Locating Warehouses*. See "Model" on page 229.

Solve

In this lesson, you will learn how to decompose a problem and how to set limits on the search. Using this approach, Solver will not search the entire search tree and it may miss an optimal solution. However, Solver will explore a more promising part of the search tree more quickly. You will also learn how to use another search strategy—Depth-Bounded Discrepancy Search (DDS).

In Chapter 13, *Controlling the Search: Locating Warehouses*, you created goals for three decision variables: `cost`, `supplier`, and `totalCost`. These three goals were combined into one goal. You then applied a node evaluator to this combined goal. In the search, you used a search selector created from the `totalCost` variable.

In this lesson, you do not combine the goals at first. You apply node evaluators to the goals individually. You add a search limit to the first goal you execute. In the first phase, you search the search tree with a goal created from the `cost` variable. You prune the search tree based on a search limit associated with this goal. In the second phase, you search the remaining part of the search tree with a goal created from the `supplier` variable. In both phases of the search, you use a search selector created from the `totalCost` variable.

You will perform eight steps to create a final goal to be used in search:

- ◆ You use the function `IloGenerate` to return `goal1` for the decision variable `cost`.
- ◆ You use the function `IloApply`. This function applies the node evaluator `IloDDSEvaluator` to `goal1` and returns `applygoal1`.
- ◆ You use the function `IloLimitSearch`. This function places a fail limit on `applygoal1` and returns `limitgoal1`.
- ◆ You use the function `IloGenerate` to return `goal2` for the decision variable `supplier`.
- ◆ You use the function `IloApply`. This function applies the node evaluator `IloSBSEvaluator` to `goal2` and returns `applygoal2`.
- ◆ Then, you create `combinedGoal` by joining the following goals with logical AND operators: `limitgoal1`, `applygoal2`, and a goal returned by the function `IloInstantiate` for the variable `totalCost`. Goals that are combined with logical AND operators are executed in the order they are declared—from left to right.

- Next, you create a search selector, `minimizeSearchSelector`, from the `totalCost` variable. The objective, which is to minimize the total cost, is treated implicitly by the search selector.
- Finally, you use the function `IloSelectSearch`. This function takes `combinedGoal` and `minimizeSearchSelector` as parameters and returns `finalGoal`.

The member function `IloSolver::solve` takes `finalGoal` as a parameter and searches for a solution in two phases. In the first phase, Solver searches using the goal `limitgoal1`. The search tree is pruned by this goal. In the second phase, the remaining part of the search tree is searched using `applygoal2` until a solution is found. The search selector `minimizeSearchSelector` is used in both phases of the search. Figure 14.1 demonstrates this procedure.

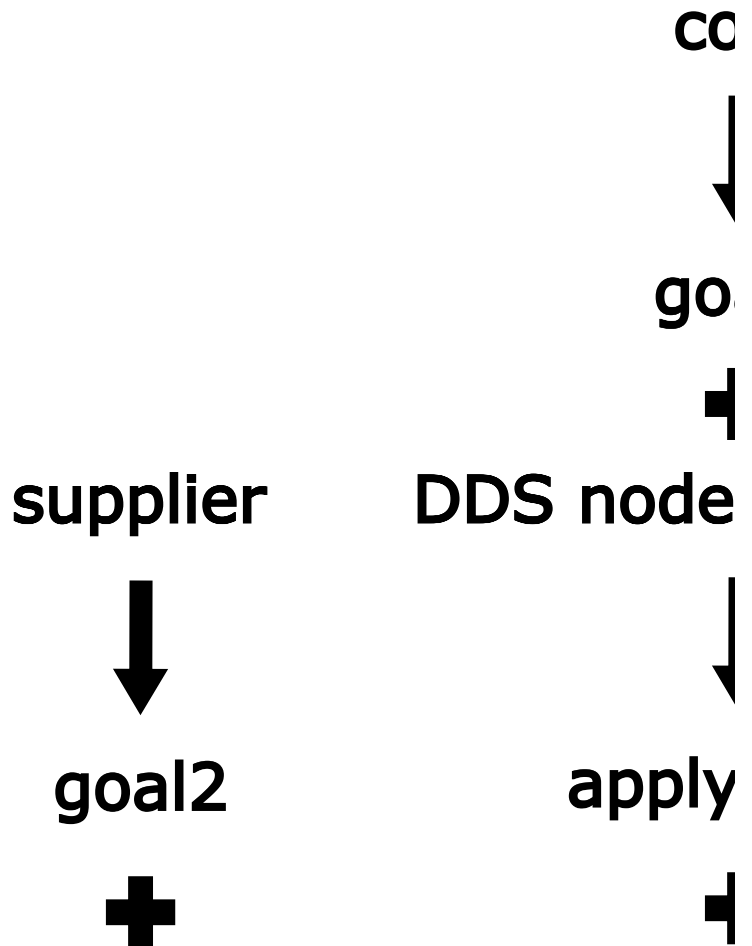

Figure 14.1 *Creating a final goal*

To start, you use the function `IloGenerate` to return `goal1` for the decision variable `cost`. You use the parameter `IloChooseMaxRegretMin` for the array of variables `cost`. This parameter uses the principle of *least regret* to guide the search strategy. For more information on `IloChooseMaxRegretMin`, see the section "Solve" on page 234.

Step 2 Create goal1

Add the following code after the comment `//Create goal1`

```
IloGoal goal1 = IloGenerate(env, cost, IloChooseMaxRegretMin);
```

Now, you apply a node evaluator to `goal1`. In this lesson, you apply a node evaluator that implements Depth-Bounded Discrepancy Search (DDS). DDS makes the assumption that mistakes are more likely near the top of the search tree rather than further down. For this reason, this procedure does not count the number of discrepancies but the *depth* of the last one.

The function `IloDDSEvaluator` creates and returns an instance of `IloNodeEvaluator` that implements Depth-Bounded Discrepancy Search. The first parameter is the environment. The second parameter `step`. Solver will first explore nodes with a depth less than `step`. After this exploration is complete, it will explore nodes with a depth between `step` and 2 * `step`, and so on. The third parameter is `width`. Solver will look first at paths, starting from the depth level, that have a number of discrepancies, or right moves, less than or equal to `width`. The fourth parameter is `maxDiscrepancy`. If this parameter is used, Solver will discard nodes with a number of discrepancies greater than `maxDiscrepancy`. The default value for `maxDiscrepancy` is `IloIntMax`, which means that Solver will not discard any nodes based on this parameter. Here is a constructor:

```
IloNodeEvaluator IloDDSEvaluator(IloEnv env,
                                 IloInt step = 4,
                                 IloInt width = 2,
                                 IloInt maxDiscrepancy = IloIntMax);
```

Figure 14.2 shows a search tree that has been traversed with a node evaluator that implements Depth-Bounded Discrepancy Search (DDS) with a depth of 2 and a width of 1. The numbers in the first row represent the order in which paths are evaluated. As you can see, Solver first evaluates paths 1, 2, and 3. It then jumps to paths 4, 5, and 6. It then jumps to paths 7, 8, and 9, and so on. The numbers in the second row represent the number of discrepancies, or right moves, in each path after the level of depth 2. This is called the width. As you can see, Solver first search the paths with 0 or 1 discrepancy and then the paths with 2 discrepancies.

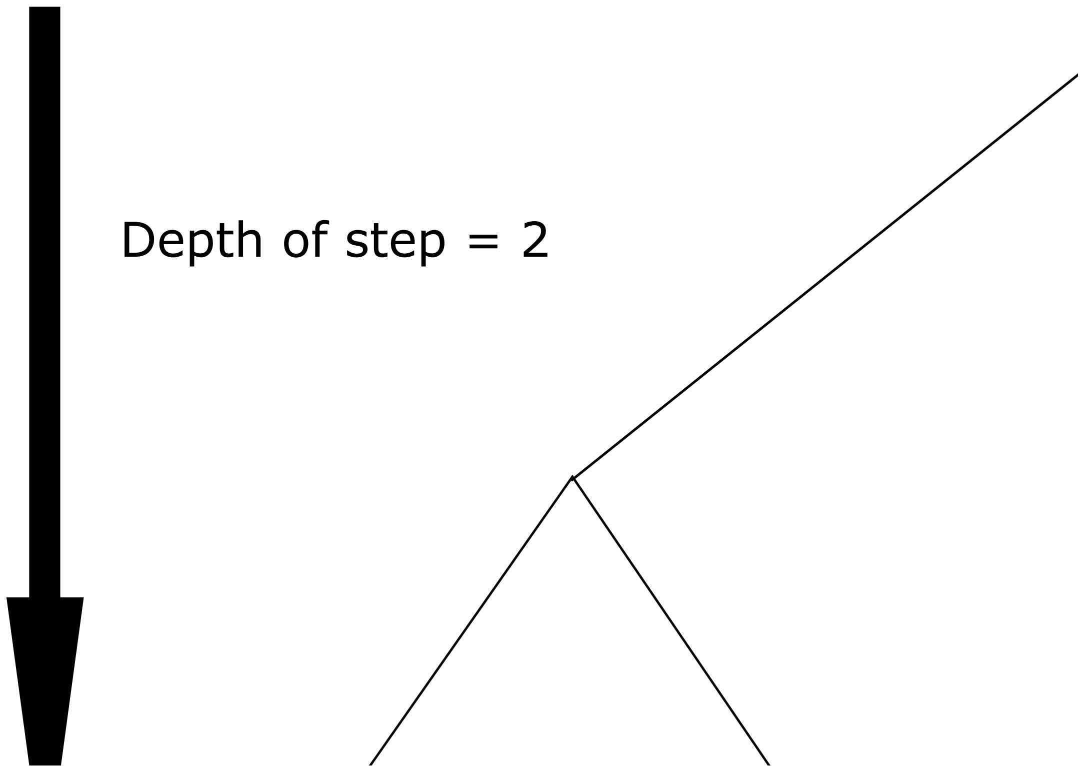

Figure 14.2 Search tree using a Depth-Bounded Discrepancy Search (DDS) with a depth of 2 and a width of 1.

You now use the function `IloApply`. This function applies the node evaluator `IloDDSEvaluator` to `goal1` and returns `applygoal1`. The node evaluator will first search paths in the search tree with a depth of 3 or less nodes. It will search paths deeper than this depth that have a number of discrepancies, or right moves, less than or equal to 1.

Step 3 Create applygoal1

Add the following code after the comment `//Create applygoal1`

```
IloGoal applygoal1 = IloApply(env, goal1, IloDDSEvaluator(env, 3, 1));
```

Now, you will apply a search limit to the goal `applygoal1`. You will use the function `IloFailLimit` to limit the search after `maxNbFails` failures have occurred. Here is a constructor:

```
IloSearchLimit IloFailLimit(IloEnv env, IloInt maxNbFails);
```

Now, you use the function `IloLimitSearch`. It takes three parameters: the environment, a goal, and a search limit. This function places a fail limit on `applygoal1` and returns `limitgoal1`. The fail limit stops the search after 5 failures have occurred.

Step 4 Create limitgoal1

Add the following code after the comment `//Create limitgoal1`

```
IloGoal limitgoal1 = IloLimitSearch(env, applygoal1, IloFailLimit(env, 5));
```

You use the function `IloGenerate` to return `goal2` for the decision variable `supplier`.

Step 5 Create goal2

Add the following code after the comment `//Create goal2`

```
IloGoal goal2 = IloGenerate(env, supplier);
```

You now use the function `IloApply`. This function applies the node evaluator `IloSBSEvaluator` to `goal2` and returns `applygoal2`. This node evaluator will first search paths in the search tree with 1 or less discrepancies, or right moves. It will also discard nodes with a number of discrepancies greater than or equal to 1. For more information on the node evaluator `IloSBSEvaluator`, see the section “Solve” on page 234 in Chapter 13, *Controlling the Search: Locating Warehouses*.

Step 6 Create applygoal2

Add the following code after the comment `//Create applygoal2`

```
IloGoal applygoal2 = IloApply(env, goal2, IloSBSEvaluator(env, 1, 1));
```

Next, you create `combinedGoal` by joining the following goals with logical AND operators: `limitgoal1`, `applygoal2`, and a goal returned by the function `IloInstantiate` for the variable `totalCost`.

Step 7 Create the combinedGoal

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

```
IloGoal combinedGoal = limitgoal1 &&
                       applygoal2 &&
                       IloInstantiate(env, totalCost);
```

Then, you create a search selector, `minimizeSearchSelector`, from the `totalCost` variable. This search selector stores the leaf of the search tree that represents a solution with the optimal value of the variable `totalCost`. After the search tree has been completely explored, it reactivates this value. When a solution is found, it adds a constraint that `totalCost` must be less than or equal to the solution minus `1`.

Step 8 Create the search selector

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

```
IloSearchSelector minimizeSearchSelector = IloMinimizeVar(env,
                                                          totalCost,
                                                          1);
```

Finally, you use the function `IloSelectSearch`. This function takes `combinedGoal` and `minimizeSearchSelector` as parameters and returns `finalGoal`. It applies the search selector `minimizeSearchSelector` to the goal `combinedGoal`.

Step 9 Create the finalGoal

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

```
IloGoal finalGoal = IloSelectSearch(env,
                                    combinedGoal,
                                    minimizeSearchSelector);
```

Now, you create an instance of `IloSolver` to solve the problem expressed in the model. This code is provided for you:

```
IloSolver solver(model);
```

You use the member function `IloSolver::solve` to search for a solution. You use `finalGoal` as a parameter of `IloSolver::solve`.

Step 10 Search for a solution

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

```
if (solver.solve(finalGoal))
```

Step 11 Compile and run the program

Compile and run the program. You should get the following results, though the information displayed by `IloSolver::printInformation` will vary depending on platform, machine, configuration, and so on:

```
Using default file
30
[[20, 24, 11, 25, 30], [28, 27, 82, 83, 74], [74, 97, 71, 96, 70], [2, 55, 73,
6
9, 61], [46, 96, 59, 83, 4], [42, 22, 29, 67, 59], [1, 5, 73, 59, 56], [10, 73,
13, 43, 96], [93, 35, 63, 85, 46], [47, 65, 55, 71, 95]]
[1, 4, 2, 1, 3]
------------------------------
Solution:
Store  0 Warehouse:  Rome
Store  1 Warehouse:  Bordeaux
Store  2 Warehouse:  Rome
Store  3 Warehouse:  Bonn
Store  4 Warehouse:  Rome
Store  5 Warehouse:  Bordeaux
Store  6 Warehouse:  Bordeaux
Store  7 Warehouse:  London
Store  8 Warehouse:  Bordeaux
Store  9 Warehouse:  London

Store  0 Cost:  30
Store  1 Cost:  27
Store  2 Cost:  70
Store  3 Cost:  2
Store  4 Cost:  4
Store  5 Cost:  22
Store  6 Cost:  5
Store  7 Cost:  13
Store  8 Cost:  35
Store  9 Cost:  55

Total cost:  383
------------------------------
Number of fails                : 5
Number of choice points        : 10
Number of variables            : 76
Number of constraints          : 76
Reversible stack (bytes)       : 12084
Solver heap (bytes)            : 52284
Solver global heap (bytes)     : 20124
And stack (bytes)              : 4044
Or stack (bytes)               : 4044
Search Stack (bytes)           : 4044
Constraint queue (bytes)       : 11160
Total memory used (bytes)      : 107784
Elapsed time since creation    : 0.02
```

As you can see, the solution is the same as that found in Chapter 13, *Controlling the Search: Locating Warehouses*. However, there are fewer fails and fewer choice points. Using problem limits and decomposition, there are only 5 fails and 10 choice points, as opposed to 25 fails and 27 choice points using slice-based search without problem limits and decomposition.

The complete program is listed in "Complete program" on page 272. You can also view it online in the file `YourSolverHome/examples/src/storecpx.cpp`.

IBM® ILOG® Solver Search Strategies

Depth-First Search (DFS)

DFS is the standard search procedure. Depth-First Search explores the search tree in a “left-to-right” fashion. It searches by selecting the left leaf at each open node until the end of the search space is reached. If it does not find a solution, it backtracks and tries again. The function `IloDFSEvaluator` implements this strategy. See Chapter 13, *Controlling the Search: Locating Warehouses*.

Best First Search (BFS)

BFS is described in N. J. Nilsson’s book, *Problem Solving Methods in Artificial Intelligence*, published by McGraw-Hill, 1971. The Solver implementation uses a parameter ε. When selecting an open node, Solver determines the set of open nodes the cost of which is at most (1+ε) worse than the best open node. If the child of the current node is in the set, Solver goes to this child. If not, Solver chooses the best open node. The function `IloBFSEvaluator` implements this strategy.

Slice-Based Search (SBS)

SBS is described by J. Christopher Beck and Laurent Perron in their paper, “Discrepancy Bounded Depth First Search,” presented at the Second International Workshop on Integration of AI and OR Technologies for Combinatorial Optimization Problems (CP-AI-OR'00), Paderborn, Germany, 2000. The *discrepancy* of a search node is defined as the number of right moves. Given a parameter *step*, slice-based search will first explore nodes with a discrepancy less than *step*. After this exploration is complete, it will explore nodes with a discrepancy between *step* and *2*step*, and so on. This search strategy cuts the search tree into slices. The function `IloSBSEvaluator` implements this strategy. See Chapter 13, *Controlling the Search: Locating Warehouses*.

Depth-Bounded Discrepancy Search (DDS)

DDS was introduced by Toby Walsh in his paper, “Depth-Bounded Discrepancy Search,” published in the *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, volume 2, August, 1997, pages 1388–1393. DDS makes the assumption that mistakes are made more likely near the top of the search tree than further down. For this reason, this procedure does not count the number of discrepancies but the *depth* of the last one. Solver implements an improved version of Walsh’s schema. Given a parameter *step*, DDS will first explore nodes with a depth less than *step*. After this exploration is complete, it will explore nodes with a depth between *step* and *2*step*, and so on. Solver’s implementation of DDS takes a parameter *width*, that allows *width* number of discrepancies beyond the depth specified by the parameter *step*. The function `IloDDSEvaluator` implements this strategy. See Chapter 14, *Limits and Problem Decomposition: Locating Warehouses*.

Interleaved Depth-First Search (IDFS)

IDFS was introduced by Pedro Meseguer in his paper, “Interleaved Depth-First Search,” published in the *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, volume 2, August, 1997, pages 1382–1387. IDFS tries to mimic the behavior of an infinite number of threads exploring the search tree. Solver implements a variation that limits the depth of this behavior. The function `IloIDFSEvaluator` implements this strategy.

Review exercises

For answers, see *Suggested answers*.

1. What is Depth-Bounded Discrepancy Search?
2. How do you implement a search limit?
3. Modify the program in this lesson to use the data file `YourSolverHome/examples/data/store_ex3.dat`. This data file contains information for 30 stores and 15 warehouses. The ten additional potential warehouse sites are: Munich, Barcelona, Prague, Dublin, Madrid, Lisbon, Berlin, Amsterdam, Brussels, and Milan.

Suggested answers

Exercise 1

What is Depth-Bounded Discrepancy Search?

Suggested Answer

This procedure counts the *depth* of the last discrepancy in the search tree. Given a parameter *step*, DDS will first explore nodes with a depth less than *step*. After this exploration is complete, it will explore nodes with a depth between *step* and *2*step*, and so on. DDS also takes a parameter *width*, that allows *width* number of discrepancies beyond the depth specified by the parameter *step*. The function `IloDDSEvaluator` implements this search strategy.

Exercise 2

How do you implement a search limit?

Suggested Answer

You use the function `IloLimitSearch`. It takes three parameters: the environment, a goal, and a search limit. This function places a search limit on a goal and returns another goal.

Exercise 3

Modify the program in this lesson to use the data file `YourSolverHome/examples/data/store_ex3.dat`. This data file contains information for 30 stores and 15 warehouses. The ten additional potential warehouse sites are: Munich, Barcelona, Prague, Dublin, Madrid, Lisbon, Berlin, Amsterdam, Brussels, and Milan.

Suggested Answer

You can view the complete program online in the file `YourSolverHome/examples/src/storecpx_ex3.cpp`. You only need to change the following line of code to add the additional warehouse sites to the enumeration:

```
const char* Suppliers[] = {"Bonn", "Bordeaux", "London", "Paris", "Rome",
                           "Munich", "Barcelona", "Prague", "Dublin", "Madrid",
                           "Lisbon", "Berlin", "Amsterdam", "Brussels",
"Milan"};
```

You change the following line of code to use the new data file:

```
      fileName = "../../../examples/data/store_ex3.dat";
```

The results will vary depending on platform, machine, configuration, and so on.

```
------------------------------
Solution:
Store  0 Warehouse:  Lisbon
Store  1 Warehouse:  Barcelona
Store  2 Warehouse:  Barcelona
Store  3 Warehouse:  Munich
Store  4 Warehouse:  Rome
Store  5 Warehouse:  Munich
Store  6 Warehouse:  Bonn
Store  7 Warehouse:  London
Store  8 Warehouse:  Berlin
Store  9 Warehouse:  Barcelona
Store  10 Warehouse:  Paris
Store  11 Warehouse:  Bordeaux
Store  12 Warehouse:  Bonn
Store  13 Warehouse:  Bordeaux
Store  14 Warehouse:  Lisbon
Store  15 Warehouse:  Munich
Store  16 Warehouse:  Bonn
Store  17 Warehouse:  Munich
Store  18 Warehouse:  Berlin
Store  19 Warehouse:  Barcelona
Store  20 Warehouse:  Bordeaux
Store  21 Warehouse:  Bonn
Store  22 Warehouse:  Paris
Store  23 Warehouse:  Prague
Store  24 Warehouse:  Barcelona
Store  25 Warehouse:  Munich
Store  26 Warehouse:  Munich
Store  27 Warehouse:  Bonn
Store  28 Warehouse:  London
Store  29 Warehouse:  Lisbon

Store  0 Cost:  2
Store  1 Cost:  15
Store  2 Cost:  8
Store  3 Cost:  1
Store  4 Cost:  4
Store  5 Cost:  1
```

```
Store  6 Cost:  1
Store  7 Cost:  13
Store  8 Cost:  6
Store  9 Cost:  22
Store  10 Cost:  2
Store  11 Cost:  5
Store  12 Cost:  11
Store  13 Cost:  15
Store  14 Cost:  10
Store  15 Cost:  11
Store  16 Cost:  1
Store  17 Cost:  10
Store  18 Cost:  5
Store  19 Cost:  22
Store  20 Cost:  8
Store  21 Cost:  1
Store  22 Cost:  9
Store  23 Cost:  19
Store  24 Cost:  22
Store  25 Cost:  1
Store  26 Cost:  1
Store  27 Cost:  10
Store  28 Cost:  19
Store  29 Cost:  2

Total cost:  557
------------------------------
------------------------------
Number of fails               : 5
Number of choice points       : 39
Number of variables           : 526
Number of constraints         : 526
Reversible stack (bytes)      : 76404
Solver heap (bytes)           : 361824
Solver global heap (bytes)    : 24144
And stack (bytes)             : 4044
Or stack (bytes)              : 4044
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 11160
Total memory used (bytes)     : 485664
Elapsed time since creation   : 0.181
```

As you can see, the solution is not the same as that found in Chapter 13, *Controlling the Search: Locating Warehouses*. The total cost is also higher; 557 for using problem limits and decomposition, and 490 for a complete Slice-Based Search. Since the search uses limits, it is not complete and may miss an optimal solution. However, Solver was able to find a solution more quickly. There are fewer fails and fewer choice points. Using problem limits and decomposition, there are only 5 fails and 39 choice points, as opposed to 68,415 fails and 65,840 choice points using Slice-Based Search without problem limits and decomposition. The performance also improved. Using problem limits and decomposition, Solver found the solution in 0.181 seconds, compared to 86.835 seconds using Slice-Based Search without problem limits and decomposition. (Solution time varies depending on platform, machine,

and so on.) This demonstrates the importance of choosing good search strategies, especially as problem size and complexity increase.

Here are the results using Slice-Based Search without problem limits and decomposition from Chapter 13, *Controlling the Search: Locating Warehouses*:

```
Using default file
Cost to build a warehouse:
30
Relative costs for stores:
[[20, 24, 11, 25, 30, 28, 27, 82, 83, 74,
2, 55, 73, 69, 61], [28, 27, 82, 83, 74, 93, 15, 63, 87, 46,
47, 57, 55, 71, 95], [74, 97, 71, 96, 70, 42, 8, 29, 67, 59,
93, 15, 63, 87, 46], [2, 55, 73, 69, 61, 1, 65, 73, 59, 56,
93, 35, 19, 85, 46], [46, 96, 59, 83, 4, 42, 22, 56, 67, 59,
11, 73, 15, 43, 96], [42, 22, 29, 67, 59, 1, 5, 57, 67, 56,
42, 8, 29, 67, 59], [1, 5, 73, 59, 56, 1, 5, 73, 59, 56,
42, 8, 29, 67, 59], [10, 73, 13, 43, 96, 42, 22, 56, 67, 59,
42, 56, 29, 56, 59], [93, 35, 63, 85, 46, 13, 22, 29, 66, 59,
65, 6, 73, 59, 56], [47, 65, 55, 71, 95, 42, 22, 29, 69, 59,
93, 25, 45, 85, 46],
[17, 19, 67, 2, 15, 93, 35, 19, 85, 46,
47, 57, 55, 71, 95], [26, 5, 78, 43, 19, 10, 15, 13, 9, 96,
68, 35, 62, 85, 46], [11, 73, 15, 43, 96, 42, 56, 29, 56, 59,
47, 57, 55, 71, 95], [93, 15, 63, 87, 46, 93, 55, 63, 92, 46,
47, 65, 55, 92, 95], [47, 65, 37, 71, 95, 47, 65, 45, 71, 95,
10, 73, 58, 43, 96], [42, 22, 29, 67, 49, 11, 45, 13, 43, 96,
42, 56, 29, 56, 59], [1, 5, 73, 59, 56, 47, 65, 45, 71, 95,
68, 35, 62, 85, 46], [10, 75, 13, 43, 62, 10, 15, 13, 9, 96,
93, 35, 19, 85, 46], [93, 25, 45, 85, 46, 65, 6, 73, 59, 56,
12, 5, 92, 59, 56], [47, 65, 55, 92, 95, 42, 22, 29, 69, 59,
47, 57, 55, 71, 95],
[42, 8, 29, 67, 59, 93, 35, 89, 85, 46,
47, 65, 56, 71, 95], [1, 5, 57, 67, 56, 42, 22, 56, 67, 5,
65, 6, 73, 59, 56], [10, 15, 13, 9, 96, 42, 22, 29, 69, 59,
47, 57, 55, 71, 95], [68, 35, 62, 85, 46, 93, 35, 19, 85, 46,
47, 57, 55, 71, 95], [47, 57, 55, 71, 95, 42, 22, 29, 69, 59,
93, 25, 45, 85, 46], [42, 22, 29, 69, 59, 1, 65, 73, 59, 56,
93, 35, 19, 85, 46], [1, 65, 73, 59, 56, 1, 65, 73, 59, 56,
93, 35, 19, 85, 46], [10, 73, 13, 83, 96, 11, 45, 13, 43, 96,
42, 56, 29, 56, 59], [93, 35, 19, 85, 46, 93, 35, 19, 85, 46,
47, 57, 55, 71, 95], [47, 65, 45, 71, 95, 28, 27, 82, 83, 74,
2, 55, 73, 69, 61]]
Warehouse capacities:
[5, 9, 12, 3, 4, 6, 8, 1, 6, 11,
3, 10, 8, 9, 3]

------------------------------
Solution:
Store  0 Warehouse:  London
Store  1 Warehouse:  Barcelona
Store  2 Warehouse:  Barcelona
Store  3 Warehouse:  Munich
Store  4 Warehouse:  Lisbon
Store  5 Warehouse:  Munich
Store  6 Warehouse:  Bordeaux
Store  7 Warehouse:  London
Store  8 Warehouse:  Munich
```

```
Store  9 Warehouse:  Barcelona
Store  10 Warehouse:  Bordeaux
Store  11 Warehouse:  Bordeaux
Store  12 Warehouse:  London
Store  13 Warehouse:  Bordeaux
Store  14 Warehouse:  Lisbon
Store  15 Warehouse:  Munich
Store  16 Warehouse:  Bordeaux
Store  17 Warehouse:  London
Store  18 Warehouse:  Barcelona
Store  19 Warehouse:  Barcelona
Store  20 Warehouse:  Bordeaux
Store  21 Warehouse:  Bordeaux
Store  22 Warehouse:  London
Store  23 Warehouse:  Bordeaux
Store  24 Warehouse:  Barcelona
Store  25 Warehouse:  Munich
Store  26 Warehouse:  Munich
Store  27 Warehouse:  London
Store  28 Warehouse:  London
Store  29 Warehouse:  Lisbon

Store  0 Cost:  11
Store  1 Cost:  15
Store  2 Cost:  8
Store  3 Cost:  1
Store  4 Cost:  11
Store  5 Cost:  1
Store  6 Cost:  5
Store  7 Cost:  13
Store  8 Cost:  13
Store  9 Cost:  22
Store  10 Cost:  19
Store  11 Cost:  5
Store  12 Cost:  15
Store  13 Cost:  15
Store  14 Cost:  10
Store  15 Cost:  11
Store  16 Cost:  5
Store  17 Cost:  13
Store  18 Cost:  6
Store  19 Cost:  22
Store  20 Cost:  8
Store  21 Cost:  5
Store  22 Cost:  13
Store  23 Cost:  35
Store  24 Cost:  22
Store  25 Cost:  1
Store  26 Cost:  1
Store  27 Cost:  13
Store  28 Cost:  19
Store  29 Cost:  2

Total cost:  490


------------------------------
```

```
Number of fails               : 68415
Number of choice points       : 65840
Number of variables           : 526
Number of constraints         : 526
Reversible stack (bytes)      : 104544
Solver heap (bytes)           : 361824
Solver global heap (bytes)    : 4584576
And stack (bytes)             : 4044
Or stack (bytes)              : 4044
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 13164
Total memory used (bytes)     : 5076240
Elapsed time since creation   : 86.835
```

Complete program

The complete locating warehouses program follows. You can also view it online in the file `YourSolverHome/examples/src/storecpx.cpp`.

```
#include <ilsolver/ilosolverint.h>
ILOSTLBEGIN

class FileError : public IloException {
public:
  FileError() : IloException("Cannot open data file") {}
};

const char* Suppliers[] = {"Bonn", "Bordeaux", "London", "Paris", "Rome"};

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

    const char* fileName;
    if ( argc != 2 ) {
      env.warning() << "usage: " << argv[0] << " <filename>" << endl;
      env.warning() << "Using default file" << endl;
      fileName = "../../../examples/data/store.dat";
    } else
      fileName = argv[1];

    ifstream file(fileName);
    if ( !file )
      throw FileError();
    // model data
    IloInt       buildingCost;
    IloIntArray2 costMatrix(env);
    IloIntArray  capacity(env);
    file >> buildingCost >> costMatrix >> capacity;
    IloInt nStores    = costMatrix.getSize();
    IloInt nSuppliers = capacity.getSize();
```

```
    env.out() << "Cost to build a warehouse:  " << endl;
    env.out() << buildingCost << endl;
    env.out() << "Relative costs for stores:  " << endl;
    env.out() << costMatrix        << endl;
    env.out() << "Warehouse capacities:  " << endl;
    env.out() << capacity     << endl;
   // build model
    IloIntVarArray supplier(env, nStores, 0, nSuppliers-1);
    IloIntVarArray cost(env, nStores, 0, 99999);
    IloIntVarArray open(env, nSuppliers, 0,1);
    for (i = 0; i < nStores; i++){
      model.add(cost[i] == costMatrix[i](supplier[i]));
      model.add(open(supplier[i])==1 );
    }
    for (j = 0; j < nSuppliers; j++ ) {
      IloIntVarArray temp(env, nStores, 0, 1);
      for (IloInt k = 0; k < nStores; k++)
        model.add(temp[k] == (supplier[k] == j));
      model.add(IloSum(temp) <= capacity[j]);
    }
    IloIntVar totalCost(env, 0, 999999);
    model.add(totalCost == IloSum(cost) + IloSum(open) * buildingCost);
    IloGoal goal1 = IloGenerate(env, cost, IloChooseMaxRegretMin);
    IloGoal applygoal1 = IloApply(env, goal1, IloDDSEvaluator(env, 3, 1));
    IloGoal limitgoal1 = IloLimitSearch(env, applygoal1, IloFailLimit(env, 5));
    IloGoal goal2 = IloGenerate(env, supplier);
    IloGoal applygoal2 = IloApply(env, goal2, IloSBSEvaluator(env, 1, 1));
    IloGoal combinedGoal = limitgoal1 && applygoal2 && IloInstantiate(env,
totalCost);
    IloSearchSelector minimizeSearchSelector = IloMinimizeVar(env,
                                                              totalCost,
                                                              1);
    IloGoal finalGoal = IloSelectSearch(env,
                                        combinedGoal,
                                        minimizeSearchSelector);
    // solve model
    IloSolver solver(model);
    if (solver.solve(finalGoal))
      {
      solver.out() << "-------------------------------" << endl;
      solver.out() << "Solution:  " << endl;
      for (i = 0; i < nStores; i ++)
        solver.out() << "Store  " << i << " " << "Warehouse:  "
                     << Suppliers[(IloInt)solver.getValue(supplier[i])]
                     << " " << endl;
        solver.out() << endl;
        for (j = 0; j < nStores ; j ++)
          solver.out() << "Store  " << j << " " << "Cost:  "
                       << solver.getValue(cost[j]) << " " << endl;
          solver.out() << endl;
          solver.out() << "Total cost:  " << solver.getValue(totalCost)
                                          << endl;
          solver.out() << "-------------------------------" << endl;

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

```
  catch (IloException& ex) {
    cout << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Results

Solution time varies depending on platform, machine, and so on.

```
Using default file
30
[[20, 24, 11, 25, 30], [28, 27, 82, 83, 74], [74, 97, 71, 96, 70], [2, 55, 73,
6
9, 61], [46, 96, 59, 83, 4], [42, 22, 29, 67, 59], [1, 5, 73, 59, 56], [10, 73,
13, 43, 96], [93, 35, 63, 85, 46], [47, 65, 55, 71, 95]]
[1, 4, 2, 1, 3]
------------------------------
Solution:
Store  0 Warehouse:  Rome
Store  1 Warehouse:  Bordeaux
Store  2 Warehouse:  Rome
Store  3 Warehouse:  Bonn
Store  4 Warehouse:  Rome
Store  5 Warehouse:  Bordeaux
Store  6 Warehouse:  Bordeaux
Store  7 Warehouse:  London
Store  8 Warehouse:  Bordeaux
Store  9 Warehouse:  London

Store  0 Cost:  30
Store  1 Cost:  27
Store  2 Cost:  70
Store  3 Cost:  2
Store  4 Cost:  4
Store  5 Cost:  22
Store  6 Cost:  5
Store  7 Cost:  13
Store  8 Cost:  35
Store  9 Cost:  55

Total cost:  383
------------------------------
Number of fails               : 5
Number of choice points       : 10
Number of variables           : 76
Number of constraints         : 76
Reversible stack (bytes)      : 12084
Solver heap (bytes)           : 52284
Solver global heap (bytes)    : 20124
And stack (bytes)             : 4044
Or stack (bytes)              : 4044
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 11160
Total memory used (bytes)     : 107784
Elapsed time since creation   : 0.02
```

CHAPTER

15

Searching for Optimal Solutions: Replanning Warehouses

In this lesson, you will learn more about how to:

- find feasible solutions
- find optimal solutions using objectives
- store solutions
- use multiphase search
- use embedded search

> ***Note:*** *This chapter is not presented using the three-stage method, as in previous chapters in this part. This subject matter lends itself more naturally to a conceptual introduction rather than procedural-based learning. However, when confronted with an optimization problem, you should still use the Describe, Model, and Solve stages.*

Finding feasible solutions

Before discussing optimal solutions in greater depth, let's review how Solver finds feasible solutions. There are two basic ways to do this. The member function `IloSolver::solve`, when used in a model without an objective, will simply find the first feasible solution. For example, in Chapter 2, *Modeling and Solving a Simple Problem: Map Coloring*, you used `IloSolver::solve` in an `if` statement to find the first feasible solution to the problem:

```
if (solver.solve())
```

The member functions `IloSolver::startNewSearch`, `IloSolver::next`, and `IloSolver::endSearch` can be used in a `while` loop to find all feasible solutions to a problem. For example, in Chapter 3, *Using Arrays and Basic Search: Changing Money*, you used these three member functions to search for all feasible solutions to the problem:

```
solver.startNewSearch();
IloInt solutionCounter = 0;
while(solver.next()) {
  solver.out() << "solution  " << ++solutionCounter << ":\t";
  for (IloInt i = 0; i < nCoins; i++)
    solver.out() << solver.getValue(coins[i]) << " ";
  solver.out() << endl;
}
solver.endSearch();
```

Finding optimal solutions using objectives

Sometimes, however, you not only want a solution to a problem but a solution that optimizes a given criterion. For that purpose, Solver offers predefined ways to get solutions that minimize or maximize an objective. The technique used by Solver is widely known as branch & bound.

In Solver, you search for an optimal solution in three steps:

- specify an objective to minimize or maximize
- add the objective to the model
- create an instance of `IloSolver` to extract the model and ask it to solve the problem

In Chapter 5, *Using Objectives: Map Coloring with Minimum Colors*, you learned how to create an objective, an instance of the class `IloObjective`, using the function `IloMaximize`. You added it to the model using `IloModel::add` and used `IloSolver::solve` to find the solution that optimized this objective.

In this section, you will model this same problem, map coloring with minimum colors, using the function `IloMinimize` and a search loop. You will learn more about how Solver uses objectives and how the member functions `IloSolver::startNewSearch`, `IloSolver::restartSearch` and `IloSolver::next` work when used with an objective to search for an optimal solution.

In this version, the objective is the maximum color used (since you number the colors). You define it by using `IloMax` on the array of colors. You want each modification of this maximum to be propagated to the other variables, and any modification of a color variable to be propagated to the maximum as well. More generally, it is essential to link the cost variable to the other variables in the problem by means of *constraints, not mere tests*.

Constraints differ from tests in an important way: once a constraint has been posted, effects of the constraint are *propagated* to reduce the domains of all variables affected by the constraint in ways that a mere test does not have. In this example, since you are using constraints rather than tests to link the objective to other variables, the choices made about the other variables are propagated to the cost variable with absolute accuracy, and the modification of the objective impinges on the other variables as soon as possible. These two ideas are critical so that as soon as possible the minimization process eliminates parts of the search tree that will not produce better solutions.

The function `IloMinimize` takes a constant or expression as its argument. This constant or expression becomes the objective to be minimized. This member function changes the behavior of `IloSolver::next`, like this:

- search for a solution in the way defined by the goals that have been added to the model
- add the constraint that the following solution must be at least a step better than the solution found previously.

In other words, each time `next` is called, the new solution will yield a better value for the objective. The last solution found is the optimal solution.

In this version, to find a solution using the least number of colors, you use `IloMinimize` and loop to find better and better solutions. As you do so, you print the cost you get for each solution.

> ***Note:*** *You could also search for the optimal solution using* `IloMinimize` *and* `IloSolver::solve` *to find only the optimal solution.*

This loop actually produces a series of solutions, ending with an optimal solution. The last call of `next` in that loop searches for a solution better than the optimal, and it consequently fails. Having failed, it backtracks, and unfortunately the optimal solution is lost.

What shall you do to retain that last optimal solution, that was inadvertently “lost” during backtracking? Easy: in the last part of the program, you search again for the best solution found so far. To do that, you first call `IloSolver::restartSearch` in order to restart the search, and then you search for exactly one solution. The member function `restart` keeps

track of the best objective value found. In that way, you insure that you find the optimal solution again.

Here is how you model the objective for this version of the problem.

```
IloNumVarArray AllVars(env, 6, Belgium, Denmark, France,
                       Germany, Netherlands, Luxembourg);
model.add(IloMinimize(env, IloMax(AllVars)));
```

The output of the program looks like this:

```
OPTIMAL Solution
Belgium:     white
Denmark:     blue
France:      blue
Germany:     white
Netherlands: blue
Luxembourg:  red
```

Storing solutions

In the previous section, you used the member function `IloSolver::restartSearch` to keep track of the best objective value found and to restart the search to find the one optimal solution again. Another way to find optimal solutions is to store them.

As discussed in the previous section, the minimization algorithm computes the best solution twice: it is first computed during a loop producing better and better solutions, and it is computed once more after that loop. In order to avoid this unnecessary computation, you can store intermediate solutions.

The process to do so is quite simple:

1. Create an instance of `IloSolution`.
2. Add the variables you want to store to the solution.
3. Store the solution object.

Once a solution object is stored, you can restore the values contained within it by passing the function `IloRestoreSolution` to the solver.

For example, the following code uses stored solutions in an optimization procedure:

```
IloSolver solver(model);
solver.setOptimizationStep(0.1);

IloSolution solution(env);
solution.add(next);
solution.add(length);

solver.startNewSearch(IloGenerate(env,next) &&
                                    IloDichotomize(env,length) &&
                                    IloStoreSolution(env, solution));

while (solver.next());
solver.endSearch();

solver.printInformation();

solver.solve(IloRestoreSolution(env,solution));
```

Multiphase search

The member functions `IloSolver::startNewSearch` and `IloSolver::next`, in conjunction with the classes `IloSolution` and `IloRestoreSolution`, enable you to search for solutions of a given problem, to edit that problem, and to search for solutions to this new problem. You can even search for solutions of the new problem that are close to solutions of the old one.

Let's assume that you want to search for all solutions of a problem provided you do not change the values of the last four variables. In order to do this, you create two solution objects, one for the variables you do not want to change, and the other for the variables that can be changed. You add the last four variables of the problem to the `IloSolution` object `frozen` and declare a goal `restoreFrozen`, which uses the function `IloRestoreSolution` to restore the values of the solution `frozen`. You add the first four

variables of the problem to the `IloSolution` object `free` and declare a goal `storeFree`, which uses the function `IloStoreSolution` to store the values of the solution `free`.

```
solver.out() << "store 4 last vars" << endl;
IloSolution frozen(env);
frozen.add(x[4]);
frozen.add(x[5]);
frozen.add(x[6]);
frozen.add(x[7]);
frozen.store(solver);
IloGoal restoreFrozen = IloRestoreSolution(env, frozen);

IloSolution free(env);
free.add(x[0]);
free.add(x[1]);
free.add(x[2]);
free.add(x[3]);
IloGoal storeFree = IloStoreSolution(env, free);

solver.startNewSearch(restoreFrozen && goal && storeFree);
while (solver.next()) {
  Print(solver, x);
}
solver.endSearch();
```

You use the following code to search for our new solutions. The member function `IloSolver::startNewSearch` calls the goals `restoreFrozen`, `goal`, and `storeFree` in the search. The solver adds the frozen variables, searches for a solution using `goal` as its objective, and stores the values of the free variables:

```
solver.out() << "repeat last solution" << endl;
IloGoal restoreFree = IloRestoreSolution(env, free);
solver.solve(restoreFrozen && restoreFree);
Print(solver, x);
```

If you want, for example, to obtain a solution that keeps the same values for the last four variables and that respects an additional constraint, it is sufficient to restore the variables you want to keep, add the additional constraint, and search again for a solution, like this:

```
solver.out() << "change solution" << endl;
model.add(x[OL] < 3);
solver.startNewSearch(restoreFrozen && goal);
while (solver.next()) {
  Print(solver, x);
}
solver.endSearch();
solver.printInformation();
```

Multiphase search: Replanning warehouses

In Chapter 13, *Controlling the Search: Locating Warehouses*, you learned how to model and solve a locating warehouses problem. In this chapter, you will model this same problem using multiphase search and stored solutions. The problem is modeled the same as in Chapter 15, *Searching for Optimal Solutions: Replanning Warehouses*.

First, you fix the suppliers of the first four stores by creating a solution object for the variables you do not want to change:

```
IloSolution solution(env);
solution.add(supplier[0]);
solution.add(supplier[1]);
solution.add(supplier[2]);
solution.add(supplier[3]); // we fix the suppliers of the first 4 stores
```

Next, you search for a solution to the whole problem using `searchGoal` and the stored solution `solution`:

```
if (solver.solve(searchGoal && IloStoreSolution(env, solution)))
```

This initial solution is displayed:

```
  {
    solver.out() << "------------------------------" << endl;
    solver.out() << "Initial Solution:  " << endl;
    for (i = 0; i < nStores; i ++)
      solver.out() << "Store  " << i << " " << "Warehouse:  "
                   << Suppliers[(IloInt)solver.getValue(supplier[i])]
                   << " " << endl;
    solver.out() << endl;
    for (j = 0; j < nStores ; j ++)
      solver.out() << "Store  " << j << " " << "Cost:  "
                   << solver.getValue(cost[j]) << " " << endl;
    solver.out() << endl;
    solver.out() << "Total cost:  " << solver.getValue(totalCost)
                 << endl;
    solver.out() << solution << endl;
    solver.out() << "------------------------------" << endl;
```

You add an objective to minimize `totalCost` and search for an optimal solution using the restored solution:

```
    model.add(IloMinimize(env, totalCost));
    solver.solve(IloRestoreSolution(env, solution) && searchGoal);
```

Finally, you display the optimal solution:

```
    solver.out() << "-----------------------------" << endl;
    solver.out() << "Final Solution:  " << endl;
    for (i = 0; i < nStores; i ++)
      solver.out() << "Store  " << i << " " << "Warehouse:  "
                   << Suppliers[(IloInt)solver.getValue(supplier[i])]
                   << " " << endl;
    solver.out() << endl;
    for (j = 0; j < nStores ; j ++)
      solver.out() << "Store  " << j << " " << "Cost:  "
                   << solver.getValue(cost[j]) << " " << endl;
    solver.out() << endl;
    solver.out() << "Total cost:  " << solver.getValue(totalCost)
                 << endl;
    solver.out() << "-----------------------------" << endl;

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

You can view the complete program online in the file `YourSolverHome/examples/src/replan_store.cpp`.

Using embedded search

Sometimes, it may be useful to search for the first solution of a given goal while you are searching for a solution to a set of goals.

For example, let's say that your problem can be decomposed into two subproblems A and B, but you are interested only in the first solution for A. In other words, you know by some means or other that the solutions of A and B are independent, and once you have a solution to A, you need not change it while you continue to search for solutions to B. Let's assume that the problem A is represented by the goal `goalA`, and problem B by `goalB`.

Then the following code find a solution to both problems:

```
solver.startNewSearch(goalA && goalB);
solver.next();
solver.endSearch();
```

However, if problem B has no solutions, backtracking occurs, and Solver searches for another solution to problem A. In fact, as you have set things up in this preliminary model, all solutions of problem B are generated—useless computation in this case. Indeed, since you have already established that they are independent, you know that changing the solution for problem A will not help you find the solution of problem B.

Thus, the following code is better suited to the problem at hand:

```
solver.startNewSearch(goalA && IloStoreSolution(env, solutionA));
solver.next();
solver.endSearch();

solver.startNewSearch(goalB && IloStoreSolution(env, solutionB));
solver.next();
solver.endSearch();
```

You will end up have a solution to both problem A and B stored in `solutionA` and `solutionB`.

Furthermore, if problem A and problem B are not independent, you may still keep this way of solving the problem by committing to the first solution found for the problem A and then searching for a solution of problem B.

This is illustrated by the following code:

```
solver.startNewSearch(goalA && IloStoreSolution(env, solutionA));
solver.next();
solver.endSearch();

solver.startNewSearch(IloRestoreSolution(env, solutionA) &&
                      goalB &&
                      IloStoreSolution(env, solutionB));
solver.next();
solver.endSearch();
```

C H A P T E R

16

Using Parallel Solver: Multithreaded Warehouse Location

In this lesson, you will learn how to:

- solve problems using parallel search
- understand how workers function
- implement a multiphase search with parallelism

This chapter explains how to use Parallel Solver to solve problems on multiprocessors or on platforms that support multithreads. Parallel Solver allows multiple workers running in different threads with copies of the same problem to explore the same shared search tree to find a solution.

> ***Note:*** *This chapter is not presented using the three-stage method, as in previous chapters in this part. This subject matter lends itself more naturally to a conceptual introduction rather than procedural-based learning.*

Understanding the architecture

Conventional Solver applications run sequentially within a single process. In contrast, an application built with Parallel Solver typically runs in multiple threads in order to take

advantage of parallel processing during the exploration of the search tree. For that reason, the architecture of an application of Parallel Solver differs slightly from the architecture of a conventional single-threaded Solver application.

Parallel Solver differs most markedly from conventional Solver with respect to its parallel search. In a Parallel Solver application, there are multiple instances of the search engine. The instances are known as *workers*; they run in different threads yet explore the same search tree. They communicate with each other across a virtual communication layer.

That virtual communication layer fulfills several major tasks:

◆ Load balancing—Parallel Solver makes sure that no worker is idle while work is available. To accomplish this aim, Parallel Solver moves nodes of the search tree from one worker to another.

◆ Bound communication—In an optimization problem, Parallel Solver communicates the best bound known so far among the workers. In a minimization problem, for example, it updates the best upper bound of the objective known so far among the workers. This communication can significantly reduce the search tree. (This bound communication might also be known as propagation, but it differs from the propagation associated with domain reduction in sequential Solver.)

◆ Termination detection—Parallel Solver knows as soon as one worker has found a solution; then it stops the work of the other workers. In contrasting situations, Parallel Solver detects when all workers are idle, and terminates the search cleanly.

The following figure shows you a time line where a main thread in a Parallel Solver application first creates its multiple worker threads. It then sleeps while those workers share a search tree where they explore possible solutions. Once a worker finds a solution, all workers exit, and the main thread awakes to continue execution.

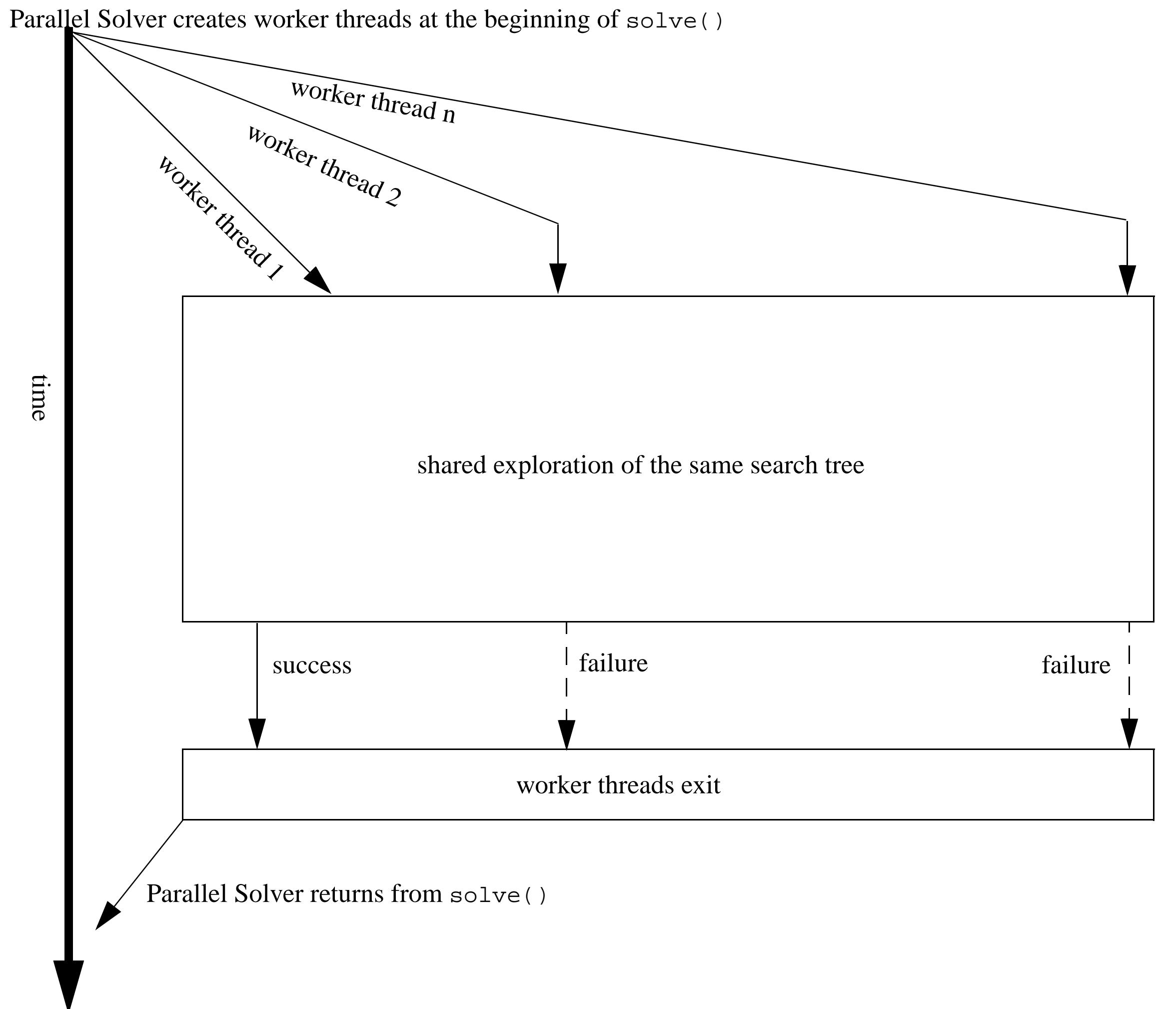

Figure 16.1 *A single Parallel Solver solve() with n workers*

Using Parallel Solver

The example `whouse_mt.cpp` is a variation of the warehouse allocation example introduced in Chapter 13, *Controlling the Search: Locating Warehouses*. It illustrates how to use Parallel Solver.

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

```
#if defined(ILCUSEMT) || defined(ILOUSEMT)

# include <ilsolver/ilopsolver.h>
ILOSTLBEGIN

void readCosts(const char*   name,
               IloInt&       buildingCost,
               IloIntArray2& costs,
               IloInt&       nbClients,
               IloInt&       nbWhouses) {
  ifstream in(name);
  in >> buildingCost >> costs;
  nbClients = costs.getSize();
  if ( nbClients ) nbWhouses = costs[0].getSize();
  else             nbWhouses = 0;
}

int
main(int argc, char** argv)
{
  IloInitMT();
  IloEnv env;
  try {
    IloInt i;
    IloModel model(env);

    IloInt       buildingCost, nbClients, nbWhouses;
    IloIntArray2 costs(env);
    const char* fileName;
    if ( argc < 2 ) {
      env.warning() << "usage: " << argv[0] << " <filename>" << endl;
      env.warning() << "Using default file" << endl;
      fileName = "../../../examples/data/whouse.dat";
    } else fileName = argv[1];
    readCosts(fileName, buildingCost, costs, nbClients, nbWhouses);


    IloIntVarArray offer(env, nbClients, 0, nbWhouses-1);
    IloIntVarArray transCost(env, nbClients, 0, 10000);
    IloBoolVarArray open(env, nbWhouses);

    for (i=0; i < nbClients; i++){
      model.add(transCost[i] == costs[i](offer[i]));
      model.add(open(offer[i]) == 1);
    }

    IloIntVar cost(env, 0, 10000, "Cost\t");
    model.add(cost == IloSum(transCost) + IloSum(open)*buildingCost);

    IloGoal goal = IloGenerate(env, offer) && IloInstantiate(env, cost);
    model.add(IloMinimize(env, cost));

    // end of model

    // We create the Parallel Solver
```

```
    IloParallelSolver psolver(model, 3);

    // We search for an optimal solution

    psolver.solve(goal);

    IloSolver solver = psolver.getWorker(psolver.getSuccessfulWorkerId());
    solver.out() << endl << "Optimal Solution" << endl;
    solver.out() << "Cost " << solver.getValue(cost) << endl;
    solver.out() << "Offer ";
    for (i=0; i<nbClients; i++)
      solver.out() << solver.getValue(offer[i]) << " ";
    solver.out() << endl;
    solver.out() << "TransCost ";
    for (i=0; i<nbClients; i++)
      solver.out() << solver.getValue(transCost[i]) << " ";
    solver.out() << endl;
    solver.out() << "Open ";
    for (i=0; i<nbWhouses; i++)
      solver.out() << solver.getValue(open[i]) << " ";
    solver.out() << endl;
    psolver.end();
  }
  catch (IloException& ex) {
    cout << "Error: " << ex << endl;
  }
  env.end();
  IloEndMT();
  return 0;
}
```

Using multiphase search

Besides the straightforward search for a single solution that you saw in the preceding section, it is also possible to implement a multiphase search in Parallel Solver, where each phase searches differently. To enable you to implement a multiphase search, Parallel Solver offers classes of such multithread objects as mutexes in `IloFastMutex`, semaphores in `IloSemaphore`, conditions in `IloCondition`, and barriers in `IloBarrier` so that your application can synchronize work more precisely among its workers.

The following example shows you how to use those multithread objects to implement multiphase search in your parallel application.

The complete program follows. You can also view it online in the file `YourSolverHome/examples/src/whouse_mt2.cpp`.

```
#if defined(ILCUSEMT) || defined(ILOUSEMT)

# include <ilsolver/ilopsolver.h>
ILOSTLBEGIN
```

```
void readCosts(const char*   name,
               IloInt&       buildingCost,
               IloIntArray2& costs,
               IloInt&       nbClients,
               IloInt&       nbWhouses) {
  ifstream in(name);
  in >> buildingCost >> costs;
  nbClients = costs.getSize();
  if ( nbClients ) nbWhouses = costs[0].getSize();
  else             nbWhouses = 0;
}

int
main(int argc, char** argv)
{
  IloInitMT();
  IloEnv env;
  try {
    IloInt i;
    IloModel model(env);

    IloInt       buildingCost, nbClients, nbWhouses;
    IloIntArray2 costs(env);
    const char* fileName;
    if ( argc < 2 ) {
      env.warning() << "usage: " << argv[0] << " <filename>" << endl;
      env.warning() << "Using default file" << endl;
      fileName = "../../../examples/data/whouse.dat";
    } else fileName = argv[1];
    readCosts(fileName, buildingCost, costs, nbClients, nbWhouses);


    IloIntVarArray offer(env, nbClients, 0, nbWhouses-1);
    IloIntVarArray transCost(env, nbClients, 0, 10000);
    IloBoolVarArray open(env, nbWhouses);

    for (i=0; i < nbClients; i++){
      model.add(transCost[i] == costs[i](offer[i]));
      model.add(open(offer[i]) == 1);
    }

    IloIntVar cost(env, 0, 10000, "Cost\t");
    model.add(cost == IloSum(transCost) + IloSum(open)*buildingCost);

    IloGoal goal = IloGenerate(env, offer) && IloInstantiate(env, cost);


    // end of model

    // We create the Parallel Solver

    IloParallelSolver psolver(model, 3);

    // We do a first run to tighten the problem
    psolver.solve(goal);
    IloSolver solver = psolver.getWorker(psolver.getSuccessfulWorkerId());
    IloInt maxCost = (IloInt)solver.getValue(cost);
    solver.out() << "Setting max cost to " << maxCost << endl;
```

```
    model.add(cost <= maxCost);

    // We search for an optimal solution
    model.add(IloMinimize(env, cost));
    psolver.solve(goal);

    solver = psolver.getWorker(psolver.getSuccessfulWorkerId());
    solver.out() << endl << "Optimal Solution" << endl;
    solver.out() << "Cost " << solver.getValue(cost) << endl;
    solver.out() << "Offer ";
    for (i=0; i<nbClients; i++)
      solver.out() << solver.getValue(offer[i]) << " ";
    solver.out() << endl;
    solver.out() << "TransCost ";
    for (i=0; i<nbClients; i++)
      solver.out() << solver.getValue(transCost[i]) << " ";
    solver.out() << endl;
    solver.out() << "Open ";
    for (i=0; i<nbWhouses; i++)
      solver.out() << solver.getValue(open[i]) << " ";
    solver.out() << endl;
    psolver.end();
  }
  catch (IloException& ex) {
    cout << "Error: " << ex << endl;
  }
  env.end();
  IloEndMT();
  return 0;
}
```

Part IV

Extending the Library

This part consists of the following lessons:

- Chapter 17, *Extraction Concepts*
- Chapter 18, *Writing a Goal: Car Sequencing*
- Chapter 19, *Writing a Constraint: Allocating Frequencies*
- Chapter 20, *Writing a Search Limit: Car Sequencing*
- Chapter 21, *Using Impacts during Search*
- Chapter 22, *Advanced Modeling with Set Variables: Configuring Tankers*

C H A P T E R

17

Extraction Concepts

In this lesson, you will learn how to:

- ◆ understand the extraction process
- ◆ synchronize the model

This lesson details what happens when your Concert Technology model is extracted to an instance of `IloSolver`.

Extractable objects

The term extractable object comes from the fact that an instance of `IloSolver` extracts pertinent information from the modeling object in a form appropriate to that search algorithm for use in solving the problem.

All extractable classes are derived from a common base class `IloExtractable`, documented in the *IBM ILOG Solver Reference Manual*. Concert Technology provides a wealth of extractable classes that allow you to define variables, ranges, columns, various kinds of constraints, and objective functions. The most prominent extractable classes include: `IloModel`, `IloNumVar`, `IloObjective`, `IloConstraint` and `IloRange`.

You can construct an extractable object only within an environment; in other words, you must pass an instance of `IloEnv` as a parameter to the constructor of an extractable object. The environment manages services for the extractable object, such as memory management.

For more information about extractable objects, see the *IBM ILOG Concert Technology User's Manual*.

Extraction process

What occurs during the extraction of a model and what is the correspondence between the model object (starting with `Ilo`) and the Solver object (starting with `Ilc`)? All the extractable objects (that is, instances of `IloExtractable` or one of its subclasses) that have been added to a model (an instance of `IloModel`) and that have not been removed from it and that are relevant to the given search algorithm will be extracted when Concert Technology extracts information from a model.

```
solver.extract(model);
```

In case you need to access the extractable objects in a model, an instance of the embedded class `IloModel::Iterator` accesses those objects. For example, the following lines create the model `model` in the environment `env`, add constraints to the model, and iterate over the extractable objects added to `model`.

```
for (IloModel::Iterator it(model); it.ok(); ++it)   {
        if ((*it).getName())
            env.out() << (*it).getName() << endl;
}
```

After you create a search algorithm in your Concert Technology application, your next step is to extract a model for that search algorithm. You call the member function extract with the model as an argument. Since these two steps—creating the search algorithm and extracting the model—are repeated in most Concert Technology applications, there is a shortcut combining them:

```
IloSolver algo(model);
```

When a search algorithm extracts a model, it scans all the extractable objects in that model and creates corresponding internal objects for optimizing the model. The search algorithm also recursively extracts the other extractable objects referenced by that extractable object.

A search algorithm extracts only one model at a time. If you extract another model for an instance of `IloSolver`, that search algorithm will discard the current model and load the new one.

Main rule of extraction: 1 to 1 correspondence

The main rule of extraction is that there is a 1 to 1 correspondence between extractables and Solver objects. This means that, for each `Ilo` object, there will be one corresponding `Ilc` object. This rule has two exceptions that will be described in the next section.

Let's go through the extraction mechanism.

When an instance of `IloSolver` extracts a model, it iterates over each object added to the model. For each of these objects, it starts extracting them. This mechanism is recursive in the sense that the extraction of an object may lead to the extraction of another object.

The following code example illustrates this process. Here's the code to be extracted:

```
IloEnv env;
IloIntVar a1(env, 0, 3);
IloIntVar a2(env, 0, 3);
IloModel model(env);
model.add(a1 < a2);
IloSolver solver(model);
```

In the last line, an instance of `IloSolver` extracts `model`. First, it takes the range constraint `a1 < a2` and starts extracting it. In order to extract the range constraint, it extracts the two variables `a1` and `a2`. The extraction of these variables creates two `IlcIntVar` objects: `ca1` and `ca2`. Then, the extraction of the range constraint continues. The instance of `IloSolver` takes the two `IlcIntVar` objects and creates the equivalent `Ilc` constraint: `ca1 < ca2`. This inequality constraint is then added to the instance of `IloSolver`.

The extraction will create the following `Ilc` code

```
IlcIntVar ca1(solver, 0, 3);
IlcIntVar ca2(solver, 0, 3);
solver.add(ca1 < ca2);
```

The extraction process can be summarized in the following way. During extraction, an instance of `IloSolver` will iterate over the model, and extract each element one after another. The extraction of an extractable involves the extraction of each of its subextractables. Given an `Ilo` object, all subextractable `Ilo` objects are extracted into `Ilc` objects. An `Ilc` object is then built that is the extraction of the original `Ilo` object.

Exceptions to the rule: The effect of preprocessing

The following rule is always true. For each `Ilo` object in the model, there is a corresponding `Ilc` object. However, this is not always a 1 to 1 correspondence due to preprocessing. Preprocessing causes the following exceptions to the 1 to 1 correspondence rule:

- Exception 1: the corresponding `Ilc` object is not always the same for the same `Ilo` object

- Exception 2: two `Ilo` objects may be extracted into one `Ilc` object.

Exception 1

The following example demonstrates the first effect of preprocessing: the corresponding `Ilc` object is not always the same for the same `Ilo` object. The extraction that takes place will merge the extracted objects.

Here's the code to be extracted

```
IloNumVar x(env, 0, 10, ILOINT);
IloNumVar y(env, 0, 10, ILOINT);
model.add(x == y + 2);
```

You might expect the following code due to the 1-to-1 correspondence rule:

```
IlcIntVar x(solver, 0, 10);
IlcIntVar y(solver, 0, 10);
solver.add(x == y + 2);
```

However, due to preprocessing the extraction will create the following `Ilc` code:

```
IlcIntVar y(solver, 0, 10);
IlcIntVar x = y + 2;
```

In this case, the 1-to-1 correspondence rule is not followed because in the preprocessing phase the expression `y + 2` and the variable `x` are merged into one object: `IlcIntVar x = y + 2`. Even though `IloNumVar x` and `IloNumVar y` are the same type of `Ilo` object they are not extracted into the same type of `Ilc` objects. `IloNumVar y` is extracted into the `Ilc` variable `IlcIntVar y` while `IloNumVar x` is merged with the expression `model.add(x == y + 2)` and extracted into `IlcIntVar x = y + 2`. Though the same type of extraction behavior is expected for these two objects of the same type, this does not occur due to the effects of preprocessing.

Exception 2

The following example demonstrates the second effect of preprocessing: two `Ilo` objects may be extracted into one `Ilc` object.

Here is the code to be extracted:

```
IloNumVar x(env, 0, 10, ILOINT);
IloNumVar y(env, 0, 10, ILOINT);
IloNumVar z(env, 0, 10, ILOINT);
model.add(x == y + 2);
model.add(z == y + 2);
```

You might expect the following code due to the 1-to-1 correspondence rule:

```
 IlcIntVar x(solver, 0, 10);
IlcIntVar y(solver, 0, 10);
IlcIntVar z(solver, 0, 10);
solver.add(x == y + 2);
solver.add(z == y + 2);
```

However, due to preprocessing the extraction will create the following `Ilc` code:

```
IlcIntVar y(solver, 0, 10);
IlcIntVar x = y + 2;
IlcIntVar z = y + 2;
```

The implementation of expressions and variables in Solver will cause `x` and `z` to be extracted into the same physical object. The two `Ilo` objects `x` and `z` are extracted into one `Ilc` object due to the effects of preprocessing. (When you build the same expression twice in Solver, it merges these expressions into the same physical object.)

The case of goals

`IloGoals` are not extractables and they are not added to the model. Instead, they are used directly by the instance of `IloSolver`. Goals are not extracted, even though there is a pseudo-extraction mechanism; the virtual method `IlcGoal IloGoalI::extract(IloSolver)` is called to create a corresponding `IlcGoal`.

The consequence of the fact that goals are not extractables is quite simple. A goal does not extract its subextractables. They must be extracted as a consequence of the extraction of the model.

For instance, the following code is incorrect:

```
IloEnv env;
IloIntVar x(env, 0, 10);
IloIntVar y(env, 0, 10);
IloIntVar z(env, 0, 10);
IloModel model(env);
model.add(x < y);
IloSolver solver(model);
solver.solver(IloInstantiate(env, z));
```

In the last line, the extraction of the `IloInstantiate` goal will raise an exception because `z` will not be extracted in the model. A simple way to ensure that this problem does not occur is to add subextractables directly to the model.

The following code is correct:

```
IloEnv env;
IloIntVar x(env, 0, 10);
IloIntVar y(env, 0, 10);
IloIntVar z(env, 0, 10);
IloModel model(env);
model.add(x < y);
model.add(z);
IloSolver solver(model);
solver.solver(IloInstantiate(env, z));
```

Table of correspondences

The following table shows the correspondences between Ilo and Ilc objects:

Table 17.1 *Table of Correspondences*

The Ilc Object	The Ilo Object
IlcAny	IloAny
IlcAnyArray	IloAnyArray
IlcAnySet	IloAnySet
IlcAnySetVar	IloAnySetVar
IlcAnySetVarArray	IloAnySetVarArray
IlcAnyTupleSet	IloAnyTupleSet
IlcAnyVar	IloAnyVar
IlcAnyVarArray	IloAnyVarArray
IlcBox	IloBox
IlcBool	IloBool
IlcConstraint For the case of constraints, the IloConstraint is extracted as the IlcConstraint with the corresponding name: IloAllDiff > IlcAllDiff	IloConstraint
IlcFloat	IloNum
IlcFloatArray	IloNumArray
IlcFloatSet	IloNumSet
IlcFloatVar	IloNumVar
IlcFloatVarArray	IloNumVarArray
IlcInt	IloInt
IlcIntArray	IloNumArray
IlcIntSet	IloNumSet
IlcIntSetVar	IloNumSetVar

Table 17.1 Table of Correspondences

The Ilc Object	The Ilo Object
IlcIntSetVarArray	IloNumSetVarArray
IlcIntVar	IloNumVar
IlcIntVarArray	IloNumVarArray
IlcIntTupleSet	IloNumTupleSet
IlcNeighborIdentifier	IloNeighborIdentifier

Synchronization of the model

This section describes how an instance of IloSolver reacts to changes in a model it has extracted. The instance of IloSolver is asynchronous with regards to changes in a model. This means that changes to an extracted model will be taken into account only at the next solve of IloSolver::startNewSearch at the top level. This synchronization can be parametrized by two modes: IloSynchronizeAndRestart and IloSynchronizeAndContinue.

There are three kind of changes that can happen to an instance of IloSolver:

- A monotonic change is a change that can be taken into account without restarting the model. Adding a constraint to a model is such a change. These changes are compatible with the synchronize mode IloSynchronizeAndContinue.
- A destructive change is a change that can only result in a full reload of the model by the instance of IloSolver. Extending a domain of a variable is such a change.
- A non-monotonic change is a change that is not compatible with the IloSynchronizeAndContinue mode, but where synchronization will not imply a full reload of the model. Removing a constraint from the model is such a change.

Using IloSynchronizeAndRestart

When using the parameter IloSynchronizeAndRestart during synchronization, the instance of IloSolver will end in a state equivalent to a deletion of the instance of IloSolver and a reload of the current model. In practice, the instance of IloSolver will look at all the changes that have happened since the last time it was synchronized with the model in the IloSynchronizeAndRestart mode. If the list of changes only contains monotonic and non-monotonic changes (no destructive changes), the instance of IloSolver will backtrack to a state before the first search and apply the changes. If the list

of changes contains a destructive change, the instance of `IloSolver` will clear itself and reload the model from scratch.

Using IloSynchronizeAndContinue

Using this parameter is valid if the list of changes that have happened since the last synchronization only contain monotonic changes. (The instance of `IloSolver` will throw an exception otherwise.) If all the changes are monotonic since the last synchronization, the instance of `IloSolver` will apply all the changes from the current state and continue from there.

Environment and Solver memory allocation

With the distinction between the model and the instance of `IloSolver` comes two memory pools. The first is linked to the environment. Memory allocated in the environment is reclaimed when the environment is terminated by the member function `IloEnv::end`. To allocate on this pool, you must pass the `env` as parameter to the `new` operator:

```
MyObject*  myobject = new (env) MyObject();
```

The second memory pool can be used during search, to store additional information reversibly. To do this, you allocate memory on the Solver heap. The memory allocated with this operator is automatically reclaimed when Solver backtracks to a point before the allocation of the object or if the member function `IloSolver::end` is called on the instance of `IloSolver` (this member function is automatically called by `IloEnv::end`). Therefore, to allocate on the Solver heap, you could write:

```
MyObject * myobject = new (solver.getHeap()) MyObject();
```

> ***Note:** You must not use the delete operator for objects allocated on the environment memory pool or on the Solver heap. These objects will be deleted when the memory is reclaimed. Note, however, that the destructor of these objects will not be called when the memory is reclaimed.*

C H A P T E R

18

Writing a Goal: Car Sequencing

In this lesson, you will learn how to:

- understand goals, subgoals, and choice points
- write your own goal
- model and solve a car sequencing problem using a custom goal

In Part I, *The Basics*, you learned in general terms how Solver carries out a search using *goals*. This lesson explains in greater detail how goals are used to implement search algorithms. You will learn how to write your own goal.

You will also learn how to design a model and solve a problem in car sequencing. The problem shows how to combine several types of predefined constraints to create new constraints in Solver. It also shows how to read data from a file as input to the problem, as you might do in an industrial situation, and how to impose an order among values in an algorithm specially designed for a car-sequencing problem. You will write your own goal to solve the problem.

Understanding goals

Goals are the building blocks used to implement search algorithms in Solver. Goals implement algorithms where the exact sequence of operations to follow is not known in

advance. This kind of programming is often called *non-deterministic*. Both predefined search algorithms and user-defined search algorithms can be expressed in Solver through goals. The class `IloGoal` represents goals in a IBM® ILOG® Concert Technology model. The class `IlcGoal` represents goals internally in a Solver search.

Goals as they are represented in a Concert Technology model depend on two classes: `IloGoal` and `IloGoalI`. An instance of the class `IloGoal` (a handle) contains a data member (the handle pointer) that points to an instance of the class `IloGoalI` (its implementation object).

Goals as they are represented in a Solver search (for example, inside a constraint or inside another goal) depend on two classes: `IlcGoal` and `IlcGoalI`. The class `IlcGoal` is the handle class. An instance of the class `IlcGoal` contains a data member (the handle pointer) that points to an instance of the class `IlcGoalI` (the implementation object) allocated on the Solver heap. You can use the macro `ILCGOALn` to define a new class of goals. If you want to use this new class of goals in a Concert Technology model, you must first wrap them using the macro `ILOCPGOALWRAPPER`.

A goal can either succeed or fail. A goal fails if a fail member function (such as `IlcGoalI::fail` or `IloSolver::fail`, for example) is called during its execution. Such a call happens when the domain of a constrained variable becomes empty because of a constraint. A goal succeeds if it does not fail.

Using goals

The member functions `IloSolver::solve` and `IloSolver::next` control the execution of goals. The first time one of these member functions is called, it creates a stack of goals, called the *goal stack*.

> ***Note:*** *Goals must be called by* `IloSolver::solve`*,* `IloSolver::next`*, or by other goals. Otherwise, they are not executed.*

Defining a goal creates two functions:

- The first one creates the goal. The name of this function is the name of the goal, and its parameters are those used to define the goal.
- The second function is a virtual function used to execute the goal. Its name is `execute`. Its body is the body used in the definition of the goal. When this function is used, the *goal is executed.*

A goal is passed as a parameter to the member functions `IloSolver::solve` or `IloSolver::next`. These functions push a goal onto the goal stack of the invoking solver; when that occurs, the *goal is called.*

Thus, each time a goal is called, it is added on top of this stack, but it is not executed immediately. When the current goal execution is finished, the member function

`IloSolver::solve` or `IloSolver::next` pops the goal which is on top of the goal stack, if any, and executes it. Thus goals are executed first in, last out. If the goal stack is empty, the call to `solve` or `next` terminates and returns `IlcTrue`.

Subgoals

A goal can be defined in terms of other goals, called its subgoals. A subgoal is not executed immediately. In fact, it is added to the goal stack, and the execute member function of the current goal terminates before the subgoal is executed. When the execution of the goal itself is complete, the subgoal will be returned.

A goal can be defined as a choice between subgoals. Only one of the subgoals can succeed.This choice point is implemented by the function `IlcOr`.

A goal can also be defined as a sequence of subgoals that must all succeed. This sequence is implemented by the function `IlcAnd`.

Creating choice points using IlcOr

A function that creates a goal takes a constrained variable as its argument (or an array of variables) and binds a value to it (or them). Solver automatically propagates the constraints posted on this constrained variable. There is a hidden problem in that description: in general, Solver does not know which value in the domain is consistent with the constraints. In other words, that binding of the constrained variable must be seen as a guess: if it leads to inconsistencies, it should be undone, and another value should be tried. Solver implement these guesses and “undos” by using choice points.

Choice points are implemented in Solver by the function `IlcOr`. A choice point defines a goal in terms of a choice between subgoals. Solver executes a choice point between two subgoals like this:

- The state of Solver (including the state of all variables and constraints and the state of the goal stack) is saved, so that it can be restored if needed. This is called *setting the choice point*.
- The first subgoal is pushed onto the top of the goal stack.
- The other subgoals are saved as untried subgoals for the choice point.
- Then the first subgoal is popped from the goal stack and executed. If this subgoal fails, the state of Solver is restored, and the first untried choice is pushed onto the goal stack. This activity is called *backtracking*, and it continues until a subgoal is found that succeeds.
- If all subgoals fail, the choice point itself fails.

When the function `IloSolver::fail` is called, goal execution resumes at the most recent choice point with untried subgoals. It is possible to make goal execution resume at an earlier choice point if you associate *labels* with choice points. Then the function

`IloSolver::fail` can be called with a label. In such a case, goal execution resumes at the most recent choice point having the same label.

You can create a choice point with up to five subgoals by means of the function `IlcOr`. For example, in order to define a choice point with eight subgoals `g1`, . . . , `g8`, you can do this:

```
IlcOr(IlcOr(g1, g2, g3, g4, g5),
      IlcOr(g6, g7, g8));
```

The member function `IloSolver::next` searches for one successful execution of a given. To search for all of them, put `next` within a loop.

Creating sequences of subgoals using IlcAnd

A goal can also be defined as a sequence of subgoals that must all succeed. This sequence is implemented by the function `IlcAnd`. This function defines a goal composed of several subgoals (up to five subgoals). The subgoals are executed from left to right.

You can overcome the apparent limitation to five subgoals by calling the function `IlcAnd` several times. For example, in order to define a choice point with eight subgoals `g1`, . . . , `g8`, it is sufficient to call:

```
IlcAnd(IlcAnd(g1, g2, g3, g4, g5),
       IlcAnd(g6, g7, g8));
```

Logical properties of IlcOr and IlcAnd

The functions `IlcOr` and `IlcAnd` are associative. For example, if `g1`, `g2`, and `g3` are goals, the execution of

```
IlcAnd(IlcAnd(g1, g2), g3);
```

is equivalent to the execution of

```
IlcAnd(g1, IlcAnd(g2, g3));
```

You can use this associativity to define goals with more than five subgoals. `IlcOr` is also associative. Thus the execution of

```
IlcOr(IlcOr(g1, g2), g3);
```

is equivalent to the execution of

```
IlcOr(g1, IlcOr(g2, g3));
```

Again, you can use this associativity for defining choice points with more than five subgoals. `IlcAnd` and `IlcOr` are distributive over each other. For example, the execution of

```
IlcAnd(IlcOr(g1, g2), g3);
```

is equivalent to the execution of

```
IlcOr(IlcAnd(g1, g3), IlcAnd(g2, g3));
```

To create a conjunction (logical AND) of goals at the model level, use `IloAndGoal`. To create a disjunction (logical OR) of goals at the model level, use `IloOrGoal`.

Writing your own goal

You can use the macro `ILCGOALn` to define a new class of goals. If you want to use this new class of goals in a Concert Technology model, you must first wrap them using the macro `ILOCPGOALWRAPPER`.

Using ILCGOALn to define a new class of goals

You define a goal using the macro `ILCGOALn`. The definition consists of three parts:

- a name for the goal
- a list of typed parameters, `n` being the number of parameters
- a body which defines how to execute the goal

For example, the following goals merely print something, but they are valid goal definitions:

```
ILCGOAL0(Print){
    cout << "Print: executing a goal without parameters\n";
    return 0;
}
ILCGOAL1(PrintX, IlcInt, x){
    cout << "PrintX: executing a goal with one parameter\n";
    cout << x << endl;
    return 0;
}
ILCGOAL2(PrintXY, IlcInt, x, IlcFloat, y){
    cout << "PrintXY: executing a goal with two parameters\n";
    cout << x << endl;
    cout << y << endl;
    return 0;
}
```

The definition of a goal defines a function for calling it. In that sense, *a goal call is a function call*. For example, the following are valid calls of the goals just defined.

```
Print(solver);
PrintX(solver, 2);
PrintXY(solver, 1, 2.);
```

The goal calling function can be declared in a header file. Its signature has the same name and parameters as the goal, and it returns a pointer to an object of the type `IlcGoal`.

***Note:** Such pointers must not be stored since goals are automatically de-allocated by Solver after they have been executed. They can be passed as arguments as long as the goal has not yet been executed.*

For example, the following declarations are valid declarations of these same goals:

```
IlcGoalI Print(IloSolver solver);
IlcGoalI PrintX(IloSolver solver, IlcInt x);
IlcGoalI PrintXY(IloSolver solver, IlcInt x, IlcFloat y);
```

You can also define a goal using subgoals. The following goal has three subgoals:

```
ILCGOAL0(PrintAll){
   IloSolver solver=getSolver();
   return IlcAnd(Print(solver), PrintX(solver, 2), PrintXY(solver, 1, 2.));
}
```

The execution of the goal `PrintAll` produces the following output:

```
Print: executing a goal without parameters
PrintX: executing a goal with one parameter
2
PrintXY: executing a goal with two parameters
1
2
```

Using ILOCPGOALWRAPPER to wrap the goals

There are several situations in which you might need to wrap a goal in an instance of `IloGoal`:

- You want to use the goal as an argument of `IloSolver::solve`.
- You want to use the goal as an argument of `IloSolver::startNewSearch`.

In such situations, use the macro `ILOCPGOALWRAPPERn` to wrap your goal.

For example, you can wrap the `PrintAll` goal in an instance of `IloGoal` in this way:

```
ILOCPGOALWRAPPER0(IloPrintAll, solver){
        return PrintAll(solver);
        }
```

To create an instance of this goal you write:

```
IloPrintAll(env);
```

The `IloGoal` object returned can then be passed as a parameter of the `IloSolver::solve` and `IloSolver:startNewSearch` functions.

Example of writing your own goal: Implementing IlcInstantiate

As an example of goal programming, you are going to look closely at how the function `IlcInstantiate` is implemented. This function takes a constrained variable as its argument and binds it. More precisely, this function selects a value in the domain of the constrained variable and assigns it to the constrained variable. Solver automatically propagates the constraints posted on this constrained variable.

With the ideas about choice points and backtracking that you have just learned in mind, now you can implement `IlcInstantiate` by means of goals. To do so, you will use the following algorithm:

- ◆ If the constrained variable is bound (that is, it has already been assigned a value), do nothing and succeed.
- ◆ Otherwise, set a choice point between two subgoals. If a selector is specified, the first subgoal assigns a valued based on the selection criteria. Otherwise, the minimum value of the domain of the constrained variable is assigned to it.
- ◆ If a contradiction is detected, execute the second subgoal. The second subgoal removes the tried and failed value from the domain of the constrained variable, and calls `IlcInstantiate` again. Indeed, this call will try another value from the domain of the constrained variable and repeat the process.

> ***Note:*** *The behavior of* `IlcInstantiate` *varies slightly, depending on the class of its arguments. See the IBM ILOG Solver Reference Manual for more detailed information.*

Assigning a value to (or removing a value from) a constrained variable is straightforward. To do so, use the Solver constraints `==` and `!=`.

The Solver code corresponding to `IlcInstantiate` for constrained integer variables is quite simple. You will see a couple versions of it to clarify a few points about goals.

As you remember, if you want to define a goal, you use a macro of the form `ILCGOALn`, where `n` is the number of arguments of the goal. The arguments of `ILCGOALn` are the name of the goal, followed by the types and names of the goal's arguments. Then the macro is followed by the body of a C++ function. This body defines how to execute the goal. The return value of that function is the subgoal of the goal defined by the macro itself. When there are no such subgoals, the function must return 0.

Now you can define `IlcInstantiate` as a choice point between two goals. Before setting a choice point, Solver checks whether the constrained integer variable has been bound, and if not, Solver selects a value in its domain. The second subgoal recursively calls `IlcInstantiate`.

Here is a first version of `IlcInstantiate`:

```
ILCGOAL1(IlcInstantiate, IlcIntVar, var) {
    if (var.isBound())
        return 0;
    IlcInt value = var.getMin();
    return IlcOr( var == value,
                  IlcAnd( var != value,
                          IlcInstantiate(getSolver(), var)));
}
```

Actually, the `ILCGOALn` macros define classes of `IlcGoalI`. As a consequence, you can use `this` to refer to the current goal. With that possibility in mind, here is a second version of `IlcInstantiate`:

```
ILCGOAL1(IlcInstantiate, IlcIntVar, var) {
    if (var.isBound())
        return 0;
    IlcInt value = var.getMin();
    return IlcOr( var == value,
                  IlcAnd( var != value,
                          this));
}
```

(The actual implementation of `IlcInstantiate` is slightly different since the choice of the value can be indicated by a parameter, but these two versions indicate the substance of the implementation.)

If you want to have an `IloGoal` object returned that can then be passed as a parameter of the `IloSolver::solve` and `IloSolver:startNewSearch` functions, use the macro `ILOCPGOALWRAPPER`.

For example, you can define `IloInstantiate` this way:

```
ILOCPGOALWRAPPER1(IloInstantiate, solver, IloNumVar, x){
       return IlcInstantiate(solver, solver.getIntVar(x));
}
```

To create an instance of this goal you write:

```
IloInstantiate(env, x);
```

Arguments of goals: Traps and pitfalls

Goals are not executed immediately when they are called. However, their arguments are computed immediately, and they are saved together with the goal on the goal stack. Those arguments and computations are used when the goal is executed, generally after the function calling the goal has returned.

> ***Note:*** For that reason, *arguments should not contain pointers to automatic objects*, *where automatic objects are objects allocated on the C++ stack, that is, without using the* `new` *operator.*

Consider the following goals:

```
ILCGOAL1(PrintI, IlcInt*, i){
  cout << *i << endl;
  return 0;
}

ILCGOAL1(CallPrintI, IlcInt*, i){
  IlcInt i = 0;
  return PrintI(&i); // error here: i is an automatic
}
```

If the goal `CallPrintI` is executed, `PrintI` is called. This (among other effects) stores the address of `i`. Then the function executing the body of `CallPrintI` returns. Thus the address of `i` is no longer a valid address. Finally, the goal `PrintI` may be executed, causing an error since the address of `i` is no longer valid.

Another subtle point is that arguments are computed before goal execution actually takes place. That fact can lead to some problems. A simple example requires the use of a constrained integer variable.

Consider the following incorrect code:

```
ILCGOAL1(WrongGoal, IlcIntVar, x){
  return IlcAnd(IlcInstantiate(x),
                PrintX(getManager(), x.getValue())); // error: x is unbound
}
```

Here is the way that goal was intended to be executed: the constrained integer variable should be bound by the execution of the goal `IlcInstantiate`; then the execution of the goal `PrintX` should print its value. However, the argument of `PrintX` is computed *before* `IlcInstantiate` is executed. That order of events causes an error since `x` is not yet bound. Indeed, `IlcInstantiate` is pushed onto the goal stack, but it is not executed immediately.

Here is the correct way to pass x as an argument (instead of passing its value):

```
ILCGOAL1(PrintVar, IlcIntVar, x){
  cout << x.getValue() << endl;
  return 0;
}
ILCGOAL1(RightGoal, IlcIntVar, x){
  return IlcAnd(IlcInstantiate(x),
                PrintVar(getManager(), x));   // no error here
}
```

As indicated in this lesson the macros `ILCGOALn` define subclasses of the class `IlcGoalI`. Arguments to the macro are data members of the new class. The most significant consequence of that fact is that modifications of those arguments in the body of the macro are saved. Such modifications often have unfortunate results, and you are strongly advised not to make such modifications.

Writing a goal: Car sequencing

Car sequencing problems arise in the automotive industry. There, a factory assembly line makes it possible to build many different types of cars, where the types correspond to a basic model with added options. In that context, one type of vehicle can be seen as a particular configuration of options. Even without loss of generality, it can be assumed that it is possible to put multiple options on the same vehicle while it is on the line. In that way, virtually any configuration (taken as an isolated case) could be produced on the assembly line. In contrast, for practical reasons (such as the amount of time needed to do so), a given option really cannot be installed on every vehicle on the line. This constraint is defined by the "capacity" of an option. The capacity of an option is usually represented as a ratio p/q where for any sequence of q cars on the line, at most p of them will have that option.

The problem in car sequencing consists of determining in which order cars corresponding to each configuration should be assembled, while keeping in mind that a certain number of cars per configuration must be built.

> ***Note:*** *A simpler version of this problem appears in Chapter 6, Using the Distribute Constraint: Car Sequencing.*

The example discussed is the Second Model in the example file `YourSolverHome/examples/src/carseq.cpp`.

Inputting data

The input for the program in this example is data read from a file. To read the data from the input file, you write a function, `readData`. This function assumes that the data are recorded in a text file with a format based on the following:

- The first line indicates the number of cars to assemble, the number of possible options, and the number of required configurations.
- The second line contains the value *p* that appears in the ration *p/q* of the capacity for each option.
- The third line contains the value *q* of that ratio of the capacity for each option.
- The remaining lines define the options required by each configuration. Each line corresponds to a particular configuration; each line begins with the identifying configuration number, followed by the number of times the option is required by the configuration. For each option, there is a 1 (one) if the option is required by the configuration or 0 (zero) if not.

You can edit either of the two input files among the examples to get a better idea of how the file is formatted.

The function `readData` reads the information from the input file. That function looks like this:

```
IloArray<IloIntArray> readData(IloEnv env,
                               IloInt example,
                               IloInt& nbCars,
                               IloInt& nbOpt,
                               IloIntArray& nbMax,
                               IloIntArray& seqWidth,
                               IloIntArray& confs,
                               IloIntArray& nbCarsByConf){
  char globalName[200];
  switch(example){
  case 1:{
    strcpy(globalName,"../../../examples/data/carseq1.dat");
  } break;
  case 2:{
    strcpy(globalName,"../../../examples/data/carseq2.dat");
  } break;
  }
  IloInt nbConfs;
  ifstream fin(globalName,ios::in);
  if (!fin) env.out() << "problem with file:" << globalName << endl;
  fin >> nbCars >> nbOpt >> nbConfs;

  IloInt i;
  confs = IloIntArray(env, nbConfs); // required configurations
  for(i=0;i<nbConfs;i++){
    confs[i]=i;
  }
  IloArray<IloIntArray> confsByOption(env, nbConfs);
  for (i=0;i<nbConfs;i++){
    confsByOption[i] = IloIntArray(env, nbOpt);
  }
  IloIntArray nbConfsByOption(env, nbOpt);
  for (i=0;i<nbOpt;i++){
    nbConfsByOption[i]=0;
  }
```

```
// read the maximum number of cars of each sequence that can take the
// invoked option
nbMax=IloIntArray(env, nbOpt);
for(i=0;i<nbOpt;i++){
  fin >> nbMax[i];
}

// read the size of a sequence for each option
seqWidth=IloIntArray(env, nbOpt);
for(i=0;i<nbOpt;i++){
  fin >> seqWidth[i];
}

// read the options required by each configuration
nbCarsByConf=IloIntArray(env, nbConfs);
IloInt dummy;
IloInt j;
for(i=0;i<nbConfs;i++){
  fin >> dummy;
  fin >> nbCarsByConf[i];
  for (j=0;j<nbOpt;j++){
    fin >> confsByOption[i][j];
    if (confsByOption[i][j] == 1){
      nbConfsByOption[j]++;
    }
  }
}
```

To represent the data for any given option, you need to know the configurations that require it. The following code does that.

```
 // compute the configurations required by each option
IloArray<IloIntArray> optConf(env, nbOpt);
for(i=0;i<nbOpt;i++){
  optConf[i]=IloIntArray(env, (IloInt)nbConfsByOption[i]);
}
IloIntArray ind(env, nbOpt);
for(i=0;i<nbOpt;i++){
  ind[i]=0;
}
for(i=0;i<nbConfs;i++){
  for (j=0;j<nbOpt;j++){
    if (confsByOption[i][j] == 1){
      optConf[j][(IloInt)ind[j]]=i;
      ind[j]++;
    }
  }
}
```

As it is read, the data is displayed on the screen by the following code.

```
  // print the configuration required by each option
  env.out() << "number of times each configuration is required:" << endl;
  for(i=0;i<nbConfs;i++){
    env.out() << nbCarsByConf[i] << " ";
  }
  env.out() << endl;
  env.out() << "configuration required by each option:" << endl;
  for(i=0;i<nbOpt;i++){
    env.out() << "option:" << i << " ";
    for (j=0;j<optConf[i].getSize();j++){
      env.out() << optConf[i][j] << " ";
    }
    env.out() << endl;
  }
  env.out() << "number of times each option is required" << endl;
  for(i=0;i<nbOpt;i++){
    IloInt cpt=0;
    for(j=0;j<optConf[i].getSize();j++){
      cpt += nbCarsByConf[(IloInt)optConf[i][j]];
    }
    env.out() << "option:" << i << " " << cpt << " x ";
    env.out() << nbMax[i] << "/" << seqWidth[i] << endl;
  }

  return optConf;
}
```

Building the model

You create a constrained variable for each car that should be assembled on the line and organize those variables into an array. You assume that the order in which the cars appear on the line is the same as the order of the constrained variables in the array. That is, `cars[0]` is the first car; `cars[n-1]` is the last.

The values that the constrained variables can assume correspond to the configurations. If a constrained variable is bound to the value 3, for example, the corresponding car will have options corresponding to configuration 3. Initially, a car can have any configuration.

Here is the code for the model

```
void secondModel(IloInt mode, IloInt example)
{
  IloEnv env;
  try {
    IloModel model(env);

    IloInt nbOptions;
    IloInt nbCars;
    IloIntArray confs(env); // array of required configurations
    IloIntArray nbCarsByConf(env); // number of cars to assign
                                    // to each configuration
    IloIntArray nbMax(env);
    IloIntArray seqWidth(env);

    IloArray<IloIntArray> optConf = readData(env,
                                              example,
                                              nbCars,
                                              nbOptions,
                                              nbMax,
                                              seqWidth,
                                              confs,
                                              nbCarsByConf);


    cout << "Pass 1 " << endl;
    const IlcInt nbConfs   = confs.getSize();

    IloInt i,j;
    IloIntVarArray cars(env,nbCars,0,nbConfs-1);
    IloIntVarArray cards(env,nbConfs);
    for (i = 0; i < nbConfs; i++){
      cards[i] = IloIntVar(env,nbCarsByConf[i],nbCarsByConf[i]);
    }
    cout << "Pass 2 " << endl;
```

Adding the distribute and sequence constraints

The number of cars required for each configuration can be expressed by means of the Solver constraint `IloDistribute`. This function takes four parameters: an environment, an array of constrained variables, an array of constrained values, and an array to count those constrained values. The following code actually implements the specific details of our example.

```
    model.add(IloDistribute(env, cards, confs, cars));
```

For more information about the distribute constraint, see Chapter 6, *Using the Distribute Constraint: Car Sequencing*.

At first glance, it seems more difficult to implement the constraints representing the capacities of options in the problem, though it is easy to express such a constraint in natural

language. However, this is possible using the `IloSequence` constraint. You use this constraint to define constraints on sequences of cars. As a sequence constraint, it groups several constraints created by `IloDistribute` in order to reduce the domains of constrained variables more efficiently. More specifically, an instance of this class enables you to constrain:

- the minimum number of allowable values in the sequence,
- the maximum number of allowable values in the sequence,
- the frequency of allowable values (that is, how often a value occurs in the sequence),
- the number of elements in the sequence.

So here for each option `opt`, you define a sequence constraint and add it to the model, like this:

```
IloInt opt;
for (opt = 0; opt < nbOptions; opt++) {
  IloIntVarArray ncard(env,optConf[opt].getSize());
  for(i = 0; i < optConf[opt].getSize(); i++){
    ncard[i] = cards[(IloInt)optConf[opt][i]];
  }
  model.add(IloSequence(env,
                        0,
                        (IloInt)nbMax[opt],
                        (IloInt)seqWidth[opt],
                        cars,
                        optConf[opt],
                        ncard));
}
cout << "Pass 3 " << endl;
```

Choosing the order of variables and values: Using slack

For the car-sequencing problem, the order in which variables are chosen and values are tried proves to be quite important. As you try to decide the order in which to choose variables, you will notice that variables at the extremities (the beginning or the end of a sequence) are less constrained than variables in the middle of the sequence. For that reason, you begin ordering the variables by choosing the one closest to the middle of the sequence.

If you do not order the values carefully, you cannot solve a car-sequencing problem within a reasonable amount of time, so you must pay attention to this part of the problem, too.

First, the most difficult constraints are tied to options. However, in a conventional backtracking problem, the basic decision is to assign a value to a variable; that is, to assign a *configuration* to a car. This practice is not tied directly to the difficult constraints. For example, suppose that option 1 is required by configurations 1, 2, and 3, and furthermore that you assign configuration 1 to a given car. There is no solution with this assignment because of the capacity of option 1. Consequently, the algorithm backtracks to try another

value, say, configuration 2 this time. But again, for the same reason—the limited capacity of the option—this trial also leads to failure. In other words, you will see thrashing, as values are tried and discarded for the same reason.

It is better to try another kind of value ordering: you first try to assign those options for which the demand is close to their capacity. To distinguish those options, the *slack* of an option with capacity pj/qj required to be scheduled kj times is defined like this:

$n - qj(kj/pj)$

In this context, slack is a good measure of how much an option is used. If the slack of an option is negative, the capacity constraint on the option cannot be satisfied. If the slack is large, however, the capacity constraint on the option is easy to satisfy.

The enumeration algorithm is a combination of these two ways of ordering variables and values; both depart from the conventional backtracking algorithm. Instead of choosing a variable and then trying a value for it, as you might conventionally do, you progressively *reduce the domains* of the variables. How do you do that? As the first option, you select the one with the least slack; then you try to assign it to the appropriate variables. To assign an option o to a variable x, you *remove* the configurations that do not use o from the *domain* of x. In other words, you generally reduce the domain.

More precisely, you select the option oj with the least slack. If this option is required kj times, you try to assign this option to k variables. To do so requires a tree search, where the variables that are closest to the middle of the sequence are tried first. If you succeed in assigning option o k times, you recursively call the algorithm with a new option. If not, you backtrack to the previous level of the algorithm.

As the algorithm backtracks, it propagates constraints at each node. However, these nodes are defined differently from those in a conventional backtracking algorithm, where branches correspond to the assignment of a value to a variable. In this case, a branch corresponds instead to the assignment of an *option* to a variable; that is, a reduction of the domain of the variable.

Moreover, a given variable may appear at different levels in the search tree, since its domain may be reduced more than once. To get a clearer idea of this, consider for example a configuration, say, $c1$ that requires option $o1$, along with the option $o2$ which is assigned to x in a given solution. In this example, x appears two times among the branches leading to that solution: once when $o2$ is assigned to x and again when $o1$ is assigned to it.

In that context, this algorithm tries first to consider the most constrained choices.

These ideas can be implemented straightforwardly by Boolean variables to represent whether a configuration assigned to a car does or does not require a given option. More formally, you use `nbOptions*nbCars` Boolean variables. The Boolean variable `o[i*nbCars+j]` is `IloTrue` if the car at the position i is assigned the configuration that

requires the option *j*; otherwise, the Boolean variable is `IloFalse`. Here is how you define those variables:

```
IloBoolVarArray bvars(env,nbOptions*nbCars);
for(i=0;i < nbOptions;i++){
  opt=(IloInt)order[i];
  IloBoolVarArray bv(env,nbCars);
  for (j = 0; j < nbCars; j++)
    bv[j] = IloBoolVar(env);
  model.add(IloBoolAbstraction(env, bv, cars, optConf[opt]));
  for(j=0;j<nbCars;j++){
    bvars[i*nbCars + j] = bv[j];
  }
}
cout << "Pass 4 " << endl;
```

In order to instantiate the variable at the middle of the sequence first, you simply reorder the array of Boolean variables. Here is the code to do that:

```
IloBoolVarArray tbvars(env,nbOptions*nbCars);
for(j=0;j<nbOptions*nbCars;j+=nbCars){
  IlcInt mid=nbCars/2 - 1;
  for(i=0;i<mid+1;i++){
    tbvars[2*i+j]=bvars[mid-i+j];
    tbvars[2*i+1+j]=bvars[mid+i+1+j];
  }
}
```

The order in which the options are chosen is given explicitly for this example. Here is the code:

```
IloIntArray order(env, nbOptions);
switch(example){
case 1:{
  order[OL]=0;
  order[1]=1;
  order[2]=4;
  order[3]=2;
  order[4]=3;
} break;
case 2:{
  order[OL]=0;
  order[1]=4;
  order[2]=1;
  order[3]=2;
  order[4]=3;
}break;
}
```

That order, of course, respects the definition of slack.

Writing the goal

You use the macro `ILCGOAL1` to define a new class of goals, `IlcGenerateAbstractVars`. This goal has one parameter, an array of integer variables. The body of the macro defines how to execute this goal. This goal instantiates each variable using `IlcDichotomize`. The function `IlcDichotomize creates` and returns a goal that is used to assign a value to a constrained variable. `IlcDichotomize` sets a choice point, then replaces the domain of the variable by one of the halves, and calls itself recursively. If failure occurs then, the domain is replaced by the other half, and `IlcDichotomize` is called recursively. The function `IlcAnd` creates a sequences of subgoals, one for each variable in the array. Here is the code to write the goal:

```
ILCGOAL1(IlcGenerateAbstractVars, IlcIntVarArray, vars) {
  IlcInt index = IlcChooseFirstUnboundInt(vars);
  if (index == -1) return 0;
  return IlcAnd(IlcDichotomize(vars[index]), this); // first tries vars[index]
== 1
}
```

You use the macro `ILOCPGOALWRAPPER` to wrap the goal `IlcGenerateAbstractVars` so it can be used in a Concert Technology model as the goal `IloGenerateAbstractVars`. Here is the code to wrap the goal

```
ILOCPGOALWRAPPER1(IloGenerateAbstractVars, solver, IloIntVarArray, vars) {
  return IlcGenerateAbstractVars(solver, solver.getIntVarArray(vars));
}
```

In the search, `IloSolver::solve` takes the goal `IloGenerateAbstractVars` as a parameter. The goal `IloGenerateAbstractVars` takes the array of variables `tbvars` and the environment as parameters. The filter levels for distribute and sequence constraints are both set to `IlcExtended`, the most thorough filter level. The search for solutions itself is carried out by the following code:

```
    IloSolver solver(model);
    if (mode) {
      solver.setDefaultFilterLevel(IlcSequenceCt,IlcExtended);
      solver.setDefaultFilterLevel(IlcDistributeCt,IlcExtended);
    }
    if (solver.solve(IloGenerateAbstractVars(env,tbvars)))
      solver.out() << "cars = " << solver.getIntVarArray(cars) << endl;
    else
    solver.out() << "No solution" << endl;
    solver.printInformation();
  }
```

Complete car sequencing program

The complete program follows. You can also view the entire program online in the file `YourSolverHome/examples/src/carseq.cpp`.

```
#include <ilsolver/ilosolverint.h>

ILOSTLBEGIN

//
// First Model
//


IloModel IloCarSequencing( IloEnv env,
                           IloInt maxCar,
                           IloIntArray maxConfs,
                           IloIntArray confs,
                           IloIntVarArray sequence){


  const IloInt abstractValue=-1; // we assume that confs[i] != -1
  const IloInt confs_size=confs.getSize();
  const IloInt sequence_size = sequence.getSize();
  IloIntArray nval(env, confs_size+1);
  IloIntVarArray ncard(env, confs_size+1);

  nval[OL]=abstractValue;
  ncard[OL]=IloIntVar(env, sequence_size - maxCar, sequence_size);

  IloInt i;
  for(i=0;i<confs_size;i++){
    nval[i+1]=confs[i];
    if (maxConfs[i] > maxCar)
      ncard[i+1]=IloIntVar(env, 0, maxCar);
    else
      ncard[i+1]=IloIntVar(env, 0, maxConfs[i]);
  }

  IloIntVarArray nvars(env,sequence_size);
  for (i = 0; i < sequence_size ; i++)
    nvars[i] = IloIntVar(env, nval);


  IloModel carseq_model(env);
  carseq_model.add(IloAbstraction(env, nvars, sequence, confs,abstractValue));
  carseq_model.add(IloDistribute(env, ncard, nval, nvars));
  return carseq_model;
}


void firstModel(){
  IloEnv env;
  try {
    IloModel model(env);

    const IloInt nbOptions = 5;
```

```
  const IloInt nbConfs   = 6;
  const IloInt nbCars    = 10;

  IloIntVarArray cars(env, nbCars,0,nbConfs-1);

  IloIntArray confs(env, 6, 0, 1, 2, 3, 4, 5);
  IloIntArray nbRequired(env, 6, 1, 1, 2, 2, 2, 2);

  IloIntVarArray cards(env, 6);
  for(IloInt conf=0;conf<nbConfs;conf++) {
    cards[conf]=IloIntVar(env, nbRequired[conf],nbRequired[conf]);
  }

  model.add (IloDistribute(env, cards, confs, cars));

  IloArray<IloIntArray> optConf(env, nbOptions);

  optConf[0] = IloIntArray(env, 3, 0, 4, 5);
  optConf[1] = IloIntArray(env, 3, 2, 3, 5);
  optConf[2] = IloIntArray(env, 2, 0, 4);
  optConf[3] = IloIntArray(env, 3, 0, 1, 3);
  optConf[4] = IloIntArray(env, 1, 2);

  IloIntArray maxSeq(env, 5, 1, 2, 1, 2, 1);
  IloIntArray overSeq(env, 5, 2, 3, 3, 5, 5);

  IloArray<IloIntArray> optCard(env, nbOptions);
  optCard[0] = IloIntArray(env, 3, 1, 2, 2);
  optCard[1] = IloIntArray(env, 3, 2, 2, 2);
  optCard[2] = IloIntArray(env, 2, 1, 2);
  optCard[3] = IloIntArray(env, 3, 1, 1, 2);
  optCard[4] = IloIntArray(env, 1, 2);

  for (IloInt opt=0; opt < nbOptions; opt++) {
    for (IloInt i=0; i < nbCars-overSeq[opt]+1; i++) {
      IloIntVarArray sequence(env,(IloInt)overSeq[opt]);
      for (IloInt j=0; j < overSeq[opt]; j++)
        sequence[j] = cars[i+j];
      model.add(IloCarSequencing(env,
                                 (IloInt)maxSeq[opt],
                                 optCard[opt],
                                 optConf[opt],
                                 sequence));
    }
  }


  IloSolver solver(model);

  solver.solve(IloGenerate(env,cars,IlcChooseMinSizeInt));
  solver.out() << "cars = " << solver.getIntVarArray(cars) << endl;

  solver.printInformation();
}
catch (IloException& ex) {
  cout << "Error: " << ex << endl;
}
env.end();
```

```
}

// Second Model
//


IloArray<IloIntArray> readData(IloEnv env,
                               IloInt example,
                               IloInt& nbCars,
                               IloInt& nbOpt,
                               IloIntArray& nbMax,
                               IloIntArray& seqWidth,
                               IloIntArray& confs,
                               IloIntArray& nbCarsByConf){
  char globalName[200];
  switch(example){
  case 1:{
    strcpy(globalName,"../../../examples/data/carseq1.dat");
  } break;
  case 2:{
    strcpy(globalName,"../../../examples/data/carseq2.dat");
  } break;
  }
  IloInt nbConfs;
  ifstream fin(globalName,ios::in);
  if (!fin) env.out() << "problem with file:" << globalName << endl;
  fin >> nbCars >> nbOpt >> nbConfs;

  IloInt i;
  confs = IloIntArray(env, nbConfs); // required configurations
  for(i=0;i<nbConfs;i++){
    confs[i]=i;
  }
  IloArray<IloIntArray> confsByOption(env, nbConfs);
  for (i=0;i<nbConfs;i++){
    confsByOption[i] = IloIntArray(env, nbOpt);
  }
  IloIntArray nbConfsByOption(env, nbOpt);
  for (i=0;i<nbOpt;i++){
    nbConfsByOption[i]=0;
  }

  // read the maximum number of cars of each sequence that can take the
  // invoked option
  nbMax=IloIntArray(env, nbOpt);
  for(i=0;i<nbOpt;i++){
    fin >> nbMax[i];
  }

  // read the size of a sequence for each option
  seqWidth=IloIntArray(env, nbOpt);
  for(i=0;i<nbOpt;i++){
    fin >> seqWidth[i];
  }

  // read the options required by each configuration
  nbCarsByConf=IloIntArray(env, nbConfs);
  IloInt dummy;
```

```
  IloInt j;
  for(i=0;i<nbConfs;i++){
    fin >> dummy;
    fin >> nbCarsByConf[i];
    for (j=0;j<nbOpt;j++){
      fin >> confsByOption[i][j];
      if (confsByOption[i][j] == 1){
        nbConfsByOption[j]++;
      }
    }
  }
  // compute the configurations required by each option
  IloArray<IloIntArray> optConf(env, nbOpt);
  for(i=0;i<nbOpt;i++){
    optConf[i]=IloIntArray(env, (IloInt)nbConfsByOption[i]);
  }
  IloIntArray ind(env, nbOpt);
  for(i=0;i<nbOpt;i++){
    ind[i]=0;
  }
  for(i=0;i<nbConfs;i++){
    for (j=0;j<nbOpt;j++){
      if (confsByOption[i][j] == 1){
        optConf[j][(IloInt)ind[j]]=i;
        ind[j]++;
      }
    }
  }
  // print the configuration required by each option
  env.out() << "number of times each configuration is required:" << endl;
  for(i=0;i<nbConfs;i++){
    env.out() << nbCarsByConf[i] << " ";
  }
  env.out() << endl;
  env.out() << "configuration required by each option:" << endl;
  for(i=0;i<nbOpt;i++){
    env.out() << "option:" << i << " ";
    for (j=0;j<optConf[i].getSize();j++){
      env.out() << optConf[i][j] << " ";
    }
    env.out() << endl;
  }
  env.out() << "number of times each option is required" << endl;
  for(i=0;i<nbOpt;i++){
    IloInt cpt=0;
    for(j=0;j<optConf[i].getSize();j++){
      cpt += nbCarsByConf[(IloInt)optConf[i][j]];
    }
    env.out() << "option:" << i << " " << cpt << " x ";
    env.out() << nbMax[i] << "/" << seqWidth[i] << endl;
  }

  return optConf;
}

ILCGOAL1(IlcGenerateAbstractVars, IlcIntVarArray, vars) {
  IlcInt index = IlcChooseFirstUnboundInt(vars);
  if (index == -1) return 0;
```

```
  return IlcAnd(IlcDichotomize(vars[index]), this); // first tries vars[index]
== 1
}

ILOCPGOALWRAPPER1(IloGenerateAbstractVars, solver, IloIntVarArray, vars) {
  return IlcGenerateAbstractVars(solver, solver.getIntVarArray(vars));
}

void secondModel(IloInt mode, IloInt example)
{
  IloEnv env;
  try {
    IloModel model(env);

    IloInt nbOptions;
    IloInt nbCars;
    IloIntArray confs(env); // array of required configurations
    IloIntArray nbCarsByConf(env); // number of cars to assign
                                   // to each configuration
    IloIntArray nbMax(env);
    IloIntArray seqWidth(env);

    IloArray<IloIntArray> optConf = readData(env,
                                             example,
                                             nbCars,
                                             nbOptions,
                                             nbMax,
                                             seqWidth,
                                             confs,
                                             nbCarsByConf);


    cout << "Pass 1 " << endl;
    const IlcInt nbConfs   = confs.getSize();

    IloInt i,j;
    IloIntVarArray cars(env,nbCars,0,nbConfs-1);
    IloIntVarArray cards(env,nbConfs);
    for (i = 0; i < nbConfs; i++){
      cards[i] = IloIntVar(env,nbCarsByConf[i],nbCarsByConf[i]);
    }
    cout << "Pass 2 " << endl;

    model.add(IloDistribute(env, cards, confs, cars));

    IloInt opt;
    for (opt = 0; opt < nbOptions; opt++) {
      IloIntVarArray ncard(env,optConf[opt].getSize());
      for(i = 0; i < optConf[opt].getSize(); i++){
        ncard[i] = cards[(IloInt)optConf[opt][i]];
      }
      model.add(IloSequence(env,
                            0,
                            (IloInt)nbMax[opt],
                            (IloInt)seqWidth[opt],
                            cars,
                            optConf[opt],
                            ncard));
```

```
    }
    cout << "Pass 3 " << endl;
    IloIntArray order(env, nbOptions);
    switch(example){
    case 1:{
      order[OL]=0;
      order[1]=1;
      order[2]=4;
      order[3]=2;
      order[4]=3;
    } break;
    case 2:{
      order[OL]=0;
      order[1]=4;
      order[2]=1;
      order[3]=2;
      order[4]=3;
    }break;
    }

    IloBoolVarArray bvars(env,nbOptions*nbCars);
    for(i=0;i < nbOptions;i++){
      opt=(IloInt)order[i];
      IloBoolVarArray bv(env,nbCars);
      for (j = 0; j < nbCars; j++)
        bv[j] = IloBoolVar(env);
      model.add(IloBoolAbstraction(env, bv, cars, optConf[opt]));
      for(j=0;j<nbCars;j++){
        bvars[i*nbCars + j] = bv[j];
      }
    }
    cout << "Pass 4 " << endl;

    IloBoolVarArray tbvars(env,nbOptions*nbCars);
    for(j=0;j<nbOptions*nbCars;j+=nbCars){
      IlcInt mid=nbCars/2 - 1;
      for(i=0;i<mid+1;i++){
        tbvars[2*i+j]=bvars[mid-i+j];
        tbvars[2*i+1+j]=bvars[mid+i+1+j];
      }
    }

    IloSolver solver(model);
    if (mode) {
      solver.setDefaultFilterLevel(IlcSequenceCt,IlcExtended);
      solver.setDefaultFilterLevel(IlcDistributeCt,IlcExtended);
    }
    if (solver.solve(IloGenerateAbstractVars(env,tbvars)))
      solver.out() << "cars = " << solver.getIntVarArray(cars) << endl;
    else
    solver.out() << "No solution" << endl;
    solver.printInformation();
  }
  catch (IloException& ex) {
    cout << "Error: " << ex << endl;
  }
  env.end();
}
```

```
int main(int argc, char** argv){
  IloInt pb=(argc>1)?atoi(argv[1]):1;
  if (pb != 1 && pb != 2)
    pb=1;

  switch(pb){
  case 1 :{
    firstModel();
  } break;
  case 2 :{
    IlcInt mode = (argc>2)?atoi(argv[2]):1;
    if (mode != 0 && mode != 1)
      mode=1;
    IlcInt example = (argc>3)?atoi(argv[3]):1;
    secondModel(mode,example);
  } break;

  }

  return 0;
}
```

Output

When you run this program, you get results like this:

```
carseq 1:
cars = IlcIntVarArrayI[[0] [1] [5] [2] [4] [3] [3] [4] [2] [5]]
Number of fails               : 1
Number of choice points       : 3
Number of variables           : 308
Number of constraints         : 239
Reversible stack (bytes)      : 40224
Solver heap (bytes)           : 168864
Solver global heap (bytes)    : 4044
And stack (bytes)             : 4044
Or stack (bytes)              : 4044
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 11144
Total memory used (bytes)     : 236408
Elapsed time since creation   : 0.11

carseq 2 1 1:
number of times each configuration is required:
5 3 7 1 10 2 11 5 4 6 12 1 1 5 9 5 12 1
configuration required by each option:
option:0 0 1 2 4 5 6 7 13
option:1 0 1 2 3 4 8 9 10 14
option:2 2 3 7 10 11 12 17
option:3 1 3 6 9 12 16
option:4 0 5 8 11 15
number of times each option is required
option:0 48 x 1/2
option:1 57 x 2/3
```

```
option:2 28 x 1/3
option:3 34 x 2/5
option:4 17 x 1/5
cars = IlcIntVarArrayI[[17] [16] [16] [7] [15] [13] [10] [13] [14] [6] [10] [6]
[15] [4] [14] [6] [10] [4] [16] [0] [14] [6] [10] [1] [15] [2] [14] [6] [10]
[0]
 [16] [4] [9] [7] [8] [4] [16] [2] [9] [5] [10] [4] [16] [2] [8] [6] [14] [2]
[1
6] [0] [3] [13] [14] [2] [15] [1] [10] [6] [14] [0] [12] [4] [9] [7] [8] [4]
[16
] [2] [9] [5] [10] [4] [16] [2] [8] [6] [10] [4] [16] [0] [10] [6] [14] [1]
[11]
 [4] [9] [7] [9] [13] [10] [6] [14] [6] [10] [13] [15] [7] [16] [16]]
Number of fails              : 0
Number of choice points      : 160
Number of variables          : 4768
Number of constraints        : 2367
Reversible stack (bytes)     : 836184
Solver heap (bytes)          : 1977864
Solver global heap (bytes)   : 4044
And stack (bytes)            : 4044
Or stack (bytes)             : 8064
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 15152
Total memory used (bytes)    : 2849396
Elapsed time since creation  : 4.677

carseq 2 1 2
number of times each configuration is required:
7 11 1 3 15 2 8 5 3 4 5 2 6 2 2 4 3 5 2 4 1 1 1 1 2
configuration required by each option:
option:0 0 1 3 5 10 11 15 16 17 18 19 20 21
option:1 1 2 4 6 9 11 12 14 15 17 18 19 21 22 23
option:2 2 3 5 7 9 11 12 13 15 18 23
option:3 0 2 5 6 8 9 15 17 20 21 22
option:4 2 11 13 14 16 19 20 21 22 23 24
number of times each option is required
option:0 50 x 1/2
option:1 67 x 2/3
option:2 32 x 1/3
option:3 37 x 2/5
option:4 20 x 1/5
No solution
Number of fails              : 41
Number of choice points      : 40
Number of variables          : 626
Number of constraints        : 0
Reversible stack (bytes)     : 872364
Solver heap (bytes)          : 2126604
Solver global heap (bytes)   : 4044
And stack (bytes)            : 4044
Or stack (bytes)             : 4044
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 21160
Total memory used (bytes)    : 3036304
Elapsed time since creation  : 5.458
```

C H A P T E R

19

Writing a Constraint: Allocating Frequencies

In this lesson, you will learn how to:

- understand constraints, metaconstraints, and demons
- write your own constraints and metaconstraints
- model and solve a frequency allocation problem using a custom constraint

In Part I, *The Basics*, you learned in general terms how Solver uses constraint propagation. This lesson explains in greater detail how constraints work. You will learn how to write your own constraints and metaconstraints.

You will also learn how to design a model and solve a problem in frequency allocation using a custom constraint and custom goals. With modern radio communication systems, using available frequencies more effectively has become a major challenge. Ordinarily, the spectrum of available frequencies for a specific application is limited, and a radio operator must pay for the use of each frequency. The radio operator, therefore, has to exploit this resource to maximize the possible traffic and minimize the number of needed frequencies.

When frequencies are allocated, physical constraints also have to be taken into account, such physical constraints as interference between broadcasts on adjacent frequencies. These physical constraints can be expressed as a minimum “distance” that must be respected between two frequencies.

Understanding constraints

As indicated in other chapters, Solver offers a wide range of predefined constraints, together with powerful logical operators for combining them. These facilities usually suffice for expressing even the most specific constraints.

However, when Solver treats a group of heterogeneous constraints together as a set, it still must deal with each of them locally. One way to offset this “localness” about the way constraints are treated is to use redundant constraints. See “Introduce redundant constraints” on page 590. Another—more radical approach—is to write a new constraint. This alternative may at times be somewhat more difficult.

When you write a new constraint linking certain variables, you must implement reductions of the domains of these variables by eliminating any values that could not participate in a solution. These reductions are carried out by means of *elementary modifiers* on the domains of these variables.

This chapter explains what you must do if you decide to implement a new constraint. This section firsts explain the *elementary modifiers* involved, and then discusses the reduction function itself. Reductions happen according to the type of modification that occurs in the domain of a variable. A complete example of how to write a new constraint is provided. See “Writing your own constraint” on page 337. “Writing your own metaconstraint” on page 343 explains how to write metaconstraints, constraints that can be imposed on other constraints.

Elementary modifiers for variables

Besides the *accessors* such as `getMin`, `getMax`, or `getValue`, defined for integer expressions, there are also predefined *elementary modifiers*. You can use these modifiers to implement constraints.

Let’s look, for example, at how `setValue(IlcInt c)`, one of these predefined modifiers, behaves.

```
IlcIntVar x(solver,0,1);
x.setValue(1);
```

That fragment of code effectively makes the domain of `x` equal to a singleton (a single element) reduced to the value 1. This behavior literally involves a domain reduction, and this behavior is consistent throughout Solver: if the value 1 were not in the initial domain, Solver would have raised an error.

For example, consider the following contrasting piece of code. It raises an error, `fail`.

```
IlcIntVar x(solver,0,1);
x.setValue(2);       // LEADS TO FAILURE
```

The point here is that elementary modifiers can be used to implement constraints, but they themselves do not behave like constraints.

It's also important to note that modifiers are *volatile*. By "volatile," it is meant that if a modifier does not have an immediate effect, it will have no later effect. This volatility is another way in which elementary modifiers differ from constraints.

```
IlcIntVar x(solver,0,1), y(solver,0,1);
(x + y).setValue(1);        // NO EFFECT
```

That code has no effect because all the values of the domains can participate in a solution.

Of course, if you use constraints instead, and post them, like this, those hazards can be avoided.

```
IlcIntVar x(solver,0,1), y(solver,0,1);
solver.add( x + y == 1);
```

It's true that this piece of code will not modify a domain, but during the generation of the values of `x` and `y`, only the two solutions where the sum is equal to 1 appear. What's happening here is that Solver is saving the constraint so at each modification of the domain of either of the variables, Solver executes the corresponding modifier `setValue`.

For the class `IlcIntExp`, here are the modifiers and what they do.

- `setValue(IlcInt value)` tries to make the expression equal to `value`.
- `setMin(IlcInt min)` tries to make the expression greater than or equal to `min`.
- `setMax(IlcInt max)` tries to make the expression less than or equal to `max`.
- `setRange(IlcInt min, IlcInt max)` tries to make the expression stay between `min` and `max`, inclusive.
- `removeValue(IlcInt value)` tries to make the expression different from `value`.
- `removeInterval(IlcInt min, IlcInt max)` tries to make the expression strictly less than `min` or strictly greater than `max`. In other words, it tries to make the expression stay *outside* the interval defined by `min` and `max`, inclusive.

Before leaving the topic of modifiers, it must be emphasized again that they only reduce domains: they don't enlarge domains. For that reason, the following code modifies nothing in the domain of `x` and, in particular, it does *not* make the upper boundary of x equal to 2.

```
IlcIntVar x(solver,0,1);
x.setMax(2);          // DOES NOTHING!
```

Exp and var: More about modifiers and expressions

Now that you have an idea about elementary modifiers and their effects on constrained integer variables, here's yet another way of understanding how elementary modifiers work. In this case, the point is illustrated by contrasting constrained *expressions* (where the

domains are *computed*) with constrained *variables* (where the domains are *stored*), a contrast that mentioned briefly in the *IBM ILOG Solver Reference Manual* in the documentation for constrained integer variables and constrained enumerated variables.

This contrast between the computed domains of constrained expressions and the stored domains of constrained variables also highlights a major difference between modifiers (like `setValue`) and constraints.

Consider this piece of code:

```
IlcIntVar x (solver, 0, 10);
IlcIntVar y (solver, 0, 10);
IlcIntExp z = x+y;
z.setRange(0, 10);
cout << z.getMax() << endl;
```

It prints 20 as its result, not 10 as you might expect. This happens because Solver computes the domain of the expression when it prints `z`; it does not store a domain for an expression. For that reason, the effect of an elementary modifier like `setRange` is volatile.

Moreover, if you later bind `x` to, say, 10, nothing happens to `y`! To see that happen for yourself, consider this slightly different piece of code:

```
IlcIntVar x (solver, 0, 10);
IlcIntVar y (solver, 0, 10);
IlcIntExp z = x+y;
z.setRange(0, 10);
x.setValue(10);
cout << y << endl;
```

It prints `[0 .. 10]`. Why? It does so because you used a *modifier*, `setRange`, so 10, the maximum, is *not stored* in `z`.

The correct code (correct in the sense of propagating values) must use a *constraint* (not a modifier), like this:

```
IlcIntVar x (solver, 0, 10);
IlcIntVar y (solver, 0, 10);
IlcIntExp z = x+y;
solver.add(z <= 10);
x.setValue(10);
cout << y << endl;
```

That piece of code stores the constraint `(z <= 10)` and thus binds `y` to 0.

So far, you've noticed that `z` is a constrained integer *expression*. Another way of "correcting" this is to make `z` a constrained integer *variable*. As mentioned in the *IBM ILOG Solver Reference Manual*, the domains of constrained expressions are not stored; they are computed. In contrast, the domains of constrained variables are stored.

Thus if `z` is a constrained *variable*, `y` is bound to 0, as in this piece of code:

```
IlcIntVar x (solver, 0, 10);
IlcIntVar y (solver, 0, 10);
IlcIntVar z = x+y;
z.setRange(0, 10);
x.setValue(10);
cout << y << endl;
```

However, the best way of writing (that is, the officially recommended way, supported by the Solver development team) is to use constrained variables and constraints, like this:

```
IlcIntVar x (solver, 0, 10);
IlcIntVar y (solver, 0, 10);
IlcIntVar z = x+y;
solver.add(z <= 10);
x.setValue(10);
cout << y << endl;
```

In that case, the constraint is created, though it is not stored because it is a *unary* constraint on a variable. This is the only case where constraints are not stored.

Let's summarize the conventions that Solver observes about constrained variables, constrained expressions, elementary modifiers, and constraints.

- In the sense that the domain of an expression is computed from the domains of its subexpressions, an expression has no domain itself. For that reason, it costs less memory.
- You cannot access the domain-delta (that is, the old min, the old max) of an expression because there is no domain for storing that information.
- Constraints on expressions are always stored, again because there is no stored domain for the expression.
- Assigning an expression to a variable creates a domain for the expression. The expression is a variable in fact, and a constraint is automatically and internally posted between that new variable and the original subexpressions.

On the basis of those facts, here are a few recommendations to follow in your programming. Of course, there may be good and overriding reasons not to follow these practices in certain special cases, but in general these recommendations are sound.

- Use variables rather than expressions. Exception: when you are defining a new constraint yourself, you may need expressions.
- Use constraints rather than modifiers. Exception: when you are defining a new constraint yourself, you may need modifiers.
- Use modifiers in these special situations:
  - to define a new constraint;
  - to code the virtual function, `propagate`;

- to code any demons of a new constraint that you write yourself (if any such demons are needed).

◆ Use expressions only in the definition of a new constraint simply because expressions avoid the creation of a domain.

◆ Don't use expressions in the definition of a new constraint if you want to use the domain-delta.

Invariants: What to propagate?

A constraint that links variables is an object with a task: that of reducing the domains of those variables in terms of the semantics of those variables. For example, once you have posted the constraint `x <= y`, that constraint must insure that the values in the domains of `x` and `y` satisfy the constraint. From a working point of view, that constraint must eliminate values from the domains of the variables, values that are definitely inconsistent.

When you are writing a constraint, it is a good idea to start by defining the *invariant* of that constraint before you implement it by means of elementary modifiers. (The term invariant is used here in the usual sense of software engineering.)

Let's look again at the example of `x <= y` in this context. Its invariant is easy to determine, and can be expressed this way:

◆ the values in the domain of x must be less than or equal to the maximum of the values in the domain of y.

◆ the values in the domain of y must be greater than or equal to the minimum of the values in the domain of x.

With elementary modifiers, this invariant can be translated this way:

```
x.setMax(y.getMax());
y.setMin(x.getMin());
```

The next section shows what Solver does with such an invariant.

The constraint propagation algorithm

To see how the propagation algorithm works with constraints, let's look again at the constraint `x <= y`, and consider the following fragment of code:

```
IlcIntVar x(solver,0,3), y(solver,0,2);
solver.add( x <= y );
```

When that constraint is posted, the invariant expressed in the previous section becomes active and reduces the domain of `x` by removing the value 3. For the moment, that's all that you can deduce about that constraint. Since that constraint has to be taken into account by

Solver every time one of the variables in it is modified, the constraint itself is physically attached to these two variables.

Solver was designed specifically to automate and to optimize the reduction of the domains of constrained variables. The Solver algorithm for that purpose is straightforward in principle. When a constrained variable is modified, the constraints posted on that variable are examined to determine whether any values in the domains of other constrained variables are now inconsistent. If this is the case, necessary domain reductions are carried out in turn.

The examination of the constraints on a variable is triggered by any modification of that variable. There are several different kinds of modifications, depending on the class of variable under consideration. Those modifications are referred to as *propagation events*.

For the class of integer variables, there are, in fact, these propagation events:

- *value* means that a value has been assigned to the constrained variable, that is, the variable has been bound.
- *range* indicates that the minimum of the domain has increased or the maximum of the domain has decreased.
- *domain* indicates that the domain of the constrained variable has been modified.

When you define a new class of constraint, you must also define the propagation events for that class of constraint. You do so by means of the pure virtual member function, `post`.

Sometimes more than one event can be triggered after a variable modification. Specifically, the value event is always accompanied by the range and domain events. Likewise, a range event is always accompanied by a domain event.

To take another example, let's consider a constrained enumerated variable, `var`, with a domain containing only two values, `value1` and `value2`, where `value1 < value2`. If you post a constraint like this `solver.add(var != value1)`, three events are triggered:

- the domain event is triggered since `value1` is actually removed from the domain of `var`;
- the range event is triggered since the minimal boundary of the domain of `var` has been increased.
- the value event is triggered since the variable is bound by the reduction of its domain to one value.

Writing your own constraint

Obviously, a constraint is an object in Solver. More precisely, a constraint in Solver is an instance of a class with two pure virtual functions, `propagate` and `post`, and a third virtual function, `isViolated`. The virtual function `propagate` implements the invariant of the constraint; `post` defines on which events `propagate` executes; and `isViolated` returns

IlcTrue if the constraint cannot be satisfied. The virtual function isViolated will be called if the constraint is to be used as a metaconstraint.

As indicated, Solver lets you define new *classes* of constraints. You do so by defining a subclass of IlcConstraintI. (See the section “Implementation classes and handles” on page 340 for an explanation of the I at the end of the name.) The data members of this subclass include, among others, the constrained variables on which the constraint is posted.

Consequently, the definition of a new class of constraint looks something like this:

```
class MyConstraint : public IlcConstraintI {
     ... // parameters for the constraint
public :
    MyConstraint(IloSolver solver, ... args ...); // constructor
    ~MyConstraint(){} // destructor; usually empty
    void post();
    void propagate();
}
```

To clarify how to define a new class of constraint, let’s look at the constraint x != y. The parameters of this constraint are obviously the expressions x and y themselves.

```
class IlcDiffConstraint :public IlcConstraintI {
    IlcIntExp _x, _y;
  public:
    IlcDiffConstraint(IloSolver solver, IlcIntExp x, IlcIntExp y);
    ~IlcDiffConstraint(){}; // empty destructor

    virtual void propagate ();
    virtual void post();
    virtual IlcBool isViolated() const;
};
```

The role of the constructor is to initialize the data members, like this:

```
IlcDiffConstraint::IlcDiffConstraint(IloSolver solver,
                                     IlcIntExp x,
                                     IlcIntExp y)
    : IlcConstraintI(solver), _x(x), _y(y) {}
```

The invariant of x != y is easy to determine, as indicated earlier:

- if x is bound to a value, the domain of y does not contain this value;
- likewise, if y is bound to a value, the domain of x does not contain this value.

The virtual member function `propagate` implements the reduction of the domains. Using what you've already written, you would get this:

```
void IlcDiffConstraint::propagate () {
    if (_x.isBound()) _y.removeValue(_x.getValue());
    if (_y.isBound()) _x.removeValue(_y.getValue());
}
```

The second virtual member function to write is `post`. It connects the constraint to its arguments by specifying the events that set it off. For integer expressions, there are three kinds of attachment, one kind for each type of event. For the class `IlcIntExp`, these member functions are as follows: (The class `IlcConstraint` derives from the class `IlcDemon`.)

- `whenValue(IlcDemon ct)` attaches the demon `ct` to the value event of the expression under consideration. Every time the expression takes a unique value, the `execute` function for `ct` is called.
- `whenRange(IlcDemon ct)` attaches the demon `ct` to the range event of the expression under consideration. Every time the domain of the expression gets a new boundary, the `execute` function for `ct` is called.
- `whenDomain(IlcDemon ct)` attaches the demon `ct` to the domain event. Every time the domain of the expression is modified, the `execute` function for `ct` is called.

For a constraint, the `execute` member function calls `propagate`.

In our example of the constraint `x != y`, you want to propagate the constraint every time `x` or `y` is bound. Here's the code for doing that:

```
void IlcDiffConstraint::post(){
    _x.whenValue(this);
    _y.whenValue(this);
}
```

If you want to use our constraint as a metaconstraint, you may want to define under what conditions `IlcDiffConstraint` is definitely not satisfied. In our example, `x != y` is violated if, and only if, variables `x` and `y` are bound to the same value. Hence, the definition of `isViolated`:

```
IlcBool IlcDiffConstraint::isViolated() const {
    return (_x.isBound() && _y.isBound() &&
            _x.getValue()==_y.getValue());
}
```

Implementation classes and handles

Throughout this manual, it has been taken for granted that the major entities of Solver—variables and constraints themselves, for example—are simply *objects* in the full sense of C++. In fact, Solver entities depend on two classes: a *handle* class and an *implementation* class, where an object of the handle class points to an object of the corresponding implementation class. As documented in the *IBM ILOG Solver Reference Manual*, handles are passed by value, and they are created as automatic objects, where “automatic” has the usual C++ meaning.

For the most part, you will work only with handles, that is, the objects that are really pointers to corresponding implementation objects on the Solver heap. Solver takes care of all the allocation, memory-management, and de-allocation of these internal implementation objects, keeping the details of these implementation classes “out of sight.” However, if for some reason, you need to define new classes of Solver objects, you need to do so in a two-step process: you’ll have to define the underlying implementation class plus the corresponding class of handles. This chapter explains how to do that for constraints.

For a predefined Solver class, `IlcConstraintI`, implementing an object with its own data members and virtual functions, Solver defines the handle class `IlcConstraint`. Essentially, such a handle class contains objects that are really only pointers to instances of the corresponding implementation class `IlcConstraintI`.

This has the practical effect of elevating the idea of a pointer to the level of an object itself. A number of advantages derive from this; for one thing, it provides greater certainty about how pointers are used. In particular, there is no longer any risk that the compiler will mistakenly interpret the expression `x+1` as a pointer and apply pointer arithmetic to it. Instead, pointers are consistently and systematically treated as objects.

However, the fundamental precepts of Solver that there are “No more pointers” and that “Everything is an object” mean that when you extend Solver by defining a new implementation class, you must also define a function that returns a handle.

This rule applies particularly to the definition of new classes of constraints. Up to now, you’ve seen only the definition of a new *implementation* class of a new constraint. The definition of an implementation class generally looks something like this:

```
class MyConstraint : public IlcConstraintI {
     ... // parameters for the constraint
public :
    MyConstraint(IloSolver solver, ... args ...); // constructor
    ~MyConstraint(){} // destructor; usually empty
    void post();
    void propagate();
}
```

Handles for constraints are instances of the class `IlcConstraint`. That class provides a constructor of an implementation class, or more precisely, `IlcConstraintI*`. You must define a function that returns a handle, and the minimal service that the handle provides is to manage the memory allocation of instances in your implementation class. The form that it generally takes looks like this:

```
IlcConstraint MyConstraint( /* parameters for the constraint */){
        // get the solver from a variable
        return new (solver.getHeap()) MyConstraintI(solver, /* ...args... */);
}
```

The body of that function builds an instance of `MyConstraint` by calling the constructor and allocates a place on the Solver heap by calling the overloaded operator, `new (solver.getHeap() ... )`. When you use this Solver `new`, you'll get an efficient allocator that automatically manages how space is recovered in case of backtracking.

In our example, `x != y`, you call `IlcDiff` the function for creating the constraint. Here's the code for it:

```
class IlcDiffConstraint :public IlcConstraintI {
    IlcIntExp _x, _y;
  public:
    IlcDiffConstraint(IloSolver solver, IlcIntExp x, IlcIntExp y);
    ~IlcDiffConstraint(){}; // empty destructor

    virtual void propagate ();
    virtual void post();
    virtual IlcBool isViolated() const;
};

IlcDiffConstraint::IlcDiffConstraint(IloSolver solver,
                                     IlcIntExp x,
                                     IlcIntExp y)
    : IlcConstraintI(solver), _x(x), _y(y) {}


void IlcDiffConstraint::propagate () {
    if (_x.isBound()) _y.removeValue(_x.getValue());
    if (_y.isBound()) _x.removeValue(_y.getValue());
}

void IlcDiffConstraint::post(){
    _x.whenValue(this);
    _y.whenValue(this);
}

IlcBool IlcDiffConstraint::isViolated() const {
    return (_x.isBound() && _y.isBound() &&
            _x.getValue()==_y.getValue());
}

IlcConstraint IlcDiff(IlcIntExp x, IlcIntExp y){
  IloSolver solver=x.getSolver();
    return new (solver.getHeap()) IlcDiffConstraint(solver, x, y);
}
```

That definition, of course, has two parts: the implementation and the handle. To use this new constraint, you simply write:

```
    solver.add(IlcDiff(x, y));
```

That statement invokes all the mechanisms of Solver that insure what's taken into account, such mechanisms as propagation, backtracking, and so forth. In general, it can be said that a call to `IlcDiff` as you've written it builds an instance of `IlcDiffConstraint` and returns a handle for that object (in other words, an encapsulation of the object) to post by means of the `post` member function. The instance of the implementation class is connected to the value event of `x` and of `y`. Every time these variables are bound, the `propagate` member function executes.

The definition just outlined is the *minimal* definition for a new constraint. It is not adequate if you want to use the new definition as a metaconstraint.

Using ILOCPCONSTRAINTWRAPPER to wrap constraints

You can use the macro `ILOCPCONSTRAINTWRAPPER` to wrap an existing instance of `IlcConstraint` when you want to use it within Concert Technology model objects.

This macro defines a constraint class named `_thisI` with *n* data members. When *n* is greater than zero, the types and names of the data members must be supplied as arguments to the macro. Each data member is defined by its type `t`*i* and a name `a`*i*.

```
ILOCPCONSTRAINTWRAPPER0(_this, solver)
ILOCPCONSTRAINTWRAPPER1(_this, solver, t1, a1)
ILOCPCONSTRAINTWRAPPER2(_this, solver, t1, a1, t2, a2)
ILOCPCONSTRAINTWRAPPER3(_this, solver, t1, a1, t2, a2, t3, a3)
ILOCPCONSTRAINTWRAPPER4(_this, solver, t1, a1, t2, a2, t3, a3, t4, a4)
```

In order to use an instance of `IlcConstraint` in this way, you need to follow these steps:

1. Use the macro to wrap the instance of `IlcConstraint` in an instance of `IloConstraint`.
2. Extract your model and model objects for an instance of `IloSolver` by calling the member function `IloSolver::extract`. During extraction, `IloSolver::extract` will put back the instance of `IloConstraint`.

You must use the following `IloCPConstraintI` member functions to force extraction of an extractable or an array of extractables:

```
void use(const IloSolver solver, const IloExtractable ext)const;
void use(const IloSolver solver, const IloExtractableArray extArray)const;
```

For more information on wrapping constraints for use with Concert Technology, see the documentation for the macro `ILOCPCONSTRAINTWRAPPER` and the class `IloCPConstraintI` in the *IBM ILOG Solver Reference Manual*. The section “Using the constraint wrapper” on page 357 shows you how to do this for the frequency allocation example.

Writing your own metaconstraint

If you want to define a new metaconstraint, that is, a constraint on constraints, you’ll need to define additional virtual functions (besides those you just defined for a new constraint: `propagate` and `post`). Those additional virtual functions are `isViolated`, `makeOpposite` and `metaPostDemon`. This section shows you how to define `makeOpposite` and `metaPostDemon`.

Let's assume that you want to express the idea that the variable `x` is different from some other variable, `y`, or that `y` is different from `z`. You can write that very simply by using the logical *or* operator provided by Solver.

```
solver.add(IlcDiff(x, y) || IlcDiff(y, z));
```

Once one of those two constraints proves false, Solver makes the other one true. To do so, Solver obviously needs to know the truth value of these constraints. In fact, Solver needs only to know when one of these constraints is violated. To detect it, this *disjunctive* constraint has to be posted on the variables that are involved. This is the task of the `metaPostDemon` member function.

```
void IlcDiffConstraint::metaPostDemon(IlcDemonI* ct){
    _x.whenValue(ct);
    _y.whenValue(ct);
}
```

Now, if you write:

```
solver.add(IlcDiff(x, y) || IlcDiff(y, z));
```

`metaPostDemon` is called with the disjunctive constraint as its argument. Thus, you get the effect that once one of the two constraints proves false (that is, `isViolated` returns `IlcTrue`), Solver invokes the `propagate` function of the other one, insuring that the disjunctive constraint is taken into account.

There's still one more point to consider. Let's assume now that you want to affirm that the constraint `IlcDiff(x, y)` is false; in other words, that `x == y` is true. You can use the negation operator to write this:

```
solver.add(!IlcDiff(x, y));
```

But if you do so, how will Solver know that you want to impose the negation of a specific constraint? You have to tell Solver explicitly what the opposite constraint is, and that is the work of the member function `makeOpposite`.

In our example, the opposite constraint of `x != y` is obviously `x == y`. Remember that `x == y` is a *handle*. Indeed, it's an instance of the class `IlcConstraint`. With Solver, you get the object implementation from the `getImpl()` member function of the handle class. For example, `(x == y).getImpl()` returns the pointer to the implementation of `x == y`.

Thus, the code is:

```
IlcConstraintI* IlcDiffConstraint::makeOpposite() const {
    return (_x == _y).getImpl();
}
```

Finally, here's the code of the *metaconstraint* `IlcDiff`. (First, you'll see the definitions of the appropriate class and its member functions; then `IlcDiff` itself near the end.)

```
class IlcDiffConstraint :public IlcConstraintI {
    IlcIntExp _x, _y;
  public:
    IlcDiffConstraint(IloSolver solver, IlcIntExp x, IlcIntExp y);
    ~IlcDiffConstraint(){}; // empty destructor

    virtual void propagate ();
    virtual void post();
    virtual IlcBool isViolated() const;
    virtual IlcConstraintI* makeOpposite() const;
    virtual void metaPostDemon(IlcDemonI* expr);
};

IlcDiffConstraint::IlcDiffConstraint(IloSolver solver,
                                     IlcIntExp x,
                                     IlcIntExp y)
    : IlcConstraintI(solver), _x(x), _y(y) {}

void IlcDiffConstraint::propagate () {
    if (_x.isBound()) _y.removeValue(_x.getValue());
    if (_y.isBound()) _x.removeValue(_y.getValue());
}

void IlcDiffConstraint::post(){
    _x.whenValue(this);
    _y.whenValue(this);
}

IlcBool IlcDiffConstraint::isViolated() const {
    return (_x.isBound() && _y.isBound() &&
            _x.getValue()==_y.getValue());
}

IlcConstraint IlcDiff(IlcIntExp x, IlcIntExp y){
  IloSolver solver=x.getSolver();
    return new (solver.getHeap()) IlcDiffConstraint(solver, x, y);
}

void IlcDiffConstraint::metaPostDemon(IlcDemonI* ct){
    _x.whenValue(ct);
    _y.whenValue(ct);
}

IlcConstraintI* IlcDiffConstraint::makeOpposite() const {
    return (_x == _y).getImpl();
}
```

All the predefined constraints of Solver are defined like this, as metaconstraints.

As you saw in "Writing your own constraint" on page 337, defining an ordinary new constraint yourself is not as difficult as defining a metaconstraint. Only because you want to exploit `IlcDiff` as a metaconstraint were you obliged to redefine the virtual functions `isViolated`, `metaPostDemon` and `makeOpposite`.

Programming tips

This section describes some useful tips, collected from users of Solver.

Exhaustive propagation

Constraint programming is more efficient when it effectively reduces the variable domains as early as possible.

In order to reduce the domains of constrained variables as quickly as possible, you have to carry out all propagation. You do so by propagating constraints in every direction. While predefined constraints respect this rule, you have to exercise care when writing a new constraint to insure that your user-defined constraints respect this rule as well.

Another way of looking at this issue is to see that constraints are not *directed*, but must be seen as *equations* in the widest sense of the term. The final results of the search for a solution must be the same, regardless of the order in which constraints are posted, even user-defined constraints.

Let's consider the constraint $x + y = z$. The domain of z must be modified when the domain of x or of y is modified. At the same time, you must not forget that the domain of x must be modified when the domain of y or of z is modified. A similar rule applies for y as well. This is what is meant by propagating in all directions.

More generally, when designing a new constraint class, you must not overlook any direction of propagation.

Using demons to propagate selectively

Let's look again at our `IlcDiff` example. Each time *one* of its variables is bound, `isBound` is checked against the *two* variables. You can avoid these redundant tests by using intermediate *demons*, that is, by posting different demons on the different variables. You might think of demons as goals without subgoals, that is, goals which return the null pointer.

You add two member functions to the class `IlcDiffConstraint`, each of them responsible for the propagation of the constraint on one variable when the other is bound. Here is the modification.

```
class IlcDiffConstraint :public IlcConstraintI {
    IlcIntExp _x, _y;
  public:
    IlcDiffConstraint(IloSolver solver, IlcIntExp x, IlcIntExp y);
    ~IlcDiffConstraint(){}; // empty destructor

    virtual void propagate ();
    virtual void post();
    virtual IlcBool isViolated() const;
    virtual IlcConstraintI* makeOpposite() const;
    virtual void metaPostDemon(IlcDemonI* expr);
    void xDemon(){
        _y.removeValue(_x.getValue());
    }
    void yDemon(){
        _x.removeValue(_y.getValue());
    }
};
```

Now you need to define intermediate demons on variables that call the new member functions. You do that in this way.

```
ILCCTDEMONO(IlcDiffCst_xDemon,IlcDiffConstraint,xDemon);
ILCCTDEMON0(IlcDiffCst_yDemon,IlcDiffConstraint,yDemon);
```

Finally, you change the `post` member function to take this modification into account, like this:

```
void IlcDiffConstraint::post(){
    _x.whenValue(IlcDiffCst_xDemon(getSolver(),this));
    _y.whenValue(IlcDiffCst_yDemon(getSolver(),this));
}
```

This new version of the `IlcDiffConstraint` is faster that the previous one since the `isBound` tests in `propagate` are avoided. The price to pay for this efficiency is the memory used by the two intermediate goals.

Writing a constraint: Two examples

In this section, you walk through two examples. One defines a new class of constraints, using demons to propagate modifications. The other example defines a class of constraint exploiting iterators. Both examples begin by designing a model for the semantics of the

constraint involved. That step of defining a model for the semantics is crucial when you find it necessary to define a class of constraints yourself.

A new class of constraint using demons

Let's consider the constraint $x <= y + c$, where x and y are constrained integer expressions, and c is a constant integer. Here's a rule to serve as a model of the semantics of that constraint.

- The constraint is necessarily violated if the minimum of x is greater than the maximum of $y + c$.

The constraint must be propagated when one of the boundaries of x or y is modified.

- If the minimum of x is *val*, y must be greater than or equal to $val - c$.
- If the maximum of y is *val*, x must be less than or equal $val + c$.

The negation of the constraint is $y <= x - c - 1$.

With that semantic model in mind, you can define the corresponding class of constraints like this:

```
class IlcLeConstraint: public IlcConstraintI {
  // x <= y + c
  IlcIntExp _x;
  IlcIntExp _y;
  IlcInt _c;
public:
  IlcLeConstraint(IloSolver solver, IlcIntExp x, IlcIntExp y, IlcInt c=0)
    : IlcConstraintI(solver), _x(x), _y(y), _c(c){}
  void post();
  void propagate();
  IlcBool isViolated() const;
  void metaPostDemon(IlcDemonI*);
  IlcConstraintI* makeOpposite() const;
};
void IlcLeConstraint::post(){
  _x.whenRange(this);
  _y.whenRange(this);
}

void IlcLeConstraint::metaPostDemon(IlcDemonI* demon){
  _x.whenRange(demon);
  _y.whenRange(demon);
}

IlcBool IlcLeConstraint::isViolated() const {
  return _x.getMin() > _y.getMax() + _c;
}

void IlcLeConstraint::propagate(){
  _y.setMin(_x.getMin() - _c);
  _x.setMax(_y.getMax() + _c);
}
```

```
IlcConstraintI* IlcLeConstraint::makeOpposite() const {
  IloSolver solver=getSolver();
  return new (solver.getHeap()) IlcLeConstraint(solver,_y,_x,-_c -1);
}

IlcConstraint IlcLeConstraint(IlcIntExp x, IlcIntExp y, IlcInt c){
  IloSolver solver=x.getSolver();
  return IlcConstraint(new (solver.getHeap()) IlcLeConstraint(solver,x,y,c));
}
```

The constraint is defined with `IlcIntExp`, and not with `IlcIntVar`, in order to avoid the creation of new constrained integer variables.

You can use auxiliary demons for this constraint, instead of calling `propagate` for all propagations. Indeed, when the minimum of *x* is modified, you only need to apply the first of the two rules:

- If the minimum of *x* is *val*, *y* must be greater than or equal to *val* - *c*.
- If the maximum of *y* is *val*, *x* must be less than or equal *val* + *c*.

The new definition is thus:

```
class IlcLeConstraint: public IlcConstraintI {
  // x <= y + c
  IlcIntExp _x;
  IlcIntExp _y;
  IlcInt _c;
public:
  IlcLeConstraint(IloSolver solver, IlcIntExp x, IlcIntExp y, IlcInt c=0)
    : IlcConstraintI(solver), _x(x), _y(y), _c(c){}
  void post();
  void propagate();
  IlcBool isViolated() const;
  void metaPostDemon(IlcDemonI*);
  IlcConstraintI* makeOpposite() const;
  void xDemon(){
    _y.setMin(_x.getMin() - _c);
  }
  void yDemon(){
    _x.setMax(_y.getMax() + _c);
  }
};

ILCCTDEMON0(IlcLeConstraintXDemon,IlcLeConstraint,xDemon);

ILCCTDEMON0(IlcLeConstraintYDemon,IlcLeConstraint,yDemon);

void IlcLeConstraint::post(){
  _x.whenRange(IlcLeConstraintXDemon(getSolver(),this));
  _y.whenRange(IlcLeConstraintYDemon(getSolver(),this));
}

void IlcLeConstraint::metaPostDemon(IlcDemonI* demon){
```

```
  _x.whenRange(demon);
  _y.whenRange(demon);
}

IlcBool IlcLeConstraint::isViolated() const {
  return _x.getMin() > _y.getMax() + _c;
}

void IlcLeConstraint::propagate(){
  xDemon();
  yDemon();
}

IlcConstraintI* IlcLeConstraint::makeOpposite() const {
  IloSolver solver=getSolver();
  return new (solver.getHeap()) IlcLeConstraint(solver,_y,_x,-_c -1);
}

IlcConstraint IlcLeConstraint(IlcIntExp x, IlcIntExp y, IlcInt c){
  IloSolver solver=x.getSolver();
  return IlcConstraint(new (solver.getHeap()) IlcLeConstraint(solver,x,y,c));
}
```

The differences from the previous definition appear in the member functions `xDemon`, `yDemon`, and `propagate`, and in the two demons `IlcLeConstraintXDemon` and `IlcLeConstraintYDemon`.

A new class of constraint using iterators

Let's consider the constraint $i \in E$ if and only if $i + 1 \notin E$ on a set E of integers i. In other words, this set of integers must not contain two consecutive values.

The following rules serve as a model of the semantics of this constraint.

◆ The constraint is necessarily violated if i and i+1 belong to the set.

The constraint must be propagated when new required elements are added to the set.

◆ If i is added to the set, i-1 and i+1 must be removed from the set.

To keep this example to the point, you won't define the negation of the constraint. That is, you will not define `isViolated`, `makeOpposite` or `metaPost`, as you would have to do if you were creating a metaconstraint. With that in mind, here's the definition of the corresponding class of constraint.

```
class IlcNoConsecConstraint : public IlcConstraintI {
    // set E doesn't contain two consecutive values.
    IlcIntSetVar _E;
public:
    IlcNoConsecConstraint(IloSolver solver, IlcIntSetVar E)
        :IlcConstraintI(solver), _E(E){}
    virtual void post         ();
    virtual void propagate    ();
```

```
    void propagateDomain();
};

IlcBool IlcNoConsecConstraint::isViolated() {
    for(IlcIntSetIterator iter(_E.getRequiredSet()); iter.ok(); ++it) {
        IlcInt val = *iter;
              if (_E.isRequired(val +1))
                 return IlcTrue;
    }
    return IlcFalse;
}

ILCCTDEMON(NoConsecConstraintDomainDemon,
           IlcNoConsecConstraintI,
           propagateDomain);
}

void IlcNoConsecConstraint::post(){
    _E.whenDomain(NoConsecConstraintDomainDemon(getSolver(), this));
}

void IlcNoConsecConstraint::propagate() {
    for(IlcIntSetIterator iter(_E.getRequiredSet()); iter.ok(); ++iter){
        IlcInt val = *iter;
        _E.removePossible(val-1);
        _E.removePossible(val+1);
    }
}

void IlcNoConsecConstraintI::propagateDomain() {
    for(IlcIntDeltaRequiredIterator iter(_E); iter.ok(); ++iter){
        IlcInt val = *iter;
        _E.removePossible(val-1);
        _E.removePossible(val+1);
    }
}

IlcConstraint noConsecutiveValues(IlcIntSetVar E){
    IloSolver solver = E.getSolver();
    return IlcConstraint(new (solver.getHeap()) IlcNoConsecConstraint(solver,
E));
}
```

Advanced issues: Propagating constraints after modifying variables

Suppose that you have a constraint *C* involving the variables *x*, *y*, *z*, and *t*. Suppose that *x* is modified, then *C* is propagated. Now if the modification of *x* leads to a modification of *y*, *z*, and *t*, these variables are propagated and each of those variables will propagate *C* again, even if these variables have not been modified by constraints other than *C*.

Imagine that you want to avoid these repeated propagations because you happen to know that the `propagate` member function of *C* is costly. Also imagine that you want to

propagate *C* once for all the modifications of the variables involved in *C* but *not* for the modifications of each variable individually.

There is a mechanism in Solver that meets these aims. It is possible to have constraints that are *not* propagated within demons.

Such a constraint is *not* propagated after each modification of each variable; rather, it is propagated after all its variables have been propagated.

In fact, in Solver there are two mechanisms for propagating the modifications of variables.

◆ First Level: When a variable is modified, all the demons linked to this variable are called. This mechanism is repeated while there is a variable for which all the demons have not yet been called.

◆ Second Level: The `propagate` function of some constraint known as a *global constraint* is called.

If, after the termination of a `propagate` member function, a variable has been modified, you immediately return to the first level. This means that the process at the first level has greatest priority.

In the second level, a *global constraint* (say, `ct`) is pushed by a call to the `push` member function, like this `ct->push();`.

This member function says to Solver that `ct` is a global constraint and must be propagated by the second level mechanism. This member function must be called within a demon.

Consider the following example simplified for illustration: you would like to define the constraint $x + y = s$ without considering modifications of s.

The code of that constraint could be something like this:

```
class MySum : public IlcConstraintI {
private:
IlcIntVar _x;
IlcIntVar _y;
IlcIntVar _s;
public:
MySum(IloSolver solver, IlcIntVar x, IlcIntVar y, IlcIntVar s);
void post();
void propagate();
};

MySum::MySum(IloSolver solver, IlcIntVar x, IlcIntVar y, IlcIntVar s):
IlcConstraintI(solver), _x(x), _y(y), _s(s){}

void MySum::propagate(){
        IlcInt m;
        // we compute the min of s
         m=_x.getMin() + _y.getMin();
         _s.setMin(m);
         // we compute the max of s
         m=_x.getMax() + _y.getMax();
         _s.setMax(m);
}
```

Now you add a demon for each variable `x` and `y`, and that demon is triggered when the boundary of `x` or `y` is modified (that is, when a range event occurs). Let's say this demon is called `MySumDemon`, and it takes the global constraint as a parameter. Here is the code for its `post` member function:

```
void MySum::post(){
        _x.whenRange(MySumDemon(getSolver(),this));
        _y.whenRange(MySumDemon(getSolver(),this));
}
```

Now here is the code of the demon. Instead of calling a particular function of `ct` that you define, you call the `push` member function, like this:

```
ILCCTPUSHDEMON(MySumDemon, MySum){
ct->push();
}
```

How does this tactic differ from the conventional propagation mechanism? In this case, the demon is called, but the `propagate` member function of `MySum` is not called immediately. Rather, `ct` is pushed. When there are no more demons to trigger, the constraints that have been pushed are called. Furthermore, in this way, a constraint is propagated only once even if you push it several times.

If you want to refine the propagation mechanism, for example, if you want to treat the variables that have been modified in a special way, you have to manage that treatment by

using internal data structures within the constraint. For example, suppose that if x is modified, you want to call your own function `propagateX()` and if y has been modified, you want to call your own `propagateY()`. To do so, you add reversible Boolean data members to `MySum`, like this:

```
IlcRevBool _xIsModified
IlcRevBool _yIsModified
```

Now you define special demons for `x` and for `y`, like this:

```
MySum::demonForX(){
  memorizeThatYisModified();
  push();
}

MySum::demonForY(){
  memorizeThatXIsModified();
  push();
}

ILCCTDEMON0(MySymDemonForX, MySum, demonForX);
ILCCTDEMON0(MySumDemonForY, MySum, demonForY);

void MySum::memorizeThatXisModified(){
_xIsModified=IlcTrue;
}
```

Now the `propagate` member function looks like this:

```
void MySum::propagate(){
        if (_xIsModified){
                propagateX();
              _xIsModified=IlcFalse;
        }
        if (_yIsModified){
                propagateY();
                _yIsModified=IlcFalse;
        }
}
```

Thus you have created a global constraint that is propagated only once and propagated only after the propagation of variables in that constraint.

Writing a constraint: Frequency allocation

The problem is given here in the form of discrete data; that is, each frequency is represented by a number that can be called its channel number. For practical purposes, the network is divided into cells. (This problem is an actual cellular phone problem.) In each cell, there is a transmitter which uses different channels. The shape of the cells have been determined, as

well as the precise location where the transmitters will be installed. For each of these cells, traffic requires a number of frequencies.

The problem of frequency assignment is to avoid interference. As a consequence, the distance between the frequencies within a cell must be greater than 16. To avoid inter-cell interference, the distance must vary because of the geography. In the example, you are assuming that the same frequencies can be used in the first cell and in the fourth cell, but you cannot use the same frequencies in the first cell and the third cell. Between two cells, the distance between frequencies appears in a matrix.

```
const int nbCell              = 25;
const int nbAvailFreq         = 256;
const int nbChannel[nbCell] =
  { 8,6,6,1,4,4,8,8,8,8,4,9,8,4,4,10,8,9,8,4,5,4,8,1,1 };
const int dist[nbCell][nbCell] = {
  { 16,1,1,0,0,0,0,0,1,1,1,1,1,2,2,1,1,0,0,0,2,2,1,1,1 },
  { 1,16,2,0,0,0,0,0,2,2,1,1,1,2,2,1,1,0,0,0,0,0,0,0,0 },
  { 1,2,16,0,0,0,0,0,2,2,1,1,1,2,2,1,1,0,0,0,0,0,0,0,0 },
  { 0,0,0,16,2,2,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,1,1 },
  { 0,0,0,2,16,2,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,1,1 },
  { 0,0,0,2,2,16,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,1,1 },
  { 0,0,0,0,0,0,16,2,0,0,1,1,1,0,0,1,1,1,1,2,0,0,0,1,1 },
  { 0,0,0,0,0,0,2,16,0,0,1,1,1,0,0,1,1,1,1,2,0,0,0,1,1 },
  { 1,2,2,0,0,0,0,0,16,2,2,2,2,2,2,1,1,1,1,1,1,1,0,1,1 },
  { 1,2,2,0,0,0,0,0,2,16,2,2,2,2,2,1,1,1,1,1,1,1,0,1,1 },
  { 1,1,1,0,0,0,1,1,2,2,16,2,2,2,2,2,2,1,1,2,1,1,0,1,1 },
  { 1,1,1,0,0,0,1,1,2,2,2,16,2,2,2,2,2,1,1,2,1,1,0,1,1 },
  { 1,1,1,0,0,0,1,1,2,2,2,2,16,2,2,2,2,1,1,2,1,1,0,1,1 },
  { 2,2,2,0,0,0,0,0,2,2,2,2,2,16,2,1,1,1,1,1,1,1,1,1,1 },
  { 2,2,2,0,0,0,0,0,2,2,2,2,2,2,16,1,1,1,1,1,1,1,1,1,1 },
  { 1,1,1,0,0,0,1,1,1,1,2,2,2,1,1,16,2,2,2,1,2,2,1,2,2 },
  { 1,1,1,0,0,0,1,1,1,1,2,2,2,1,1,2,16,2,2,1,2,2,1,2,2 },
  { 0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,2,2,16,2,2,1,1,0,2,2 },
  { 0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,2,2,2,16,2,1,1,0,2,2 },
  { 0,0,0,1,1,1,2,2,1,1,2,2,2,1,1,1,1,2,2,16,1,1,0,1,1 },
  { 2,0,0,0,0,0,0,0,1,1,1,1,1,1,1,2,2,1,1,1,16,2,1,2,2 },
  { 2,0,0,0,0,0,0,0,1,1,1,1,1,1,1,2,2,1,1,1,2,16,1,2,2 },
  { 1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,1,1,16,1,1 },
  { 1,0,0,1,1,1,1,1,1,1,1,1,1,1,1,2,2,2,2,1,2,2,1,16,2 },
  { 1,0,0,1,1,1,1,1,1,1,1,1,1,1,1,2,2,2,2,1,2,2,1,2,16 }
};
```

Problem representation: Designing the model

You've decided to represent each transmitter in every cell by a finite domain variable (of type `IlcIntVar`) which takes its value from the set of available channel numbers.

Distance constraints are expressed in a deterministic way by using the predefined symbolic constraint `IloAbs`. This deterministic expression is important with respect to the complexity of the algorithm, since the constraint `IloAbs(X-Y) >= D` replaces the disjunction of the two constraints `X-Y >= D` and `Y-X >= D`. This principle—of using a symbolic constraint

instead of the disjunction of two or more constraints—is one that can help you reduce the complexity of other problems as well.

This problem, for which the data set (namely, the distance matrix) is rather small, translates into a quite large number of constraints—more than 8000.

Search strategy: Choosing values

The most important part of this program is related to the strategy for choosing the values to give to the different variables. The strategy can be described informally as trying the most used frequency first. Consequently, you have to keep track of which frequencies are used. You manage this record-keeping in an array of `IlcRevInt*`. (You used an array of `IlcRevInt*`, instead of an array of `IlcRevInt` because some C++ compilers are unable to initialize an array of objects.)

A demon is activated each time a variable is bound. In that case, you increment the number of times that the corresponding frequency is used.

```
IlcRevInt** freqUsage;


class IlcFreqConstraintI : public IlcConstraintI {
protected:
  IlcIntVarArray _x;
 public:
  IlcFreqConstraintI(IloSolver s, IlcIntVarArray x)
    : IlcConstraintI(s), _x(x) {}
  ~IlcFreqConstraintI() {}
  virtual void post();
  virtual void propagate() {}
  void varDemon(IlcIntVar var);
};

ILCCTDEMON1(FreqDemon, IlcFreqConstraintI, varDemon, IlcIntVar, var);

void IlcFreqConstraintI::post () {
  IlcInt i;
  for (i = 0; i < _x.getSize(); i++)
    _x[i].whenValue(FreqDemon(getSolver(), this, _x[i]));
}

void IlcFreqConstraintI::varDemon (IlcIntVar var) {
  IlcRevInt& freq = *freqUsage[var.getValue()];
  freq.setValue(getSolver(), freq.getValue() + 1);
}

IlcConstraint IlcFreqConstraint(IloSolver s, IlcIntVarArray x) {
  return new (s.getHeap()) IlcFreqConstraintI(s, x);
}
```

Using the constraint wrapper

You use the macro `ILOCPCONSTRAINTWRAPPER` to wrap an existing instance of `IlcConstraint` when you want to use it within Concert Technology model objects. You must use the member function `IloCPConstraintI::use` to force extraction of an extractable or an array of extractables:

```
ILOCPCONSTRAINTWRAPPER1(IloFreqConstraint, solver, IloIntVarArray, _vars) {
  use(solver, _vars);
  return IlcFreqConstraint(solver, solver.getIntVarArray(_vars));
}
```

For more information on wrapping constraints for use with Concert Technology, see the documentation for the macro `ILOCPCONSTRAINTWRAPPER` and the class `IloCPConstraintI` in the *IBM ILOG Solver Reference Manual*.

Defining the Instantiate goal

Now you can define the goal `Instantiate` this way:

```
IlcInt bestFreq(IlcIntVar var) {
  IlcInt max = -1;
  IlcInt maxIndex = 0;
  IlcInt i;
  for(IlcIntExpIterator iter(var);iter.ok();++iter){
    i=*iter;
    if (*freqUsage[i] > max){
      max = *freqUsage[i];
      maxIndex = i;
    }
  }
  return maxIndex;
}

ILCGOAL1(Instantiate, IlcIntVar, var) {
  if (var.isBound()) return 0;
  IlcInt val = bestFreq(var);
  return IlcOr(var == val,
               IlcAnd(var != val,
                      this));
}
```

For more information on writing your own goal and using a goal wrapper, see Chapter 18, *Writing a Goal: Car Sequencing*.

Search strategy: Choice function

Next, it's necessary to determine the order in which the different variables will be tried. After careful thinking and many tries, you learned that variables with the smallest domain must be tried first, and that in case of ties, it is best first to try variables which correspond to cells where there are few channels.

Each variable is therefore associated with the number of channels needed at the corresponding cell. The following fragment of code makes that association between a cell and the number of channels it needs. The number of channels is stored in the object associated with the variable.

```
void SetCluster(IlcIntVar var, IlcInt clusterSize) {
  var.setObject((IlcAny) clusterSize);
}

IlcInt GetCluster(IlcIntVar var) {
  return (IlcInt) (var.getObject());
}
```

Then the choice criterion is implemented like this:

```
static IlcChooseIndex2(IlcChooseClusterFreq,
                       var.getSize(),
                       GetCluster(var),
                       IlcIntVar)
```

Search strategy: Algorithm

It is now possible to generate the different solutions. As explained before, the instantiation goal first tries the most used frequency, while the generation goal chooses the variable with the smallest domain and the least cardinality for a cell. The following code makes those choices.

```
ILCGOAL1(MyGenerate, IlcIntVarArray, vars) {
  IloSolver solver = getSolver();
  IlcInt chosen = IlcChooseClusterFreq(vars);
  if(chosen == -1)
    return 0;
  return IlcAnd(Instantiate(solver, vars[chosen]),this);
}

ILOCPGOALWRAPPER1(MyIloGenerate, solver, IloIntVarArray, vars) {
  return MyGenerate(solver, solver.getIntVarArray(vars));
}
```

Main program

The `main` program is straightforward. It uses a single array to represent all the variables, so "housekeeping" is simple. To access these variables, there are only two features: the function `acc` and the `partialsum` array. Apart from that, the program simply sets the constraints deterministically using `IloAbs`, as mentioned before.

The function `acc` accesses a two-dimensional array representing the channels for each cell. The first index denotes the cell number. Since the number of channels per cell varies, the number of elements in each row in the two-dimensional array varies as well. Consequently, the jth element in row i has a partial sum as its index, $partialSum[i]+j$, where $partialSum[i]$ is defined recursively, like this:

- At zero, $partialSum[0] = 0$.
- For $i >= 1$, $partialSum[i] = rowSize[i-1] + partialSum[i-1]$ where $rowSize[i]$ is the number of channels.

In the loops in this program, the indexes *i*, *j*, *ii*, and *jj* indicate that for each pair *(i, j)* of cells, and for each channel *ii* of *i* and *jj* of *j*, the distance constraint holds between such two channels (taken from the distance matrix).

You can see the entire program online in the file `freq.cpp`. The `main` program follows:

```
IlcInt partialsum[nbCell];
inline IlcInt acc(IlcInt i, IlcInt j) {
  return partialsum[i]+j;
}

int main(){
  IloEnv env;
  try {
    IloModel model(env);
    IloSolver solver(env);

    IlcInt i, ii, j, jj;
    IlcInt usedFreq = 0;
    IlcInt nbXmiter = 0;

    for (i = 0; i < nbCell; i++) {
      partialsum[i] = nbXmiter;
      nbXmiter += nbChannel[i];
    }

    IloIntVarArray X(env, nbXmiter, 0, nbAvailFreq - 1);

    for (i = 0; i < nbCell; i++)
      for (j = i; j < nbCell; j++)
        for (ii = 0; ii < nbChannel[i]; ii++)
          for (jj = 0; jj < nbChannel[j]; jj++)
            if (dist[i][j] != 0 && (i != j || ii != jj))
              model.add(IloAbs(X[acc(i,ii)] - X[acc(j,jj)]) >= dist[i][j]);

    model.add(IloFreqConstraint(env, X));
```

```
    solver.extract(model);

    for (i = 0; i < nbCell; i++)
      for (ii = 0; ii < nbChannel[i]; ii++)
        SetCluster(solver.getIntVar(X[acc(i,ii)]), nbChannel[i]);

    freqUsage = new (env) IlcRevInt* [nbAvailFreq];
    for (i = 0; i < nbAvailFreq; i++)
      freqUsage[i] = new (env) IlcRevInt(solver);

    solver.solve(MyIloGenerate(env, X));

    for (i = 0; i < nbCell; i++) {
      for (j = 0; j < nbChannel[i]; j++)
        solver.out() << solver.getValue(X[acc(i,j)]) << "  " ;
      solver.out() << endl;
    }
    solver.out() << "Total # of sites       " << nbXmiter << endl;
    for (i = 0; i < nbAvailFreq; i++)
      if (freqUsage[i]->getValue() != 0)
        usedFreq++;
    solver.out() << "Total # of frequencies " << usedFreq << endl;
  }
  catch (IloException& ex) {
    cout << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Complete frequency allocation program

The complete program follows. You can also view the entire program online in the file `YourSolverHome/examples/src/freq.cpp`.

```
#include <ilsolver/ilosolverint.h>

ILOSTLBEGIN

const int nbCell              = 25;
const int nbAvailFreq         = 256;
const int nbChannel[nbCell] =
  { 8,6,6,1,4,4,8,8,8,8,4,9,8,4,4,10,8,9,8,4,5,4,8,1,1 };
const int dist[nbCell][nbCell] = {
  { 16,1,1,0,0,0,0,0,1,1,1,1,1,2,2,1,1,0,0,0,2,2,1,1,1 },
  { 1,16,2,0,0,0,0,0,2,2,1,1,1,2,2,1,1,0,0,0,0,0,0,0,0 },
  { 1,2,16,0,0,0,0,0,2,2,1,1,1,2,2,1,1,0,0,0,0,0,0,0,0 },
  { 0,0,0,16,2,2,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,1,1 },
  { 0,0,0,2,16,2,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,1,1 },
  { 0,0,0,2,2,16,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,1,1 },
  { 0,0,0,0,0,0,16,2,0,0,1,1,1,0,0,1,1,1,1,2,0,0,0,1,1 },
  { 0,0,0,0,0,0,2,16,0,0,1,1,1,0,0,1,1,1,1,2,0,0,0,1,1 },
  { 1,2,2,0,0,0,0,0,16,2,2,2,2,2,2,1,1,1,1,1,1,1,0,1,1 },
  { 1,2,2,0,0,0,0,0,2,16,2,2,2,2,2,1,1,1,1,1,1,1,0,1,1 },
  { 1,1,1,0,0,0,1,1,2,2,16,2,2,2,2,2,2,1,1,2,1,1,0,1,1 },
```

```
  { 1,1,1,0,0,0,1,1,2,2,2,16,2,2,2,2,2,1,1,2,1,1,0,1,1 },
  { 1,1,1,0,0,0,1,1,2,2,2,2,16,2,2,2,2,1,1,2,1,1,0,1,1 },
  { 2,2,2,0,0,0,0,0,2,2,2,2,2,16,2,1,1,1,1,1,1,1,1,1,1 },
  { 2,2,2,0,0,0,0,0,2,2,2,2,2,2,16,1,1,1,1,1,1,1,1,1,1 },
  { 1,1,1,0,0,0,1,1,1,1,2,2,2,1,1,16,2,2,2,1,2,2,1,2,2 },
  { 1,1,1,0,0,0,1,1,1,1,2,2,2,1,1,2,16,2,2,1,2,2,1,2,2 },
  { 0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,2,2,16,2,2,1,1,0,2,2 },
  { 0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,2,2,2,16,2,1,1,0,2,2 },
  { 0,0,0,1,1,1,2,2,1,1,2,2,2,1,1,1,1,2,2,16,1,1,0,1,1 },
  { 2,0,0,0,0,0,0,0,1,1,1,1,1,1,1,2,2,1,1,1,16,2,1,2,2 },
  { 2,0,0,0,0,0,0,0,1,1,1,1,1,1,1,2,2,1,1,1,2,16,1,2,2 },
  { 1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,1,1,16,1,1 },
  { 1,0,0,1,1,1,1,1,1,1,1,1,1,1,1,2,2,2,2,1,2,2,1,16,2 },
  { 1,0,0,1,1,1,1,1,1,1,1,1,1,1,1,2,2,2,2,1,2,2,1,2,16 }
};

IlcRevInt** freqUsage;


class IlcFreqConstraintI : public IlcConstraintI {
protected:
  IlcIntVarArray _x;
 public:
  IlcFreqConstraintI(IloSolver s, IlcIntVarArray x)
    : IlcConstraintI(s), _x(x) {}
  ~IlcFreqConstraintI() {}
  virtual void post();
  virtual void propagate() {}
  void varDemon(IlcIntVar var);
};

ILCCTDEMON1(FreqDemon, IlcFreqConstraintI, varDemon, IlcIntVar, var);

void IlcFreqConstraintI::post () {
  IlcInt i;
  for (i = 0; i < _x.getSize(); i++)
    _x[i].whenValue(FreqDemon(getSolver(), this, _x[i]));
}

void IlcFreqConstraintI::varDemon (IlcIntVar var) {
  IlcRevInt& freq = *freqUsage[var.getValue()];
  freq.setValue(getSolver(), freq.getValue() + 1);
}

IlcConstraint IlcFreqConstraint(IloSolver s, IlcIntVarArray x) {
  return new (s.getHeap()) IlcFreqConstraintI(s, x);
}



ILOCPCONSTRAINTWRAPPER1(IloFreqConstraint, solver, IloIntVarArray, _vars) {
  use(solver, _vars);
  return IlcFreqConstraint(solver, solver.getIntVarArray(_vars));
}


IlcInt bestFreq(IlcIntVar var) {
  IlcInt max = -1;
```

```
  IlcInt maxIndex = 0;
  IlcInt i;
  for(IlcIntExpIterator iter(var);iter.ok();++iter){
    i=*iter;
    if (*freqUsage[i] > max){
      max = *freqUsage[i];
      maxIndex = i;
    }
  }
  return maxIndex;
}

ILCGOAL1(Instantiate, IlcIntVar, var) {
  if (var.isBound()) return 0;
  IlcInt val = bestFreq(var);
  return IlcOr(var == val,
               IlcAnd(var != val,
                      this));
}

void SetCluster(IlcIntVar var, IlcInt clusterSize) {
  var.setObject((IlcAny) clusterSize);
}

IlcInt GetCluster(IlcIntVar var) {
  return (IlcInt) (var.getObject());
}

static IlcChooseIndex2(IlcChooseClusterFreq,
                       var.getSize(),
                       GetCluster(var),
                       IlcIntVar)

ILCGOAL1(MyGenerate, IlcIntVarArray, vars) {
  IloSolver solver = getSolver();
  IlcInt chosen = IlcChooseClusterFreq(vars);
  if(chosen == -1)
    return 0;
  return IlcAnd(Instantiate(solver, vars[chosen]),this);
}

ILOCPGOALWRAPPER1(MyIloGenerate, solver, IloIntVarArray, vars) {
  return MyGenerate(solver, solver.getIntVarArray(vars));
}

IlcInt partialsum[nbCell];
inline IlcInt acc(IlcInt i, IlcInt j) {
  return partialsum[i]+j;
}

int main(){
  IloEnv env;
  try {
    IloModel model(env);
    IloSolver solver(env);

    IlcInt i, ii, j, jj;
    IlcInt usedFreq = 0;
```

```
    IlcInt nbXmiter = 0;

    for (i = 0; i < nbCell; i++) {
      partialsum[i] = nbXmiter;
      nbXmiter += nbChannel[i];
    }

    IloIntVarArray X(env, nbXmiter, 0, nbAvailFreq - 1);

    for (i = 0; i < nbCell; i++)
      for (j = i; j < nbCell; j++)
        for (ii = 0; ii < nbChannel[i]; ii++)
          for (jj = 0; jj < nbChannel[j]; jj++)
            if (dist[i][j] != 0 && (i != j || ii != jj))
              model.add(IloAbs(X[acc(i,ii)] - X[acc(j,jj)]) >= dist[i][j]);

    model.add(IloFreqConstraint(env, X));

    solver.extract(model);

    for (i = 0; i < nbCell; i++)
      for (ii = 0; ii < nbChannel[i]; ii++)
        SetCluster(solver.getIntVar(X[acc(i,ii)]), nbChannel[i]);

    freqUsage = new (env) IlcRevInt* [nbAvailFreq];
    for (i = 0; i < nbAvailFreq; i++)
      freqUsage[i] = new (env) IlcRevInt(solver);

    solver.solve(MyIloGenerate(env, X));

    for (i = 0; i < nbCell; i++) {
      for (j = 0; j < nbChannel[i]; j++)
        solver.out() << solver.getValue(X[acc(i,j)]) << "  " ;
      solver.out() << endl;
    }
    solver.out() << "Total # of sites       " << nbXmiter << endl;
    for (i = 0; i < nbAvailFreq; i++)
      if (freqUsage[i]->getValue() != 0)
        usedFreq++;
    solver.out() << "Total # of frequencies " << usedFreq << endl;
  }
  catch (IloException& ex) {
    cout << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Output

The strategy used here allows the program to find good solutions. Indeed, on this data, it leads directly to the optimal solution. Here is the output:

```
25  41  57  73  89  105  121  137
1  21  37  53  71  87
6  23  39  55  75  103
0
2  18  34  50
4  20  36  52
75  10  27  43  59  96  112  128
13  30  46  62  80  99  115  131
13  35  51  67  83  99  115  131
19  64  80  96  112  128  144  160
0  16  32  48
8  26  42  58  74  90  106  122  138
2  24  40  56  72  88  104  120
4  28  44  60
11  30  46  62
20  36  52  68  84  100  116  132  148  164
18  34  50  66  82  98  114  130
9  25  41  57  73  89  105  121  137
7  23  39  55  71  87  103  119
5  21  37  53
10  27  43  59  75
6  22  38  54
0  21  37  53  71  87  103  128
1
3
Total # of sites       148
Total # of frequencies 95
```

C H A P T E R

20

Writing a Search Limit: Car Sequencing

In this lesson, you will learn how to:

- ◆ understand search limits
- ◆ write your own limit
- ◆ solve a car sequencing problem using a custom search limit

You can redefine the behavior of Solver search procedures to adapt to your particular problem using search limits.

Understanding search limits

The class `IloSearchLimit` represents search limits in an IBM® ILOG® Concert Technology model. The class `IlcSearchLimit` represents search limits internally in a Solver search.

Instances of the class `IlcSearchLimit` are handles to instances of the class `IlcSearchLimitI` and let you indicate to Solver that it should stop the search when it reaches a limit that you set, regardless of its progress in the search. Typical limits that you might set are time limits, for example.

Instances of the class `IloSearchLimit` are handles to instances of the class `IloSearchLimitI`, objects linked to a model. They are allocated on the environment, an

instance of `IloEnv`, and not on the Solver reversible heap. When an instance of the class `IloSearchLimit` is used, the `IloSolver::extract` method is called. This member function returns an instance of the class `IlcSearchLimit` to be used during the search.

Predefined search limits in Concert Technology include `IloTimeLimit`, `IloFailLimit`, and `IloOrLimit`. For many problems, you will achieve satisfactory solutions by using those predefined objects. However, if you want to change search behavior, you can write your own implementation of one or more of the virtual member functions to get the effects you want.

Writing your own search limits

To demonstrate how to write a search limit, you will write a time limit on the search. You will use a floating-point value to build the time limit, and this choice gives you the following API:

```
class IlcTimeLimitI : public IlcSearchLimitI {
private:
  IlcFloat _startTime;
  IlcFloat _limit;
public:
  IlcTimeLimitI(IloSolver solver, IlcBool duplicate, IlcFloat l);
  void init();
  IlcBool check() const;
  IlcSearchLimitI* duplicateLimit(IloSolver);
};
```

Its constructor simply initializes the `_limit` member.

```
IlcTimeLimitI::IlcTimeLimitI (IloSolver solver, IlcBool duplicate, IlcFloat l)
  : IlcSearchLimitI(solver, duplicate), _startTime(0.0), _limit(l) {}
```

You write a member function `init` to set the initial time to be used later when Solver checks the limit.

```
void IlcTimeLimitI::init() {
  _startTime = getSolver()->getTime();
}
```

Now you write a member function `check` to compare the current time to the initial time plus the time limit.

```
IlcBool IlcTimeLimitI::check() const {
  return (getSolver()->getTime() >= _startTime + _limit);
}
```

The member function `duplicateLimit` clones the current object and sets the value of the `duplicate` parameter in the constructor to `IlcTrue`. It allocates the duplicated object on the C++ heap.

```
IlcSearchLimitI* IlcTimeLimitI::duplicateLimit(IloSolver solver) {
  return new IlcTimeLimitI(solver, IlcTrue, _limit);
}
```

The function `IlcTimeLimit` returns a handle of type `IlcSearchLimit` built from an instance of the class `IlcTimeLimitI`. This instance is built with the `duplicate` parameter set to `IlcFalse`, and it must be allocated on the reversible Solver heap.

```
IlcSearchLimit IlcTimeLimit(IloSolver solver, IlcFloat l) {
  IlcSearchLimitI* limit = new (solver.getHeap()) IlcTimeLimitI(solver,
IlcFalse, l);
  return IlcSearchLimit(limit);
}
```

Writing a search limit: Car sequencing

You will now write a custom search limit for a car sequencing problem. Apart from the custom search limit, the problem is that same as that modeled in Chapter 6, *Using the Distribute Constraint: Car Sequencing*.

> **Note:** *This is a simplified version of the car sequencing problem that appears in Chapter 18, Writing a Goal: Car Sequencing.*

Assume that you have 8 cars to paint in three available colors: green, yellow, and blue. Due to technical limitations on the assembly line, no more than three cars can be painted green, exactly three cars must be painted yellow, and no more than two cars can be painted blue. The first car off the assembly line cannot be painted green.

You will add a search limit that stops the search after two solutions have been found that contain a sequence where the sixth car is green.

> **Note:** *This example implements this unusual limit to demonstrate the fact that the limit behavior is completely customizable.*

Writing the custom search limit

You use the class `IlcSearchLimitI` to write a custom search limit:

```
class IlcMyLimitI : public IlcSearchLimitI {
private:
  IlcInt& _counter;
  IlcInt _limit;
public:
  IlcMyLimitI(IloSolver, IlcBool, IlcInt& counter, IlcInt limit);
  void init(const IlcSearchNode);
  IlcBool check(const IlcSearchNode) const;
  IlcSearchLimitI* duplicateLimit(IloSolver);
};
```

The constructor initializes the `_limit` member:

```
IlcMyLimitI::IlcMyLimitI (IloSolver s, IlcBool dup, IlcInt& counter,
                              IlcInt limit) :
  IlcSearchLimitI(s, dup), _counter(counter), _limit(limit) {}
```

The member function `init` stores a reference state for the node evaluator. This will be used when Solver checks the limit.

```
void IlcMyLimitI::init(const IlcSearchNode) {}
```

The member function `check` checks to see if the solution counter has passed the search limit:

```
IlcBool IlcMyLimitI::check(const IlcSearchNode node) const {
  if (_counter >= _limit) {
    node.getSolver().out()
      << "Limit crossed: the counter has attained the limit "
      << _limit << endl;
    return IlcTrue;
  }
  return IlcFalse;
}
```

The member function `duplicateLimit` clones the current object and sets the value of the duplicate parameter in the constructor to `IlcTrue`. It allocates the duplicated object on the C++ heap:

```
IlcSearchLimitI* IlcMyLimitI::duplicateLimit(IloSolver s) {
  return new IlcMyLimitI(s, IlcTrue, _counter, _limit);
}
```

The function IlcMyLimit returns a handle of type IlcSearchLimit built from an instance of the class IlcMyLimitI. This instance is built with the duplicate parameter set to IlcFalse, and it must be allocated on the reversible Solver heap:

```
IlcSearchLimit IlcMyLimit(IloSolver s, IlcInt& c, IlcInt l) {
  IlcSearchLimitI* limit = new (s.getHeap()) IlcMyLimitI(s, IlcFalse , c, l);
  return IlcSearchLimit(limit);
}
```

Custom search limits in a Concert Technology model

Instances of the class IloSearchLimit are objects linked to a model. They are allocated on the environment, an instance of IloEnv, and not on the Solver reversible heap. When an instance of this class is used, the IloSolver::extract method is called. This method returns an instance of the class IlcSearchLimit to be used during the search.

To use a custom search limit in a Concert Technology model, you must use the class IloSearchLimitI.

```
class IloMyLimitI : public IloSearchLimitI {
  IloInt& _counter;
  IloInt _limit;
public:
  IloMyLimitI(IloEnvI*, IloInt&, IloInt);
  virtual IlcSearchLimit extract(const IloSolver) const;
  virtual IloSearchLimitI* makeClone(IloEnvI* env) const;
  virtual void display(ILOSTD(ostream&)) const;
};
```

The constructor initializes the _limit member:

```
IloMyLimitI::IloMyLimitI(IloEnvI* e, IloInt& counter, IloInt limit) :
  IloSearchLimitI(e), _counter(counter), _limit(limit) {}
```

The member function extract returns the internal search limit extracted for solver from the invoking search limit of the model:

```
IlcSearchLimit IloMyLimitI::extract(const IloSolver solver) const {
  return IlcMyLimit(solver, _counter, _limit);
}
```

The member function makeClone is called internally to duplicate the current search limit:

```
IloSearchLimitI* IloMyLimitI::makeClone(IloEnvI* env) const {
  return new (env) IloMyLimitI(env, _counter, _limit);
}
```

The member function `display` prints the invoking search limit on an output stream:

```
void IloMyLimitI::display(ostream& str) const {
  str << "IloMyLimit(" << _counter << ", " << _limit << ") ";
}
```

The function `IloMyLimit` returns a handle of type `IloSearchLimit` built from an instance of the class `IloMyLimitI`.

```
IloSearchLimit IloMyLimit(const IloEnv env, IloInt& counter, IloInt limit) {
  return new (env) IloMyLimitI(env.getImpl(), counter, limit);
}
```

Using the search limit in a goal

The custom search limit is used in `finalGoal`. The solution counter used by the search limit only increments if the sixth car in the sequence is green. When the solution counter reaches the search limit, the search is stopped.

```
    IloSolver solver(model);
    IloGoal goal = IloGenerate(env, cars, IloChooseMinSizeInt);
    IloInt counter=0;
    IloSearchLimit limit = IloMyLimit(env, counter, 2);
    IloGoal finalGoal = IloLimitSearch(env, goal, limit);
    solver.startNewSearch(finalGoal);
    while (solver.next())
      {
      solver.out() << solver.getStatus() << " Solution" << endl;
      if (solver.getValue(cars[5]) == 0) { // Car 6 is green
        for (i = 0; i < nbCars; i++) {
          solver.out() << "Car " << i+1 <<  " color:     "
                       << Names[(IloInt)solver.getValue(cars[i])]
                       << endl;
        }
        counter++;
      }
    }
      solver.endSearch();
```

Complete program

The complete program follows. You can also view the entire program online in the file `YourSolverHome/examples/src/carseq_limit.cpp`.

```
#include <ilsolver/ilosolver.h>

ILOSTLBEGIN

class IlcMyLimitI : public IlcSearchLimitI {
private:
  IlcInt& _counter;
```

```
  IlcInt _limit;
public:
  IlcMyLimitI(IloSolver, IlcBool, IlcInt& counter, IlcInt limit);
  void init(const IlcSearchNode);
  IlcBool check(const IlcSearchNode) const;
  IlcSearchLimitI* duplicateLimit(IloSolver);
};

IlcMyLimitI::IlcMyLimitI (IloSolver s, IlcBool dup, IlcInt& counter,
                              IlcInt limit) :
  IlcSearchLimitI(s, dup), _counter(counter), _limit(limit) {}

void IlcMyLimitI::init(const IlcSearchNode) {}

IlcBool IlcMyLimitI::check(const IlcSearchNode node) const {
  if (_counter >= _limit) {
    node.getSolver().out()
      << "Limit crossed: the counter has attained the limit "
      << _limit << endl;
    return IlcTrue;
  }
  return IlcFalse;
}

IlcSearchLimitI* IlcMyLimitI::duplicateLimit(IloSolver s) {
  return new IlcMyLimitI(s, IlcTrue, _counter, _limit);
}

IlcSearchLimit IlcMyLimit(IloSolver s, IlcInt& c, IlcInt l) {
  IlcSearchLimitI* limit = new (s.getHeap()) IlcMyLimitI(s, IlcFalse , c, l);
  return IlcSearchLimit(limit);
}

class IloMyLimitI : public IloSearchLimitI {
  IloInt& _counter;
  IloInt _limit;
public:
  IloMyLimitI(IloEnvI*, IloInt&, IloInt);
  virtual IlcSearchLimit extract(const IloSolver) const;
  virtual IloSearchLimitI* makeClone(IloEnvI* env) const;
  virtual void display(ILOSTD(ostream&)) const;
};

IloMyLimitI::IloMyLimitI(IloEnvI* e, IloInt& counter, IloInt limit) :
  IloSearchLimitI(e), _counter(counter), _limit(limit) {}

IlcSearchLimit IloMyLimitI::extract(const IloSolver solver) const {
  return IlcMyLimit(solver, _counter, _limit);
}

IloSearchLimitI* IloMyLimitI::makeClone(IloEnvI* env) const {
  return new (env) IloMyLimitI(env, _counter, _limit);
}

void IloMyLimitI::display(ostream& str) const {
  str << "IloMyLimit(" << _counter << ", " << _limit << ") ";
}
```

```
IloSearchLimit IloMyLimit(const IloEnv env, IloInt& counter, IloInt limit) {
  return new (env) IloMyLimitI(env.getImpl(), counter, limit);
}


const char* Names[] = {"green", "yellow", "blue"};

int main() {
  IloEnv env;
  try {
    IloModel model(env);
    const IloInt nbCars    = 8;
    const IloInt nbColors = 3;
    IloInt i;
    IloIntVarArray cars(env, nbCars, 0, nbColors-1);
    IloIntArray colors(env, 3, 0, 1, 2);

    IloIntVarArray cards(env, nbColors);
    cards[0] = IloIntVar(env, 0, 3);
    cards[1] = IloIntVar(env, 3, 3);
    cards[2] = IloIntVar(env, 0, 2);

    model.add(IloDistribute(env, cards, colors, cars));
    model.add(cars[0] != 0);

    IloSolver solver(model);
    IloGoal goal = IloGenerate(env, cars, IloChooseMinSizeInt);
    IloInt counter=0;
    IloSearchLimit limit = IloMyLimit(env, counter, 2);
    IloGoal finalGoal = IloLimitSearch(env, goal, limit);
    solver.startNewSearch(finalGoal);
    while (solver.next())
      {
      solver.out() << solver.getStatus() << " Solution" << endl;
      if (solver.getValue(cars[5]) == 0) { // Car 6 is green
        for (i = 0; i < nbCars; i++) {
          solver.out() << "Car " << i+1 <<  " color:     "
                       << Names[(IloInt)solver.getValue(cars[i])]
                       << endl;
        }
        counter++;
      }
    }
      solver.endSearch();
  }
  catch (IloException& ex) {
    cout << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

CHAPTER

21

Using Impacts during Search

In this lesson, you will learn how to:

- understand impacts
- create customizable goals for integer variables
- restart the search while using impacts

During search, when making choices on a variable `x`, like `x = a`, constraint propagation reduces domains. Impacts produce a certain measure of the domain reduction of such an instantiation that gives rise to a family of efficient search strategies.

Understanding impacts

The impact of an assignment `x = a` is an estimation of the proportion of the search space that this assignment eliminates by constraint propagation. This proportion is computed by considering the size of variable domains before and after the assignment. In a way, the impact is an estimation of how constrained the assignment is.

In general, the impact of a given assignment `x = a` does not vary much from one node of the search tree to another. As a consequence, the value used is the average of the eliminated proportion observed on every other assignment of `x = a`, at a given point in the search. Moreover, impacts are available without significant overhead.

In practical terms, to use impacts it is necessary to add an instance of `IlcConstraintAggregator` and to specify the variable on which you want to produce impacts. For example, if `vars` is such an array of variables, you need to call the following function before model extraction:

```
solver.use(IlcImpactInformation(env, vars));
```

The (averaged) impact of the assignment `x = a` is obtained by calling the function:

```
IlcFloat IloSolver::getImpact(const IlcIntVar x,
                              IlcInt a,
                              IlcBool countFails = IlcTrue) const;
```

The `countFails` parameter specifies if failures of the instantiation `x = a` are counted in the average. A failure has an impact equal to 1 because it eliminates the whole search space below it.

For a variable it is assumed that, a priori, every value of the variable will be tried to find a solution. Consequently, the impact is the sum of the impacts of the values in the current domain of the variable. In other words, it is an estimation of the effort needed to explore the whole search space starting with this variable.

The impact of a variable is obtained by calling the function:

```
IlcFloat IloSolver::getImpact(const IlcIntVar x) const;
```

It is important to note that for impacts to be updated, you must use the goals returned by the following functions, which make choices:

```
IlcGoal IlcSetValue(const IlcIntVar var, const IlcInt val);
IlcGoal IlcRemoveValue(const IlcIntVar var, const IlcInt val);
```

In some situations (for example, when the impacts vary more than expected in the search tree) it is worthwhile to compute the impact of an assignment at a node instead of using the averaged observations. This value is obtained for values and variables with the functions:

```
IlcFloat IloSolver::getLocalImpact(const IlcIntVar x, IlcInt v) const;
IlcFloat IloSolver::getLocalVarImpact(const IlcIntVar x,
                                      IlcInt depth = -1) const;
```

Note that a call to this function does instantiations and propagation to compute the local impact. Therefore, it can create a significant overhead.

Impact-based search strategies

In general, a good search strategy starts by:

- ◆ first instantiating variables that are the most constrained—since you have to give a value to *every* variable it is better to start with the most constrained first

- once a variable is chosen, first try the values that are the least constrained in order to have room for making better choices in the future

Using impacts, this means that you start with the variable `x` having the largest value returned by the call `solver.getImpact(x);` and a value `a` of the domain of `x` having the lowest value returned by the call `solver.getImpact(x, a);`.

The default goal described in the next section provides examples of strategies using impacts.

Initialization of impacts

Impacts are obtained from observations of instantiation during the search tree. However, before starting the search no impacts are available and this can lead to bad choices of variables and values at the beginning of the search.

To initialize impacts a goal is available that performs a dichotomous search on each variable. The goal is:

```
IloGoal IloInitializeImpactGoal(IloEnv env, IloInt depth = -1);
```

It performs a search on each variable on which impacts were required when the function `solver.use(IlcImpactInformation(env, vars))` was called.

The parameter `depth` controls the depth of the dichotomous search. The larger the depth, the more accurate the impact values are. When `depth` equals -1, there is no limit imposed on the depth of dichotomy.

Customizable Goal for Integer Variables

The class `IloCustomizableGoal` is a goal for integer variables based on filters. It implements a family of strategies based on filters. An instance of `IloCustomizableGoal` is created with an array of variables to perform the search on:

```
IloCustomizableGoal d = IloCustomizableGoal(env, x);
```

Variables and value filters can then be added to this goal to define a search strategy.

Variable Filters

A filter takes as input an array of variables. It evaluates them once and selects a subset. The most basic filter contains an evaluator (see the concept *Selectors* in the *IBM ILOG Solver Reference Manual*). It evaluates each variable and selects the variables that have the smallest evaluation.

For example, if we consider the evaluator that returns the domain size:

```
ILOEVALUATOR0(SizeEvaluator, IlcIntVar, x) {
  return x.getSize();
}
```

we can build a basic filter with it and add it to an instance `g` of `IloCustomizableGoal` with the call:

```
g.addFilter(SizeEvaluator(env));
```

It will filter variables and keep only the one that has the smallest domain. Ties can be broken by adding another filter that will select variables from those remaining after filtering with `SizeEvaluator`. If there are still ties and no filter to break them, then the variable selected is the one having the smallest index in the array `vars`.

Other ways of selecting variables are available. A filter added with the following member function selects at least `number` variables among the best ones:

```
void IloCustomizableGoal::addMinNumberFilter(IloEvaluator<IlcIntVar> e,
                                             IlcFloat number);
```

The following member function selects a number of variables that is proportional to the size of the input. The value `proportion` must be strictly greater than zero and less than or equal to 1:

```
void IloCustomizableGoal::addMinProportionFilter(IloEvaluator<IlcIntVar> e,
                                                 IlcFloat proportion);
```

The following function selects variables that are at a distance of at most `tol` from the best evaluation:

```
void IloCustomizableGoal::addAbsoluteToleranceFilter(IloEvaluator<IlcIntVar> e,
                                                     IlcFloat tol);
```

The following function does the same but the tolerance is relative:

```
void IloCustomizableGoal::addRelativeToleranceFilter(IloEvaluator<IlcIntVar> e,
                                                     IlcFloat tol);
```

To illustrate the effects of different filters assume a filter with an input of 10 variables whose evaluations are:

Table 21.1 *Filter with an input of 10 variables*

x1	x2	x3	x4	x5	x6	x7	x8	x9
1.1	1.0	1.1	1.7	1.1	1.0	1.2	1.3	1.5

The smallest evaluation is 1.0. A basic filter will select two variables: `x2` and `x6` because they have the same —best—evaluation. A filter with an absolute tolerance of 0.1 will select

all variables having an evaluation between 1.0 (the best one) and 1.1. namely x1, x2, x3, x5, and x6. A filter with a relative tolerance of 0.2 will select six variables: x1, x2, x3, x4, x5, and x6. A filter with a minimum number of three will select x1, x2, x3, x5, and x6. Note that in this case we had to keep five variables because there are only two variables with the best evaluation and x1, x3, and x5 have the second best evaluation.

There are a certain number of already defined functions returning evaluators, among them the following, which return, respectively, the impact, the local impact, and the domain size of a variable:

```
IloEvaluator<IlcIntVar> IlcImpactVarEvaluator(IloEnv env);
IloEvaluator<IlcIntVar> IlcImpactVarEvaluator(IloSolver solver);
IloEvaluator<IlcIntVar> IlcLocalImpactVarEvaluator(IloEnv env, IloInt depth);
IloEvaluator<IlcIntVar> IlcLocalImpactVarEvaluator(IloSolver solver,
                                                    IloInt depth);
IloEvaluator<IlcIntVar> IlcSizeVarEvaluator(IloEnv env);
IloEvaluator<IlcIntVar> IlcSizeVarEvaluator(IloSolver solver);
```

There is also the following function, which returns a number between 0 and 1. This evaluator is used to build a filter that chooses randomly to break remaining ties.:

```
IloEvaluator<IlcIntVar> IlcRandomVarEvaluator(IloEnv env);
IloEvaluator<IlcIntVar> IlcRandomVarEvaluator(IloSolver solver);
```

Value Filters

Filters over values are defined in the same way variable evaluators are. However, value evaluators need to have the selected variable in a context. For example, if we consider the evaluator that returns the impact of a variable:

```
ILOCTXEVALUATOR0(IlcImpactValueEvaluator, IlcInt, value, IlcIntVar, y) {
  return y.getSolver().getImpact(y, value);
}
```

we can build a basic filter with it and add it to an instance g of IloCustomizableGoal with the call

```
g.addFilter(ImpactValueEvaluator(env));
```

As for variable filters, there are a number of already defined functions returning evaluators, such as:

```
IloEvaluator<IlcInt> IlcImpactValueEvaluator(IloEnv env);
IloEvaluator<IlcInt> IlcImpactValueEvaluator(IloSolver solver);
IloEvaluator<IlcInt> IlcRandomValueEvaluator(IloEnv env);
IloEvaluator<IlcInt> IlcRandomValueEvaluator(IloSolver solver);
```

which return, respectively, the impact of a value and the domain size of a variable.

Selection

An instance of `IloCustomizableGoal` is created with an array of variables to perform the search on:

```
IloCustomizableGoal d = IloCustomizableGoal(env, x);
```

By default the goal contains filters to perform selection in the following way:

1. Select a variable by:
   - getting the variable with the largest impact
   - breaking ties randomly
2. Select a value for the variable selected at Step 1 by:
   - getting the value with the smallest impact
   - breaking ties randomly

If we want to change these defaults one can add filters. An addition of a variable filter will discard the default filters for variables. An addition of a value filter will discard the default filters for the values.

To define a goal that selects at least five variables with the largest impact and then selects, among these promising variables, those having the largest local impact and that, finally breaks ties randomly, you can write:

```
IloEvaluator<IlcIntVar> e1 = -IlcImpactVarEvaluator(env);
IloEvaluator<IlcIntVar> e2 = -IlcLocalImpactVarEvaluator(env, -1);
IloEvaluator<IlcIntVar> e3 = IlcRandomVarEvaluator(env);
d.addMinNumberFilter(e1, 5);
d.addFilter(e2);
d.addFilter(e3);
```

To select the value that has the smallest impact and to break ties randomly you can write:

```
IloEvaluator<IlcInt> e4 = IlcImpactValueEvaluator(env);
IloEvaluator<IlcInt> e5 = IlcRandomValueEvaluator(env);
d.addFilter(e4);
d.addFilter(e5);
```

Of course the class `IloCustomizableGoal` is not devoted to strategies based on impacts only but can also be used to select variables according to other criteria like the domain size, the variable degree, or any evaluator that a user can define.

Restarts

As mentioned in the previous section, impacts are learned during search from the observation of domain reduction. As the search progresses, more accurate impacts will be

found and you may need to restart the search to make better choices at the beginning where this is crucial.

Additionally, when using randomness to break ties between equivalently good choices, depth-first search will try another good choice only when the whole subtree will be explored. It may be worth restarting the search from time to time in order to give different parts of the search tree that are equally promising an equal chance to be visited.

For this purpose, the following function creates a goal that calls `g` until the number of fails of this goal reaches `failLimit`, then it sets `failLimit` to `failLimit*factor` and calls `g` again:

```
IloGoal IloRestartGoal(IloEnv env,
                       IloGoal g,
                       IlcInt failLimit,
                       IlcFloat factor = 1.0);
```

For example, in the `YourSolverHome\examples\src\magicsq_impact.cpp` file, the default goal can be restarted with the fail limit of 1000 for each run by solving the goal:

```
 case 1: g =    IloInitializeImpactGoal(env, -1)
               && IloRestartGoal(env, IloCustomizableGoal(env, square), 1000);
```

C H A P T E R

22

Advanced Modeling with Set Variables: Configuring Tankers

In this lesson, you will learn more about how to:

- model with set variables
- use the function `IloFunction` to define accessors
- use the constraints `IloEqSum`, `IloEqMin`, `IloEqMax`, `IloEqUnion`, and `IloEqPartition`

This chapter uses constraints on *set variables* to solve a configuration problem of how to fill tanks and configure them on trucks to meet customer orders for various products. One of the main advantages of set variables is that they facilitate object-oriented modeling of problem knowledge. A set variable is capable of representing one-to-many (that is, 1–N) relations between objects with a single variable. In that way, it is possible to encapsulate in a class all the attributes, relations, and constraints corresponding to a family of objects. You take advantage of that capability in the object model you design for this configuration problem.

Describe

Let's assume you have a collection of customer orders to fulfill where the problem is to compute the minimal set of trucks needed to deliver the orders to the customers.

Our imaginary company is able to deliver five types of products. Each order defines the quantity to deliver for each type of product.

A truck is composed of a set of tanks. A truck can be configured with at most five different tanks. There are five types of tanks that correspond to five different capacities.

You assign an order to one and only one truck. Within an order, you must assign the products to the tanks of the truck. You must assign a product in an order to one and only one tank. In a single tank, you can mix the products of different orders, but all these products mixed in one tank must be of the same type.

There is a limit on the overall capacity of each truck: a truck cannot be loaded over 12000 units. A tank cannot be loaded over its capacity.

There is a limit on the types of products that you can assign to a truck: a truck cannot contain more than three different types of products.

Two of the products are incompatible. That is, they cannot be transported in the same truck.

Model

As you design an object model for this problem, you consider the classes, the attributes, and the relations to declare as well as the constraints to define on them. The knowledge contained in the problem description (that is, the attributes, relations, and constraints) can be encapsulated in a few classes that you define here. Constraints are defined in class constructors so that they are automatically added for each new instance of the class.

You can use these different classes to model the problem:

- `OrderI` represents the orders to fulfill on trucks
- `ProductI` represents the quantity of a given type of product in an order
- `TruckI` represents a truck
- `TankI` represents a tank on a truck

After you define those classes, you will define accessors for their attributes so that the attributes can be used directly by constraints on set variables.

Declaring the variables

To represent the orders the imaginary company receives from customers, you first define the class `OrderI`, identifying each order uniquely and indicating its products, like this:

```
class OrderI {
private:
  IloInt       _id;
  IloInt       _quantity;
  IloAnySetVar _products;
public:
  OrderI(IloInt id) : _id(id), _quantity(0) {}

  IloInt       getId()       const { return _id;  }
  IloInt       getQuantity() const { return _quantity; }
  void         setQuantity(IloInt quantity) { _quantity = quantity; }
  IloAnySetVar getProducts() const { return _products; }

  void makeProductVar(IloEnv env, IloAnyArray prods);

  void printName(ostream& out) const;
};
```

You group orders (instances of `OrderI`) into a global array (`Orders`) to represent all the orders to fulfill.

To represent the products demanded within an order, you define the class `ProductI`, indicating which order, which type of product, and which quantity, like this:

```
class ProductI {
protected:
  Order    _order;
  IloInt   _type;
  IloInt   _quantity;
public:
  ProductI(Order order, IloInt type, IloInt quantity)
    :_order(order) ,_type(type), _quantity(quantity) {}

  Order    getOrder() const    { return _order; }
  IloInt   getType() const     { return _type; }
  IloInt   getQuantity() const { return _quantity; }

  void printName(ostream& out) const;
};
```

The five different types of products are represented by the integers {1, 2, 3, 4, 5}. You say that 3 and 4 are the two types that cannot be transported in the same truck.

You also represent the trucks by a class—`TruckI`—with a unique identifier for each truck, an indicator of whether or not that truck is used, a *set variable* to show which orders are

filled by that truck, and another *set variable* to indicate which tanks make up the truck. You define that class like this:

```
class TruckI {
private:
  IloInt            _id;
  IloBoolVar         _used;
  IloAnySetVar      _orders;
  IloArray<Tank>    _tanks;
  IloAnySetVar      _tankVar;
public:
  TruckI(IloModel model, IloInt id);

  IloInt             getId() const        { return _id; }
  IloBoolVar          getUsed() const      { return _used; }
  IloAnySetVar       getOrders() const    { return _orders; }
  IloInt             getNbTanks() const   { return 5; }
  IloArray<Tank>     getTanks() const     { return _tanks; }
  IloAnySetVar       getTankVar() const   { return _tankVar; }

  void display(IloSolver solver) const;
};
```

To represent the tanks that make up a truck, you first enumerate the *capacity* of the different types of tanks: `{1000, 2000, 3000, 4000, 5000}`. Then you define the class `TankI`, like this:

```
class TankI {
protected:
  IloInt       _id;
  IloIntVar    _type;
  IloIntVar    _load;
  IloInt       _capaMax;
  IloAnySetVar _prodOrders;
public:
  TankI(IloModel model, IloInt id, IloInt capaMax);

  IloInt       getId() const            { return _id; }
  IloIntVar    getType() const          { return _type; }
  IloIntVar    getLoad() const          { return _load; }
  IloInt       getCapacityMax() const   { return _capaMax; }
  IloAnySetVar getProductOrders() const { return _prodOrders; }

  void display  (IloSolver solver) const;
  void printName(ostream& out) const {
    out << "Tank#" << _id;
  }
};
```

Using these classes, you can express many of the constraints in the problem through their constructors, but before you add the constraints, you must look more closely at how to access attributes of those classes when their objects serve in the model as elements in the *domain* of a set.

Defining accessors for attributes of set elements

In this model, as you consider a set of tanks on a truck, there are attributes of a tank (for example, its capacity, its current load, the type of product it currently contains) that you want to access in the constraints. Likewise, as you assign a set of orders to the tanks of a truck, you know that there are attributes of the orders that figure in the problem constraints, so you need ways of accessing those attributes. In short, you need set variables to represent constrained relations between an object (a truck, a tank, an order) and a set of related objects (the set of tanks of a truck, the set of orders fulfilled by a truck, and so on). You also need to access the attributes of the related objects to be able to constrain the relation; for example, the cumulative quantities assigned to a truck must be less than or equal to the capacity of the truck. Solver offers you a useful way of accessing such set object attributes.

To define such accessors, use the predefined Solver class `IloFunction`.

Accessing the quantity of an order: OQuantityI

To access the quantity of an order, define `OQuantityI`, like this:

```
class OQuantityI : public IloFunctionI<IloAny,IloInt> {
public:
  OQuantityI(IloEnvI* e) : IloFunctionI<IloAny,IloInt>(e) {}
  IloInt getValue(IloAny elt);
};

IloInt OQuantityI::getValue(IloAny elt) {
  return ((Order)elt)->getQuantity();
}

IloFunction<IloAny,IloInt> OQuantity(IloEnv env) {
  return new (env) OQuantityI(env.getImpl());
}
```

Accessing the load in a tank: TankLoadI

To access the load in a tank, define `TankLoadI`, like this:

```
class TankLoadI : public IloFunctionI<IloAny,IloIntVar> {
public:
  TankLoadI(IloEnvI* e) : IloFunctionI<IloAny,IloIntVar>(e) {}
  IloIntVar getValue(IloAny elt);
};

IloIntVar TankLoadI::getValue(IloAny elt) {
  return ((Tank)elt)->getLoad();
}

IloFunction<IloAny,IloIntVar> TankLoad(IloEnv env) {
  return new (env) TankLoadI(env.getImpl());
}
```

Accessing the products in a tank: TankProductI

To access the products in a tank, define `TankProductI`, like this:

```
class TankProductI : public IloFunctionI<IloAny,IloAnySetVar> {
public:
  TankProductI(IloEnvI* e) : IloFunctionI<IloAny, IloAnySetVar>(e) {}
  IloAnySetVar getValue(IloAny elt);
};

IloAnySetVar TankProductI::getValue(IloAny elt) {
  return ((Tank)elt)->getProductOrders();
}

IloFunction<IloAny,IloAnySetVar> TankProducts(IloEnv env) {
  return new (env) TankProductI(env.getImpl());
}
```

Adding constraints

With the classes you have defined for this model, and with the accessors for attributes of objects that are elements in the domains of sets, you can now represent the constraints of the problem. In fact, most of the constraints mentioned in the problem description can be added to the model by the constructors of the classes you just defined. You will be using the predefined Concert Technology constraints `IloEqSum`, `IloEqMin`, `IloEqMax`, and `IloEqUnion` to define the problem constraints on set variables.

Constraints in the Tank constructor

In the assignments of orders to trucks, you must assign products to tanks while respecting the constraint that you must not mix products of different types within a tank. To make sure a tank contains only products of the same type, regardless of which order the product fulfills, you add these constraints in the constructor of `Tank`:

```
  model.add( IloEqMin(env,  _prodOrders, _type, PType(env)) );
  model.add( IloEqMax(env,  _prodOrders, _type, PType(env)) );
```

The capacity of a tank is taken into account in the constructor of the `_load` variable. The following constraint uses `IloEqSum` to compute the load of a given tank:

```
  model.add( IloEqSum(env,  _prodOrders, _load, PQuantity(env)) );
```

Constraints in the Truck constructor

In the model, a truck is used if an order is assigned to it. If a truck is not used, you must not assign an order to it. As you continue to define constraints in the class constructors, you use

the cardinality variable (supplied by the predefined Solver function `IloCard`) of the set variable `_orders` to indicate whether or not a particular truck is used, like this:

```
_used = IloBoolVar(env);
model.add( _used == (IloCard(_orders) > 0));
```

Likewise, in the constructor of `TruckI`, you express the idea that the two products 3 and 4 are incompatible and thus cannot be loaded on the same truck by adding a constraint on the set of different product types assigned to the tanks of the trucks. You define a constrained set variable `ttanks`; its domain represents the types of the possible products. Then, you use the predefined Concert Technology function `IloEqUnion` to constrain `ttanks` to correspond to the types of the possible products for the possible tanks of the truck at hand.

```
IloNumSetVar ttanks(env, IloIntArray(env, 5, 1, 2, 3, 4, 5));
model.add(IloEqUnion(env, _tankVar, ttanks, TankType(env)));
model.add( ! (IloMember(env, 3, ttanks) && IloMember(env, 4, ttanks)) );
```

You define a set variable `tprodVar` to represent the products in the tanks of the truck. To do so, you use the predefined Solver constraint `IloEqUnion` again with `TankProducts`, the accessor you defined in “Accessing the products in a tank: TankProductI” on page 386, in this way:

```
IloAnySetVar tprodVar(env, prodOrdersArray);
model.add( IloEqUnion(env, _tankVar, tprodVar, TankProducts(env) ));
```

Likewise, you define a constrained set variable `oprodVar` to represent the products in the orders fulfilled by a truck, and you use `IloEqUnion` with `OrderProducts` to take the union of products in orders assigned to the truck, like this:

```
IloAnySetVar oprodVar(env, prodOrdersArray);
model.add( IloEqUnion(env, _orders, oprodVar,  OrderProducts(env) ));
```

Then you combine those two set variables in a constraint to make sure that the products filling the tanks on a truck are exactly the products demanded in the orders fulfilled by that truck.

```
model.add( tprodVar == oprodVar );
```

To be sure that all the products of the orders fulfilled by the truck are assigned to one and only one tank of the truck, you define a partition constraint between the previously defined set variable `oprodVar` (that represents the products assigned globally to the truck through the orders) and the set variable `prodVars[i]` of each possible tank of the truck (that

represents the products assigned to that particular tank). You use the predefined Solver constraint `IloEqPartition` like this:

```
model.add( IloEqPartition(env,  oprodVar, prodVars ) );
```

Also in the constructor of `TruckI`, you enforce the stipulation that no more than three different types of products can be loaded on the same truck. To do so, you directly constrain the cardinality `IloCard` of the variable `ttanks` which represents this set of types, like this:

```
model.add( IloCard( ttanks ) <= 3 );
```

To insure that the 12000-unit limit on the maximum load of a truck is respected, you use the predefined constraint `IloEqSum` to accumulate the quantities specified in orders assigned to a given truck. You add a constraint that the cumulative load must be less than or equal to the limit, and you also add a constraint (again using `IloEqSum`) that the load on the truck is the same as the sum of the loads in its tanks. All those steps occur in the constructor of `TruckI`, like this:

```
IloIntVar load(env, 0, 12000);

model.add( IloEqSum(env,  _orders, load,  OQuantity(env) ));
model.add( IloEqSum(env,  _tankVar, load, TankLoad(env) ));
```

Solve

With that model, you know that in the worst case, if you have *N* orders, and you assign one order per truck, you surely have a solution using *N* trucks, though it is not necessarily an optimal one. To find an optimal solution, you first look for an assignment of orders to trucks that uses at most *N* trucks. Let's say it uses only *N'* trucks, where *N'* is less than or equal to *N*. If you find such an assignment, you look for a solution using at most *N'-1* trucks. If you are unable to find such an *N'*, the optimal is *N*.

The algorithm looks like this:

```
For N orders, M represents the number of trucks needed.
M = N
Loop {
    Eliminate N - M trucks. Mark them as unused.
    Search for an assignment on M' trucks with M' <= M
    If (not found)
    then
        Optimal is at M + 1
    else
        M = M' - 1
}
```

The algorithm to search for an assignment to at most *M* trucks is decomposed into two nested phases:

- Assign orders to trucks.
- Each time an order is assigned, assign its products to tanks on the chosen truck.

To guide that search, you implement two goals. As the next branch to explore, one goal chooses the truck that is hardest to fill—hardest in the sense that it has the smallest domain. The other goal chooses the tank that is hardest to fill, also in the sense that it has the smallest domain.

Goal to choose truck

As you search for an assignment of orders to trucks, you first try to assign the largest order, that is, the order with the greatest quantity. For a given order, you first try the "hardest" truck; that is, the one with the smallest domain; in other words, the one for which you have the fewest possible choices of orders that fit.

Here is the goal to choose the hardest truck to fill:

```
Truck ChooseTruck(IloSolver solver,
                   Order order,
                   IloInt nbTrucks, IloArray<Truck> trucks,
                   IlcBool aNewOne) {
  Truck best=0;
  IloInt eval = IlcIntMax;
  for(IloInt i=0; i < nbTrucks; i++) {
    IlcAnySetVar oVar = solver.getAnySetVar(trucks[i]->getOrders());
    IloInt cardMin = IlcCard(oVar).getMin();
    if (aNewOne) {
      if (cardMin > 0)
        continue;
    } else {
      if (cardMin == 0)
        continue;
      if (oVar.isRequired(order))
        return trucks[i];
    }
    if (! oVar.isPossible(order))
      continue;
    IloInt size = oVar.getPossibleSet().getSize();
    if (size < eval) {
      eval = size;
      best = trucks[i];
    }
  }
  return best;
}

ILCGOAL3(AssignOrder, Order, order, IloInt, nbTrucks, IloArray<Truck>, trucks)
{
  IloSolver solver = getSolver();
  //Choose first an already used truck
  Truck t = ChooseTruck(solver, order, nbTrucks, trucks, IlcFalse);
```

```
  if (t == 0) {
    //Choose a new truck
    t = ChooseTruck(solver, order, nbTrucks, trucks, IlcTrue);
  }
  if (t == 0)
    fail();

  IlcAnySetVar tVar = solver.getAnySetVar(t->getOrders());
  if (tVar.isRequired(order))
    return AssignProducts(solver, order, t);
  else
    return IlcOr(IlcAnd(IlcMember(order, tVar),
                        AssignProducts(solver, order, t)),
                 IlcAnd(IlcNotMember(order, tVar),
                        this));
}

ILOCPGOALWRAPPER3(IloAssignOrder, solver, Order, ord, IloInt, nbTrucks,
IloArray<Truck>, trucks) {
  return AssignOrder(solver, ord, nbTrucks, trucks);
}

IloGoal IloAssignOrders (IloEnv env, IloAnyArray orders, IloInt nbTrucks,
IloArray<Truck> trucks) {
  IloGoal goal = IloGoalTrue(env);
  for(IloInt i = 0; i < orders.getSize(); i++)
    goal = goal && IloAssignOrder(env, (Order)orders[i], nbTrucks, trucks);
  return goal;
}

IloGoal IloEliminateTrucks (IloEnv env, IloInt nbTrucks, IloInt nbMax,
IloArray<Truck> trucks) {
  IloGoal goal = IloGoalTrue(env);
  for(IloInt i = nbTrucks; i > nbMax; i--)
    goal = goal && IloAddConstraint(trucks[i-1]->getUsed() == 0);
  return goal;
}
```

Goal to choose tank

Once you have assigned an order to a truck, you distribute the products demanded by that order among the tanks on that truck. For a given product, you choose the “hardest” tank; that is, the one for which you have the fewest possible choices of products; in other words, the tank with the smallest domain.

The goal to choose the hardest tank to fill looks like this:

```
Tank ChooseTank(IloSolver solver, Product p, IlcAnySet tankset) {
  Tank best=0;
  IloInt eval = IlcIntMax;
  for(IlcAnySetIterator it(tankset); it.ok(); ++it) {
    Tank tank = (Tank)(*it);
    IlcAnySetVar pvar = solver.getAnySetVar(tank->getProductOrders());
    if (pvar.isRequired(p))
      return tank;
```

```
    if (! pvar.isPossible(p))
      continue;

    IloInt size = pvar.getPossibleSet().getSize();
    if (size < eval) {
      eval = size;
      best = tank;
    }
  }
  return best;
}

ILCGOAL2(AssignProduct, Product, product, Truck, truck) {
  IloSolver solver = getSolver();
  //Choose an already used tank
  IlcAnySetVar stanks = solver.getAnySetVar(truck->getTankVar());
  Tank tank = ChooseTank(solver, product, stanks.getRequiredSet());
  if (tank == 0)
    //Choose a new tank
    tank = ChooseTank(solver, product, stanks.getPossibleSet());
  if (tank == 0)
    fail();

  IlcAnySetVar stankvar = solver.getAnySetVar(tank->getProductOrders());
  if (stankvar.isRequired(product))
    return 0;
  return IlcOr(IlcMember(product, stankvar),
               IlcAnd(IlcNotMember(product, stankvar),
                      this));
}

ILCGOAL2(AssignProducts, Order, order, Truck, truck) {
  IloSolver solver = getSolver();
  IlcAnySetVar prodVar = solver.getAnySetVar(order->getProducts());
  IlcGoal g = IlcGoalTrue(solver);
  for(IlcAnySetIterator it(prodVar.getRequiredSet()); it.ok(); ++it)
    g = IloAndGoal(solver, g,  AssignProduct(solver, (Product)(*it), truck));
  return g;
}
```

Complete program

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

```
#include <ilsolver/ilosolver.h>

ILOSTLBEGIN


class OrderI;
class ProductI;

typedef OrderI* Order;
```

```
typedef ProductI* Product;

//---------------------------------------------------------
// Data
//---------------------------------------------------------

const IloInt NbProducts = 5;
const IloInt NbOrders   = 11;

//Sorted in decreasing order of total quantity
const IloInt Quantities[NbOrders][NbProducts] = {
  { 2000, 3000, 5000,    0,    0 },
  {    0, 3000, 2000,    0, 4000 },
  { 2000,    0,    0,    0, 4000 },
  {    0, 2500,    0,  500, 1000 },
  { 1000,    0,  500,    0, 2000 },
  {    0,  500,    0, 2000,  500 },
  {    0,    0,    0,    0, 3000 },
  {    0, 1000,    0,    0, 1000 },
  { 1000,  500,    0,    0,    0 },
  { 1500,    0,    0,    0,    0 },
  {    0,  500,  500,    0,    0 }
};

Order   Orders[NbOrders];
Product Products[NbOrders*NbProducts];


//---------------------------------------------------------
class ProductI {
protected:
  Order    _order;
  IloInt   _type;
  IloInt   _quantity;
public:
  ProductI(Order order, IloInt type, IloInt quantity)
    :_order(order) ,_type(type), _quantity(quantity) {}

  Order    getOrder() const    { return _order; }
  IloInt   getType() const     { return _type; }
  IloInt   getQuantity() const { return _quantity; }

  void printName(ostream& out) const;
};

IloFunction<IloAny,IloInt> PType(IloEnv env);
IloFunction<IloAny,IloInt> PQuantity(IloEnv env);

//---------------------------------------------------------
class OrderI {
private:
  IloInt       _id;
  IloInt       _quantity;
  IloAnySetVar _products;
public:
  OrderI(IloInt id) : _id(id), _quantity(0) {}

  IloInt       getId()       const { return _id;  }
```

```
  IloInt       getQuantity() const { return _quantity; }
  void         setQuantity(IloInt quantity) { _quantity = quantity; }
  IloAnySetVar getProducts() const { return _products; }

  void makeProductVar(IloEnv env, IloAnyArray prods);

  void printName(ostream& out) const;
};

IloFunction<IloAny,IloInt> OQuantity(IloEnv env);
IloFunction<IloAny,IloAnySetVar> OrderProducts(IloEnv env);

//-------------------------------------------------------
void OrderI::makeProductVar(IloEnv env, IloAnyArray prods) {
  _products = IloAnySetVar(env, prods);
  for(IloInt i = 0; i < NbOrders; i++) {
    for(IloInt j = 0; j < NbProducts; j++) {
      Product p = Products[i*NbProducts + j];
      if (p->getQuantity() == 0)
        _products.removePossible(p);
      else if (_id == i+1)
        _products.addRequired(p);
      else
        _products.removePossible(p);
    }
  }
}

void OrderI::printName(ostream& out) const {
  out << "Order#" << _id;
}

//-------------------------------------------------------

void ProductI::printName(ostream& out) const {
  _order->printName(out);
  out << ".P" << _type;
}

//-------------------------------------------------------

IloAnyArray MakeOrders(IloEnv env) {
  IloInt i;

  for(i = 0; i < NbOrders; i++) {
    Orders[i] = new (env) OrderI(i+1);
    IloInt quantity = 0;
    for (IloInt j = 0; j < NbProducts; j++) {
      IloInt q = (IloInt)Quantities[i][j];
      quantity += q;
      Products[i*NbProducts + j] = new (env) ProductI(Orders[i], j+1, q);
    }
    Orders[i]->setQuantity(quantity);
  }

  IloAnyArray prodOrdersArray(env, NbOrders*NbProducts);
  for(i = 0; i < NbOrders*NbProducts; i++)
    prodOrdersArray[i] = (IloAny)Products[i];
```

```
  for(i = 0; i < NbOrders; i++)
    Orders[i]->makeProductVar(env,prodOrdersArray);

  IloAnyArray ordersArray(env, NbOrders);
  for(i = 0; i < NbOrders; i++)
    ordersArray[i] = (IloAny)Orders[i];

  return ordersArray;
}

//-------------------------------------------------------
class TankI {
protected:
  IloInt       _id;
  IloIntVar    _type;
  IloIntVar    _load;
  IloInt       _capaMax;
  IloAnySetVar _prodOrders;
public:
  TankI(IloModel model, IloInt id, IloInt capaMax);

  IloInt       getId() const            { return _id; }
  IloIntVar    getType() const          { return _type; }
  IloIntVar    getLoad() const          { return _load; }
  IloInt       getCapacityMax() const   { return _capaMax; }
  IloAnySetVar getProductOrders() const { return _prodOrders; }

  void display  (IloSolver solver) const;
  void printName(ostream& out) const {
    out << "Tank#" << _id;
  }
};
typedef TankI* Tank;

IloFunction<IloAny,IloIntVar>     TankLoad(IloEnv env);
IloFunction<IloAny,IloIntVar>     TankType(IloEnv env);
IloFunction<IloAny,IloAnySetVar>  TankProducts(IloEnv env);

//-------------------------------------------------------
TankI::TankI(IloModel model, IloInt id, IloInt capaMax)
  :_id(id), _capaMax(capaMax) {

  IloEnv env = model.getEnv();

  _type = IloIntVar(env, 1, 5);
  _load = IloIntVar(env, 0, capaMax);

  IloAnyArray prodOrdersArray = IloAnyArray(env, NbOrders*NbProducts);
  for (IloInt i = 0; i < NbOrders*NbProducts; i++)
    prodOrdersArray[i] = (IloAny)Products[i];

  _prodOrders = IloAnySetVar(env, prodOrdersArray);

  //All the products assigned to a tank must belong
  //to the same type of product

  model.add( IloEqMin(env,  _prodOrders, _type, PType(env)) );
```

```
  model.add( IloEqMax(env,  _prodOrders, _type, PType(env)) );
  //The loading of the tank cannot exceed its capacity
  model.add( IloEqSum(env,  _prodOrders, _load, PQuantity(env)) );
}

void TankI::display(IloSolver solver) const {
  if (solver.getAnySetVar(_prodOrders).getCardinality().getMax() == 0)
    return; //Empty
  solver.out() << "       <";
  printName(solver.out());
  solver.out() << ">" << endl
      << "              --> Capacity  = " << _capaMax << endl
      << "              --> Type      = " << solver.getIntVar(_type) << endl
      << "              --> Load      = " << solver.getIntVar(_load) << endl
      << "              --> Orders    = (";
  for(IlcAnySetIterator it(solver.getAnySetVar(_prodOrders).getRequiredSet());
      it.ok(); ++it) {
    ((Product)(*it))->printName(solver.out());
    solver.out() << " ";
  }
  solver.out() << ")" << endl ;
}

//-------------------------------------------------------
class TruckI {
private:
  IloInt             _id;
  IloBoolVar          _used;
  IloAnySetVar       _orders;
  IloArray<Tank>     _tanks;
  IloAnySetVar       _tankVar;
public:
  TruckI(IloModel model, IloInt id);

  IloInt              getId() const        { return _id; }
  IloBoolVar           getUsed() const      { return _used; }
  IloAnySetVar        getOrders() const    { return _orders; }
  IloInt              getNbTanks() const   { return 5; }
  IloArray<Tank>      getTanks() const     { return _tanks; }
  IloAnySetVar        getTankVar() const   { return _tankVar; }

  void display(IloSolver solver) const;
};

typedef TruckI* Truck;

TruckI::TruckI(IloModel model, IloInt id) :_id(id) {
  IloEnv env = model.getEnv();
  IloInt i;

  IloAnyArray ordersArray = IloAnyArray(env, NbOrders);
  for (i = 0; i < NbOrders; i++)
    ordersArray[i] = (IloAny)Orders[i];

  _orders = IloAnySetVar(env, ordersArray);

  _tanks = IloArray<Tank> (env, getNbTanks());
```

```
  IloAnySetVarArray prodVars(env, getNbTanks());
  for(i = 0; i < getNbTanks(); i++) {
    _tanks[i] = new (env) TankI(model, i+1, (i+1)*1000);
    prodVars[i] = _tanks[i]->getProductOrders();
  }

  IloAnyArray tanksArray = IloAnyArray(env, getNbTanks());
  for (i = 0; i < getNbTanks(); i++)
    tanksArray[i] = (IloAny)_tanks[i];

  _tankVar = IloAnySetVar(env, tanksArray);

  //The truck is used if one order is assigned to it
  _used = IloBoolVar(env);
  model.add( _used == (IloCard(_orders) > 0));

  //Maximal loading of the truck
  IloIntVar load(env, 0, 12000);

  model.add( IloEqSum(env,  _orders, load,  OQuantity(env) ));
  model.add( IloEqSum(env,  _tankVar, load, TankLoad(env) ));

  IloAnyArray prodOrdersArray = IloAnyArray(env, NbOrders*NbProducts);
  for (i = 0; i < NbOrders*NbProducts; i++)
    prodOrdersArray[i] = (IloAny)Products[i];

  //Products assigned to the tanks of the truck
  IloAnySetVar tprodVar(env, prodOrdersArray);
  model.add( IloEqUnion(env, _tankVar, tprodVar, TankProducts(env) ));

  //Products of the orders assigned to the truck
  IloAnySetVar oprodVar(env, prodOrdersArray);
  model.add( IloEqUnion(env, _orders, oprodVar,  OrderProducts(env) ));

  //Product assigned to the tanks are the products of the
  //orders assigned to the truck
  model.add( tprodVar == oprodVar );

  //All the products must be assigned to one and only one tank
  model.add( IloEqPartition(env,  oprodVar, prodVars ) );


  //The types of tanks on a truck
  IloNumSetVar ttanks(env, IloIntArray(env, 5, 1, 2, 3, 4, 5));
  model.add(IloEqUnion(env, _tankVar, ttanks, TankType(env)));

  //A truck can be assigned at most 3 different types of products
  model.add( IloCard( ttanks ) <= 3 );
  //Product 3 and 4 are incompatibe
  model.add( ! (IloMember(env, 3, ttanks) && IloMember(env, 4, ttanks)) );
}

//-------------------------------------------------------

void TruckI::display(IloSolver solver) const {
  solver.out();
  if (! solver.getIntVar(_used).isBound())
    return;
```

```
  if (solver.getIntVar(_used).getValue() == 0)
        return;
  solver.out() << "<Truck T#" << _id << ">" << endl
     << "       --> Orders = (";
  for(IlcAnySetIterator it(solver.getAnySetVar(_orders).getRequiredSet());
      it.ok(); ++it) {
    ((Order)(*it))->printName(solver.out());
    solver.out() << " ";
  }
  solver.out() << ")" << endl;
  solver.out() << "       --> Tank      = (";
  for(IlcAnySetIterator itT(solver.getAnySetVar(_tankVar).getRequiredSet());
      itT.ok(); ++itT ) {
    ((Tank)(*itT))->printName(solver.out());
    solver.out() << " ";
  }
  solver.out() << ")" << endl;

  for(IlcAnySetIterator itT2(solver.getAnySetVar(_tankVar).getRequiredSet());
      itT2.ok(); ++itT2 ) {
    ((Tank)(*itT2))->display(solver);
  }
  solver.out() << endl;
}

//-------------------------------------------------------
// Solving goals
//-------------------------------------------------------

// Choose the tank the most hard to fill;
// with the smallest domain.
Tank ChooseTank(IloSolver solver, Product p, IlcAnySet tankset) {
  Tank best=0;
  IloInt eval = IlcIntMax;
  for(IlcAnySetIterator it(tankset); it.ok(); ++it) {
    Tank tank = (Tank)(*it);
    IlcAnySetVar pvar = solver.getAnySetVar(tank->getProductOrders());
    if (pvar.isRequired(p))
      return tank;
    if (! pvar.isPossible(p))
      continue;

    IloInt size = pvar.getPossibleSet().getSize();
    if (size < eval) {
      eval = size;
      best = tank;
    }
  }
  return best;
}

ILCGOAL2(AssignProduct, Product, product, Truck, truck) {
  IloSolver solver = getSolver();
  //Choose an already used tank
  IlcAnySetVar stanks = solver.getAnySetVar(truck->getTankVar());
  Tank tank = ChooseTank(solver, product, stanks.getRequiredSet());
  if (tank == 0)
    //Choose a new tank
```

```
    tank = ChooseTank(solver, product, stanks.getPossibleSet());
  if (tank == 0)
    fail();

  IlcAnySetVar stankvar = solver.getAnySetVar(tank->getProductOrders());
  if (stankvar.isRequired(product))
    return 0;
  return IlcOr(IlcMember(product, stankvar),
               IlcAnd(IlcNotMember(product, stankvar),
                      this));
}

ILCGOAL2(AssignProducts, Order, order, Truck, truck) {
  IloSolver solver = getSolver();
  IlcAnySetVar prodVar = solver.getAnySetVar(order->getProducts());
  IlcGoal g = IlcGoalTrue(solver);
  for(IlcAnySetIterator it(prodVar.getRequiredSet()); it.ok(); ++it)
    g = IloAndGoal(solver, g,  AssignProduct(solver, (Product)(*it), truck));
  return g;
}
// Choose the truck the most hard to fill;
// with the smallest domain.
Truck ChooseTruck(IloSolver solver,
                   Order order,
                   IloInt nbTrucks, IloArray<Truck> trucks,
                   IlcBool aNewOne) {
  Truck best=0;
  IloInt eval = IlcIntMax;
  for(IloInt i=0; i < nbTrucks; i++) {
    IlcAnySetVar oVar = solver.getAnySetVar(trucks[i]->getOrders());
    IloInt cardMin = IlcCard(oVar).getMin();
    if (aNewOne) {
      if (cardMin > 0)
        continue;
    } else {
      if (cardMin == 0)
        continue;
      if (oVar.isRequired(order))
        return trucks[i];
    }
    if (! oVar.isPossible(order))
      continue;
    IloInt size = oVar.getPossibleSet().getSize();
    if (size < eval) {
      eval = size;
      best = trucks[i];
    }
  }
  return best;
}

ILCGOAL3(AssignOrder, Order, order, IloInt, nbTrucks, IloArray<Truck>, trucks)
{
  IloSolver solver = getSolver();
  //Choose first an already used truck
  Truck t = ChooseTruck(solver, order, nbTrucks, trucks, IlcFalse);
  if (t == 0) {
    //Choose a new truck
```

```
    t = ChooseTruck(solver, order, nbTrucks, trucks, IlcTrue);
  }
  if (t == 0)
    fail();

  IlcAnySetVar tVar = solver.getAnySetVar(t->getOrders());
  if (tVar.isRequired(order))
    return AssignProducts(solver, order, t);
  else
    return IlcOr(IlcAnd(IlcMember(order, tVar),
                        AssignProducts(solver, order, t)),
                 IlcAnd(IlcNotMember(order, tVar),
                        this));
}

ILOCPGOALWRAPPER3(IloAssignOrder, solver, Order, ord, IloInt, nbTrucks,
IloArray<Truck>, trucks) {
  return AssignOrder(solver, ord, nbTrucks, trucks);
}

IloGoal IloAssignOrders (IloEnv env, IloAnyArray orders, IloInt nbTrucks,
IloArray<Truck> trucks) {
  IloGoal goal = IloGoalTrue(env);
  for(IloInt i = 0; i < orders.getSize(); i++)
    goal = goal && IloAssignOrder(env, (Order)orders[i], nbTrucks, trucks);
  return goal;
}

IloGoal IloEliminateTrucks (IloEnv env, IloInt nbTrucks, IloInt nbMax,
IloArray<Truck> trucks) {
  IloGoal goal = IloGoalTrue(env);
  for(IloInt i = nbTrucks; i > nbMax; i--)
    goal = goal && IloAddConstraint(trucks[i-1]->getUsed() == 0);
  return goal;
}

//-------------------------------------------------------
// Main program
//-------------------------------------------------------

int main () {
  IloEnv env;
  try {
    IloModel model(env);

    IloAnyArray  ordersArray = MakeOrders(env);
    IloAnySetVar orders(env, ordersArray);

    model.add(IloCard(orders) == NbOrders);

    //At most 1 Truck per order: Create initially NbOrders trucks

    IloArray<Truck> trucks(env, NbOrders);

    IloBoolVarArray usedVars(env, NbOrders);
    IloAnySetVarArray ovars(env, NbOrders);

    IloInt i;
```

```
  for(i=0; i < NbOrders; i++) {
    trucks[i] = new (env) TruckI(model, i+1);
    usedVars[i] = trucks[i]->getUsed();
    ovars[i] = trucks[i]->getOrders();
  }

  //All orders must be assigned to a truck

  model.add( IloEqPartition(env, orders, ovars) );

  //Cost Variable

  IloIntVar costVar(env, 0, NbOrders);

  model.add( costVar == IloSum(usedVars) );

  //Optimization

  IloSolver solver(model);

  IloInt nbMax = NbOrders+1;
  IloBool ok = IloTrue;
  IloGoal g;

  while (ok) {

    //Strategy: Eliminate unuseful trucks,
    //          Assign the orders to trucks, choosing the
    //          order with the biggest quantity first.

    g = IloEliminateTrucks(env, NbOrders, nbMax-1, trucks) &&
        IloAssignOrders(env, ordersArray, nbMax-1, trucks);

    ok = solver.solve(g);

    if (ok) {
      nbMax = (IloInt)solver.getValue(costVar);
      solver.out() << "--> Solution found at: "
                   << nbMax << " trucks." << endl;
    }
  }
  if (nbMax > NbOrders)
    solver.out() << "--> No Solution !" << endl;
  else {
    model.add(costVar == nbMax);

    g = IloEliminateTrucks(env, NbOrders, nbMax, trucks) &&
        IloAssignOrders(env, ordersArray, nbMax, trucks);

    solver.solve(g);

    solver.out() << endl;

    for(i = 0; i < NbOrders; i++)
      trucks[i]->display(solver);
  }
  solver.printInformation();
}
```

```
  catch (IloException& ex) {
    cout << "Error: " << ex << endl;
  }
  env.end();
  return 0;

}
//-------------------------------------------------------
// Accessors
//-------------------------------------------------------

class PTypeI : public IloFunctionI<IloAny,IloInt> {
public:
  PTypeI(IloEnvI* e) : IloFunctionI<IloAny,IloInt>(e)  {}
  IloInt getValue(IloAny elt);
};

IloInt PTypeI::getValue(IloAny elt) {
  return ((Product)elt)->getType();
}

IloFunction<IloAny,IloInt> PType(IloEnv env) {
  return new (env) PTypeI(env.getImpl());
}


class PQuantityI : public IloFunctionI<IloAny,IloInt> {
public:
  PQuantityI(IloEnvI* e) : IloFunctionI<IloAny,IloInt>(e) {}
  IloInt getValue(IloAny elt);
};

IloInt PQuantityI::getValue(IloAny elt) {
  return ((Product)elt)->getQuantity();
}

IloFunction<IloAny,IloInt> PQuantity(IloEnv env) {
  return new (env) PQuantityI(env.getImpl());
}

class OQuantityI : public IloFunctionI<IloAny,IloInt> {
public:
  OQuantityI(IloEnvI* e) : IloFunctionI<IloAny,IloInt>(e) {}
  IloInt getValue(IloAny elt);
};

IloInt OQuantityI::getValue(IloAny elt) {
  return ((Order)elt)->getQuantity();
}

IloFunction<IloAny,IloInt> OQuantity(IloEnv env) {
  return new (env) OQuantityI(env.getImpl());
}

class OrderProductI : public IloFunctionI<IloAny,IloAnySetVar> {
public:
  OrderProductI(IloEnvI* e) : IloFunctionI<IloAny, IloAnySetVar>(e)  {}
  IloAnySetVar getValue(IloAny elt);
```

```
};

IloAnySetVar OrderProductI::getValue(IloAny elt) {
  return ((Order)elt)->getProducts();
}

IloFunction<IloAny,IloAnySetVar> OrderProducts(IloEnv env) {
  return new (env) OrderProductI(env.getImpl());
}

class TankLoadI : public IloFunctionI<IloAny,IloIntVar> {
public:
  TankLoadI(IloEnvI* e) : IloFunctionI<IloAny,IloIntVar>(e) {}
  IloIntVar getValue(IloAny elt);
};

IloIntVar TankLoadI::getValue(IloAny elt) {
  return ((Tank)elt)->getLoad();
}

IloFunction<IloAny,IloIntVar> TankLoad(IloEnv env) {
  return new (env) TankLoadI(env.getImpl());
}

class TankTypeI : public IloFunctionI<IloAny,IloIntVar> {
public:
  TankTypeI(IloEnvI* e) : IloFunctionI<IloAny,IloIntVar>(e) {}
  IloIntVar getValue(IloAny elt);
};

IloIntVar TankTypeI::getValue(IloAny elt) {
  return ((Tank)elt)->getType();
}

IloFunction<IloAny,IloIntVar> TankType(IloEnv env) {
  return new (env) TankTypeI(env.getImpl());
}

class TankProductI : public IloFunctionI<IloAny,IloAnySetVar> {
public:
  TankProductI(IloEnvI* e) : IloFunctionI<IloAny, IloAnySetVar>(e) {}
  IloAnySetVar getValue(IloAny elt);
};

IloAnySetVar TankProductI::getValue(IloAny elt) {
  return ((Tank)elt)->getProductOrders();
}

IloFunction<IloAny,IloAnySetVar> TankProducts(IloEnv env) {
  return new (env) TankProductI(env.getImpl());
}
```

Output

Here is the output from the program.

```
--> Solution found at: 6 trucks.
--> Solution found at: 5 trucks.
--> Solution found at: 4 trucks.

<Truck T#1>
       --> Orders = (Order#1 Order#9 )
       --> Tank      = (Tank#1 Tank#2 Tank#3 Tank#4 Tank#5 )
       <Tank#1>
              --> Capacity  = 1000
              --> Type      = [2]
              --> Load      = [500]
              --> Orders    = (Order#9.P2 )
       <Tank#2>
              --> Capacity  = 2000
              --> Type      = [1]
              --> Load      = [2000]
              --> Orders    = (Order#1.P1 )
       <Tank#3>
              --> Capacity  = 3000
              --> Type      = [2]
              --> Load      = [3000]
              --> Orders    = (Order#1.P2 )
       <Tank#4>
              --> Capacity  = 4000
              --> Type      = [1]
              --> Load      = [1000]
              --> Orders    = (Order#9.P1 )
       <Tank#5>
              --> Capacity  = 5000
              --> Type      = [3]
              --> Load      = [5000]
              --> Orders    = (Order#1.P3 )

<Truck T#2>
       --> Orders = (Order#2 Order#11 )
       --> Tank      = (Tank#1 Tank#2 Tank#3 Tank#4 Tank#5 )
       <Tank#1>
              --> Capacity  = 1000
              --> Type      = [2]
              --> Load      = [500]
              --> Orders    = (Order#11.P2 )
       <Tank#2>
              --> Capacity  = 2000
              --> Type      = [3]
              --> Load      = [2000]
              --> Orders    = (Order#2.P3 )
       <Tank#3>
              --> Capacity  = 3000
              --> Type      = [2]
              --> Load      = [3000]
              --> Orders    = (Order#2.P2 )
       <Tank#4>
              --> Capacity  = 4000
```

```
                --> Type      = [5]
                --> Load      = [4000]
                --> Orders    = (Order#2.P5 )
        <Tank#5>
                --> Capacity  = 5000
                --> Type      = [3]
                --> Load      = [500]
                --> Orders    = (Order#11.P3 )

<Truck T#3>
        --> Orders = (Order#3 Order#5 Order#10 )
        --> Tank      = (Tank#1 Tank#2 Tank#3 Tank#4 Tank#5 )
        <Tank#1>
                --> Capacity  = 1000
                --> Type      = [3]
                --> Load      = [500]
                --> Orders    = (Order#5.P3 )
        <Tank#2>
                --> Capacity  = 2000
                --> Type      = [1]
                --> Load      = [2000]
                --> Orders    = (Order#3.P1 )
        <Tank#3>
                --> Capacity  = 3000
                --> Type      = [1]
                --> Load      = [2500]
                --> Orders    = (Order#5.P1 Order#10.P1 )
        <Tank#4>
                --> Capacity  = 4000
                --> Type      = [5]
                --> Load      = [4000]
                --> Orders    = (Order#3.P5 )
        <Tank#5>
                --> Capacity  = 5000
                --> Type      = [5]
                --> Load      = [2000]
                --> Orders    = (Order#5.P5 )

<Truck T#4>
        --> Orders = (Order#4 Order#6 Order#7 Order#8 )
        --> Tank      = (Tank#1 Tank#2 Tank#3 Tank#4 Tank#5 )
        <Tank#1>
                --> Capacity  = 1000
                --> Type      = [5]
                --> Load      = [1000]
                --> Orders    = (Order#4.P5 )
        <Tank#2>
                --> Capacity  = 2000
                --> Type      = [2]
                --> Load      = [1000]
                --> Orders    = (Order#8.P2 )
        <Tank#3>
                --> Capacity  = 3000
                --> Type      = [2]
                --> Load      = [3000]
                --> Orders    = (Order#4.P2 Order#6.P2 )
        <Tank#4>
                --> Capacity  = 4000
```

```
              --> Type      = [4]
              --> Load      = [2500]
              --> Orders    = (Order#4.P4 Order#6.P4 )
       <Tank#5>
              --> Capacity  = 5000
              --> Type      = [5]
              --> Load      = [4500]
              --> Orders    = (Order#6.P5 Order#7.P5 Order#8.P5 )

Number of fails              : 382
Number of choice points      : 403
Number of variables          : 466
Number of constraints        : 696
Reversible stack (bytes)     : 381924
Solver heap (bytes)          : 793372
Solver global heap (bytes)   : 4044
And stack (bytes)            : 4044
Or stack (bytes)             : 4044
Search Stack (bytes)         : 4044
Constraint queue (bytes)     : 26164
Total memory used (bytes)    : 1217636
Running time since creation  : 1.34
```

Part V

Local Search

This part consists of the following lessons:

- Chapter 23, *Basic Local Search*
- Chapter 24, *Writing a Neighborhood*
- Chapter 25, *Writing a Metaheuristic*
- Chapter 26, *Combining Complete and Local Search: Locating Warehouses*
- Chapter 27, *Minimizing Talent Wait Cost Using LNS*

C H A P T E R

23

Basic Local Search

In this lesson, you will learn how to:

- find a first solution
- use solution deltas
- use neighborhoods
- use metaheuristics

Solver includes classes and functions that perform local search. Local search methods are algorithms that take a solution to a problem, and attempt to improve it with a variety of usually small, local changes.

In general, algorithms of this kind are not guaranteed to find optimal solutions. However, they can be useful for finding good solutions quickly. This chapter describes Solver's local search features and tells how to get the most out of them.

What is local search?

Local search is a mechanism for searching for improvements to the solution of a problem by making small changes to the current solution. This concept is a familiar one in everyday life, and is the normal technique used, for instance, to add a new appointment to an already cluttered schedule. We don't recreate our entire schedule from scratch: we make small

changes so that the new appointment can be accommodated. This is the basic idea behind local search.

A two-stage approach

Solving a complete problem using a local search technique generally consists of two stages. In the first, a solution to the problem is found using an initial generation technique. In this stage, normal Solver search goals (such as `IloGenerate`) can be used to provide initial solutions.

Assuming that a first solution with a specific cost can be found, the second stage is to apply the local search procedure to improve this solution. In its basic form, the local search procedure is carried out by making a small change to the *current solution* to produce a *new solution.* This new solution is tested against the problem constraints for feasibility, and its cost is computed. If the new solution is feasible and has reduced cost, it is accepted as the current one, otherwise the current solution remains unchanged.

This process is repeated (using different solution changes on subsequent tries) until some stopping condition is met. The stopping condition can be based on time, number of changes attempted, or the fact that no changes to the solution can be found that will be accepted under the specified conditions.

The basic local search mechanism can be made more complicated, but such complications can be left until later. For now, we illustrate the process by using local search to solve a simple problem.

Solutions

In the description of local search above, we mentioned the concept of a *current solution*. IBM® ILOG® Concert Technology provides explicit support for representing solutions through the class `IloSolution`. Solver's local search support makes extensive use of `IloSolution` objects. The *IBM ILOG Solver Reference Manual* gives full details about the functions of `IloSolution` objects. Here, we will cover the parts most relevant to local search.

Using goals for local search

In Solver, a search goal such as `IloGenerate` solves the problem completely, and can find an optimum or a single solution satisfying the constraints, as desired. When we perform a local search using Solver, many goals (not just a single goal) are executed one after another. Each one of these goals makes a local move, changing the current solution to the new one.

A simple local search procedure

The combination of solutions and goals offers a simple way of implementing local search procedures. The basic process is to change the solution and then test it against the problem constraints. The algorithm is as follows:

1. Start with an initial solution *s*.
2. Store the cost of *s* as `currCost`.
3. Make a change to *s*.
4. If *s* violates constraints or has `cost >= currCost`, revert the change.
5. Go to step 2.

In this way, we only move to legal improving solutions. We reject solutions that do not improve the cost or that violate problem constraints. This is known as a *greedy improvement algorithm*, as it only makes changes to the solution that improve the cost.

In what follows, we will apply this technique to a modified version of the map coloring presented in Chapter 5, *Using Objectives: Map Coloring with Minimum Colors*.

Example: Map coloring

Consider the map coloring problem. In that problem, the map of Western Europe must be colored with only two colors. The constraints dictate that no two countries sharing a border can have the same color. However, in such a case, there is no solution. Therefore, the constraint on Luxembourg having a different color from its neighbors is relaxed, and a cost on its having the same color as a neighbor is added. In that example, the optimal assignment of colors is found using a complete search.

Here, as an introduction to using local search procedures, we replace the complete search for the minimum cost color assignment with a local search.

First of all, we list the definition of the model of the problem:

```
IloModel model(env);

IloIntVar Belgium(env, 0, 1), Denmark(env, 0, 1),
          France(env, 0, 1), Germany(env, 0, 1),
          Netherlands(env, 0, 1), Luxembourg(env, 0, 1);
IloIntVarArray AllVars(env);
AllVars.add(Belgium);
AllVars.add(Denmark);
AllVars.add(France);
AllVars.add(Germany);
AllVars.add(Netherlands);
AllVars.add(Luxembourg);
model.add(AllVars);
model.add(France != Belgium);
model.add(France != Germany);
model.add(Belgium != Netherlands);
model.add(Germany != Netherlands);
model.add(Germany != Denmark);
IloIntVar quality(env, 0, 20000);
model.add(quality == 257 * (France != Luxembourg)
                  + 9043 * (Luxembourg != Germany)
                  +  568 * (Luxembourg != Belgium));
IloObjective obj = IloMaximize(env, quality);
```

At this point, if we were using a complete search technique, we would add the objective to the model, create the solver from the model, and use `IloSolver::solve(IloGoal)` to find the optimal solution. However, here we will use a local search technique which proceeds in two phases. In the first phase, we generate an initial solution, and in the second, we improve it using local search.

Finding a first solution

To find a first solution, we create an instance of `IloSolver` from the model, create a goal which instantiates the color of each country, and then have the solver execute it:

```
IloSolver solver(model);
IloGoal generate = IloGenerate(env, AllVars);
if (!solver.solve(generate)) {
  throw "Could not find an initial solution";
}
```

We now store this initial solution in an `IloSolution` object which we create as follows:

```
IloSolution soln(env, "Colors");
```

We specify the variables whose values should be stored in the solution with the `add` member function:

```
soln.add(AllVars);
```

Since it is useful in a local search process to know the quality of current solution, we add the objective:

```
soln.add(obj);
```

This ensures that the value of the objective is stored together with the color of each country. Notice that we do not add the objective to the model, which would result in a search for the optimal solution when we call `solver.solve()`. We now store our first solution and display its quality:

```
soln.store(solver);
solver.out() << "1st solution: Objective value = "
             << soln.getObjectiveValue() << endl;
```

When storing a solution, the algorithm is passed as a parameter. The solution uses this algorithm to retrieve the current values of the variables added to the solution.

Improving the solution

After finding an initial solution, we next enter a local improvement phase. For that phase, we decide what changes to make to the solution to produce a new (changed) one.

In this example, there are only two available colors numbered zero (meaning blue) and one (meaning white). Thus, we can think of a simple change that can be made: change the color of one of the countries (from 0 to 1 or 1 to 0). This means that from any coloring, we can move to 6 other colorings (given our 6 countries). There are 6 possible changes that can be made. We therefore use an index to indicate which country should change color now:

```
IloInt curr = 0;
```

If we try to change the color of all 6 countries without success, we can stop the search. We are at what is termed a *local minimum*. This means that no change can improve the cost of the solution. It is important to recognize that a local minimum is often not the optimum.

We maintain a counter which marks off how many color changes have failed since the last success, and repeat until no move has worked for six tries:

```
IloInt failed = 0;
while (failed < 6) {
```

Inside the main loop, we change the value of `AllVars[curr]` in the solution from 0 to 1 (blue to white) or 1 to 0 (white to blue).

```
soln.setValue(AllVars[curr], 1 - soln.getValue(AllVars[curr]));
```

Now we are in a position to test the new solution for legality and to see if the cost is reduced over that of the current solution. If both of these conditions are satisfied, we save the resulting solution. This process is called making a *local move*. We construct a goal to perform these actions:

```
IloGoal move = IloAddConstraint(obj >= soln.getObjectiveValue() + 1)
            && IloRestoreSolution(env, soln)
            && IloStoreSolution(env, soln);
```

This goal is made up of three parts. First, we assert that the new objective must be at least one more than the cost of the current solution. This is because we are maximizing. Then, we restore the solution, which means that the values of the constrained variables added to `soln` are placed into the constrained variables of the model. (The variables in `AllVars` will be assigned their values stored in `soln`.) Note that the value of the objective is *not* restored when a solution is restored, only the values of variables are. Normally, the value of the objective will be completely determined by the variables on which it depends. However, if this is not the case, a *variable* representing the objective can be added to the solution, which will result in its value being restored with the solution. In the last line of the move goal, the solution is stored again. Although this does not change the values of the variables `AllVars` as stored in the solution, it does store the new value of the objective.

We execute the goal, changing the chosen country back to its previous color if the color change was not successful in increasing the objective or if the color change violated any problem constraints.

```
if (solver.solve(move)) {
  solver.out() << "Move made:    Objective value = "
               << soln.getObjectiveValue() << endl;
  failed = 0;
}
else {
  soln.setValue(AllVars[curr], 1 - soln.getValue(AllVars[curr]));
  failed++;
}
```

We then move to the next country to start the next loop:

```
  curr = (curr + 1) % 6;
}
```

After the local search loop terminates (when no country's color could be changed to increase the objective), the solution found can be presented via:

```
solver.solve(IloRestoreSolution(env, soln));
print(solver, "Belgium:     ", Belgium);
print(solver, "Denmark:     ", Denmark);
print(solver, "France:      ", France);
print(solver, "Germany:     ", Germany);
print(solver, "Netherlands: ", Netherlands);
print(solver, "Luxembourg:  ", Luxembourg);
```

In this small example, the solution delivered by local search is the optimal solution. One move is made by the local search procedure: changing the color of Luxembourg.

Making it easier: Using solution deltas

Solutions can represent not only a complete assignment of decision variables, but also a partial assignment. This allows a solution to represent the *change* to another solution. When a solution is used to represent the change to another solution, it is called a *solution delta*.

We can use solution deltas to perform local search for our map coloring problem. Solution deltas make it unnecessary to manipulate the solution directly—instead, the change can be specified in another solution. Predefined Solver goals allow us to examine the application of such a change to a solution.

For instance, consider the following fragment of code which creates four zero-one (0-1) variables and a solution. The variables are added to the solution and each variable is given a stored value of 0 within the solution.

```
IloEnv env;
IloIntVarArray vars(env, 4, 0, 1);
IloModel model(env);
model.add(vars);
IloSolution soln(env);
soln.add(vars);
IloInt i;
for (i = 0; i < vars.getSize(); i++)
  soln.setValue(vars[i], 0);
```

Suppose we want to make a change to our all-zero solution. We consider as possible changes (or moves), the setting of each variable to 1.

Solver provides a goal, `IloScanDeltas`, for investigating a set of changes to a solution. However, we first need to define the set of changes. This is done in an array as follows:

```
IloSolutionArray deltas(env, vars.getSize());
for (i = 0; i < vars.getSize(); i++) {
  deltas[i] = IloSolution(env);
  deltas[i].add(vars[i]);
  deltas[i].setValue(vars[i], 1);
}
```

The code above creates an array of deltas (one for each original 0-1 variable), and places the desired change (to set the value to 1) to each variable within the delta. We are now in a position to examine the effect each delta will have on the solution. We write:

```
IloGoal scan = IloScanDeltas(env, soln, deltas);
IloSolver solver(model);
solver.startNewSearch(scan);
while (solver.next()) {
  for (i = 0; i < vars.getSize(); i++)
    solver.out() << solver.getValue(vars[i]);
  solver.out() << endl;
}
```

This code produces the following output:

```
1000
0100
0010
0001
```

The `IloScanDeltas` goal instantiates variables according to the application of each delta to the solution. Note that the solution `soln` has *not* been changed by this goal.

`IloScanDeltas` can be seen as a way of exploring all the moves that can be made from a solution. Often this set of moves is termed the *neighborhood* of the solution. A single element of the neighborhood is called a *neighbor.*

Using solution deltas in the map coloring problem

We can modify our map coloring problem to use solution deltas, as opposed to modifying the solution directly. We address only the local search part of the code that appears after the creation of the solution object and the generation of the initial solution.

From any current solution, we know that there are 6 possible changes that can be made, one corresponding to each country. To represent this, we create an array of solutions to act as solution deltas:

```
IloSolutionArray deltas(env, 6);
```

The deltas are then created:

```
for (IloInt i = 0; i < 6; i++) {
  deltas[i] = IloSolution(env);
  deltas[i].add(AllVars[i]);
  deltas[i].setValue(AllVars[i], 1 - soln.getValue(AllVars[i]));
}
```

The variable `deltas[i]` represents a change of color for country `i`. Each delta contains a variable whose saved value is a different color from that stored in the current solution.

We define the goal to attempt a move in stages. In the first stage, a goal explores the possible changes that can be applied to the solution (that is, explores the neighborhood):

```
IloNeighborIdentifier nid(env);
IloGoal scan = IloScanDeltas(env, soln, deltas, nid);
```

`IloScanDeltas` takes a solution and an array of solution deltas, and applies each delta to the solution in turn. For each delta to be examined, each one that can be applied legally results in a leaf node of `IloScanDeltas`. The variable `nid` is a *neighbor identifier.* This class is used by `IloScanDeltas` to communicate information back to the user about the solution delta representing the current variable instantiation. Both the index of the delta and the delta itself can be accessed from the solver using `solver.getAtIndex(nid)` and `solver.getAtDelta(nid)`.

Next, one neighbor is chosen from those available. A search selector is used which selects the first legal neighbor found:

```
IloGoal selectNeighbor = IloSelectSearch(env, scan, IloFirstSolution(env));
```

Finally, we construct the goal that moves to the chosen neighbor, as follows:

```
IloGoal move = IloAddConstraint(obj >= soln.getObjectiveValue() + 1)
            && selectNeighbor
            && IloStoreSolution(env, soln);
```

The first term of the composed goal states that the objective value must be increased from its current value. The second term examines all deltas and selects a single change to be made, if a legal one exists. At this point, the decision variables `AllVars` are instantiated with the new solution. The third element stores the new solution. Now the local search optimization loop

is performed. The loop terminates when no moves can be taken. This happens when IloScanDeltas fails because none of the available deltas can legally be applied.

```
IloIIM iim(solver);
while (solver.solve(move)) {
  solver.out() << "Move made:    Objective value = "
                 << soln.getObjectiveValue() << endl;
  IloInt atIndex = iim.getAtIndex(nid);
  IloIntVar var = AllVars[atIndex];
  deltas[atIndex].setValue(var, 1 - deltas[atIndex].getValue(var));
}
```

Whenever the constrained variables of the problem are instantiated by IloScanDeltas, IloScanDeltas assures that the neighbor identifier nid contains information on the delta itself. We get the index of this delta using solver.getAtIndex(nid). We then flip the value of the variable stored in deltas[atIndex], which maintains it in the opposite sense from what is stored in the current solution. This ensures that future moves can change the value of the relevant variable once more.

When the optimization loop is over, the solution found is presented:

```
solver.solve(IloRestoreSolution(env, soln));
```

Functionally, this search is identical to the previous version which directly manipulates the solution. However, the use of IloScanDeltas is preferred to testing each delta in turn via IloRestoreSolution as it is much more efficient when large numbers of deltas are being tested.

Using neighborhoods

Using an array of deltas to specify the possible neighbors of a solution is one step from directly manipulating the solution and restoring it, but maintaining the set of deltas can be laborious. Solver provides methods for describing neighborhoods in a more structured way. Using these neighborhoods makes it easier to define the neighbors of a solution. There are various predefined neighborhoods in Solver, and you can also define your own.

Just as IloScanDeltas was used to apply each delta to a solution, a goal IloScanNHood is used to apply each neighboring solution to the current one. We can use this goal in a further refinement to our map coloring example.

Instead of creating an array of deltas to represent the neighbors of a solution, we use the class IloNHood, a special class for defining neighborhoods. The predefined neighborhood

`IloFlip` produces changes to 0-1 variables, which is exactly the type of neighborhood we require for our map coloring problem. We use that neighborhood here:

```
IloNHood nhood = IloFlip(env, AllVars);
```

The goal that explores the neighborhood and selects the neighbor to be taken is defined as:

```
IloNeighborIdentifier nid(env);
IloGoal scan = IloScanNHood(env, nhood, nid, soln);
IloGoal selectNeighbor = IloSelectSearch(env, scan, IloFirstSolution(env));
```

We use `IloFirstSolution` as we are interested in any legal neighbor. The `nid` parameter passed to `IloScanNHood` will be used for storing both the neighborhood index (numbered from 0 upwards) and the delta corresponding to any instantiated neighbor. Note that while for `IloScanDeltas` the index has a very obvious meaning (the index of the delta in the array specified), for `IloScanNHood` the index only has meaning to the neighborhood itself. This notion will become clearer later.

Then, the goal which attempts a change is defined as follows:

```
IloGoal move = IloAddConstraint(obj >= soln.getObjectiveValue() + 1)
            && selectNeighbor
            && IloNotify(env, nhood, nid)
            && IloStoreSolution(env, soln);
```

The goal is composed for four terms. In the first term the lower bound on the cost is set. In the next, the neighborhood is scanned and a neighbor selected. The third term uses the predefined Solver goal `IloNotify`. This goal takes as parameters a neighborhood and a neighborhood identifier. It indicates to the neighborhood, through the neighborhood identifier, the index of the neighbor that is about to be taken. (Such notification is important for neighborhoods which may change their behavior on subsequent scans depending on which moves have been taken. Full details on this are given in the section “Writing your own neighborhood” later in this chapter.) Finally, the fourth term stores the solution.

> ***Note:*** *The order of the last two terms of the goal is important. A neighborhood must be notified* **before** *the new solution is saved. This allows the neighborhood to compute the difference between the new solution and the old solution, if it has such a need.*

Finally, we use the following optimization loop:

```
while (solver.solve(move)) {
  solver.out() << "Move made:    Objective value = "
               << soln.getObjectiveValue() << endl;
}
```

The remainder of the code is the same as before. The result is code that is functionally equivalent to the code using `IloScanDeltas`, although its complexity is reduced. The efficiency of the code is also approximately the same.

Metaheuristics

In the map coloring example we examined in the previous sections, all our local searches were of the greedy variety. That is, they made only moves which improved the objective. This greedy technique is efficient, but can encounter difficulties even on small problems. The difficulty is that although there may be no improving moves from the current solution, this does not mean that better solutions do not exist. Moreover, there is no bound upon how much better a solution could be.

For example, consider the problem:

```
IloEnv env;
IloModel mdl(env);
IloIntVarArray a(env, 2, 0, 1);
IloInt C = 1;
IloNumVar cost(env, 0, 2 * C + 1);
mdl.add(cost == C * (a[0] + a[1]) + (C + 1.0) * (a[0] != a[1]));
```

The costs of different configurations are given below:

Table 23.1 *Example Configuration Costs*

a[0]	a[1]	cost
0	0	0
0	1	2C + 1
1	0	2C + 1
1	1	2C

If the current solution happens to be at state `a[0]=1, a[1]=1` (which we shall term `11` by omitting reference to the array `a`), and we are making moves by flipping single bits, then there is no move that will improve the cost. Moves to state `01` and `10` increase the cost by one. However, the state `00` has cost which is `2C` less than that of state `11`. Obviously, we can make this disparity arbitrarily large by increasing `C`. The configurations `01` and `10` act as a barrier to local search “guarding” the global minimum.

The well-known problem just described is the local minimum problem: greedy searches become trapped in these local minima, and thus the quality of solutions obtained can suffer. Because of this problem, metaheuristics are commonly used to *escape* from these local minima. A variety of escape techniques can be used, the simplest of which is to allow moves

which *degrade* the cost of the solution, in the hope that subsequent moves that improve the cost can be found.

Care should be taken, however, about the way in which we allow the cost to degrade. To simply accept any move without guidance leads to a *random walk* through the neighborhood space, and is unlikely to bring about a good solution. Instead metaheuristics use controlled methods of acceptance of degrading solutions, in order to both escape local minima and then go on to find better solutions.

Example: Using metaheuristics

Consider the example already described, where the solution `11` is a *local minimum* (when a single value flip is the move used), and `00` is the *global minimum*. Solver provides support for metaheuristics through the `IloMetaHeuristic` class. We can use a metaheuristic provided in Solver to get around the problem that none of the two available moves reduce the cost. To do this, we allow cost-degrading moves to be accepted. A metaheuristic that allows such moves to be accepted is simulated annealing.

We use simulated annealing in our small example to attempt to move the search from a start state of `11` to a final state of `00`.

Simulated annealing

Simulated annealing operates by allowing cost degrading moves based on a probabilistic rule. For the purposes of this description, we assume the objective is to be minimized, although both minimization and maximization can be performed within Solver. Given that the current solution has cost c, and the proposed solution has cost $c + d$ (d can be positive or negative), simulated annealing accepts the new solution with probability:

1 if $d <= 0$ {Non-degrading move}

$exp(-d / T)$ if $d > 0$ {Degrading move}

Notice first that non-degrading moves are always accepted. Second, degrading moves are accepted with a probability which increases with a decrease in cost difference d, and a decrease in a parameter T. Ignoring T for the moment, we see that if a particular move degrades the cost by a large amount, it is less likely to be accepted than one which degrades the cost by a small amount.

The parameter T is known as the *temperature* and it controls the way the search operates. The effect of T is that for large T, cost degrading moves have a high chance of acceptance. For low T, the opposite situation holds, with only moves that slightly degrade the cost having any real chance of acceptance. The idea is to begin the search with T high, and gradually lower T as search progresses. This results in the search being able to *roam* freely near the

beginning; by contrast, later on, when the temperature is low, the search looks more like greedy search, making only non-degrading moves.

It is worth noting that in simulated annealing, the moves to be examined must be drawn from the neighborhood in a random fashion. Otherwise, the moves specified first in the neighborhood will have a larger chance of acceptance than those that occur further on in the neighborhood which is contrary to the ethos of simulated annealing. A method for achieving this will be described later.

Controlling the simulated annealing search

Two things control how a simulated annealing search operates. The first is how the temperature is reduced as the search progresses. A common method, used in the predefined simulated annealing metaheuristic in Solver, is to reduce temperature after each move according to the rule:

```
T := T * factor
```

where factor is in the range *[0..1)*, with values close to unity being preferred, as this reduces the temperature more slowly. As long as the temperature is allowed to reduce enough so that by the end of search the *roaming* behavior has ceased, a slower reduction of the temperature tends to result in better solutions. Of course, such a search also takes longer to run.

The second factor which controls the search is the initial temperature. If the initial temperature is set too high, search will roam for a very long time before settling. Thus it could take a very long time to produce a good solution as degrading moves are accepted too readily for too long. If the initial temperature is set too low, local minima may trap the search too quickly. Rearranging the probability acceptance equation, we have:

$$T = -d / ln(P(accept))$$

So, if we wish a cost degradation of d to be accepted with probability $P(accept)$, at the beginning of search, then the initial value of T should be set to $-d / ln(P(accept))$. Normally, metaheuristics require some tuning for the problem being solved. These parameters for simulated annealing are examples of what to tune.

Building the model

Going back now to our small example, we are ready to solve it using simulated annealing. The code for the model and a solution, are described as follows:

```
IloEnv env;
IloModel mdl(env);
IloIntVarArray a(env, 2, 0, 1);
IloInt C = 1;
IloIntVar cost(env, 0, 2 * C + 1);
mdl.add(cost == C * (a[0] + a[1]) + (C + 1.0) * (a[0] != a[1]));
IloSolution soln(env);
soln.add(a);
soln.add(IloMinimize(env, cost));
```

Our neighborhood is one which flips a single bit of the solution, and so we can use `IloFlip` for this. However, what we eventually want is a *random* neighbor which is termed legal by the simulated annealing rule to be accepted. There is a special neighborhood modifier defined in Solver, `IloRandomize`, which, given a neighborhood, jumbles the order in a random fashion. This jumbling takes place for each neighborhood move made. We specify our neighborhood:

```
IloRandom rand(env);
IloNHood nh = IloRandomize(env, IloFlip(env, a), rand);
```

`IloRandom` is a Concert Technology class for generating pseudo-random numbers. It is passed to the `IloRandomize` neighborhood which then uses `rand` to generate the randomized neighborhood.

Creating the first solution

Now, we create the first solution (`11`):

```
IloSolver solver(mdl);
IloGoal store = IloStoreSolution(env, soln);
IloGoal firstSol = IloAddConstraint(a[0] == 1)
                && IloAddConstraint(a[1] == 1)
                && store;
solver.solve(firstSol);
```

Creating the metaheuristic object

We create an metaheuristic object which performs the simulated annealing filtering rule.

```
IloMetaHeuristic simAnn = IloSimulatedAnnealing(env, rand, 1.0, 0.99);
```

The parameters passed are, a source of random numbers (so that probabilistic tests can be carried out), the initial temperature (`1.0` in this case), and the reduction rate (the temperature is multiplied by `0.99` after each move). We use the same source of random numbers as used by the neighborhood, but we could have used a different source if we had desired (another instance of `IloRandom`). There is no need to pass a cost variable or objective to the metaheuristic. The one added to the solution will be used. This is true of all of Solver's built in metaheuristics.

Scanning the neighborhood

We then create a goal to scan the neighborhood. Information on the neighbor being scanned is stored in `nid`.

```
IloNeighborIdentifier nid(env);
IloGoal scan = IloScanNHood(env, nh, nid, soln);
```

The simulated annealing metaheuristic, like others supplied with Solver, operates by filtering the possible allowable neighbors generated by a goal that explores a neighborhood. For instance, the simulated annealing metaheuristic filters all neighbors for which $r > exp(-d/T)$, where r is a random number in $[0..1)$ and d is the degradation in cost of the proposed neighbor (both minimization and maximization are supported). A function, `IloApplyMetaHeuristic`, takes a neighborhood exploration goal and applies the filtering. We write:

```
IloGoal scanSA = IloApplyMetaHeuristic(env, soln, simAnn, scan);
```

Making a single move

Finally, we can construct a goal to make a single move via:

```
IloGoal scanSA = IloApplyMetaHeuristic(env, soln, simAnn, scan);
IloGoal choose = IloSelectSearch(env, scanSA, IloFirstSolution(env));
IloGoal notify = IloNotify(env, nh, nid) && IloNotify(env, simAnn, nid);
IloGoal move = choose && notify && store;
```

The goal to make a move is made up of three phases: a choice stage, a notification stage, and a commitment stage. These are represented by the goals `choose`, `notify`, and `store`. The `choose` goal explores the neighborhood, with the simulated annealing metaheuristic filter applied, and then chooses the first legal neighbor. The `notify` goal notifies the neighborhood and the metaheuristic that a move is about to be taken. (Notice that now we also notify the metaheuristic as well as the neighborhood.) Finally, the new solution is stored by the `store` goal. It is important that the neighborhood and the metaheuristic are notified *before* the new solution is stored. For the simulated annealing metaheuristic, the temperature is reduced by the reduction factor when the metaheuristic is notified that a move is being taken.

Searching for a solution

The search can now progress. First, we create a solution to keep the best solution found by simulated annealing over the run. As cost degrading moves can be taken, the final solution may not be the same as the best one found.

```
IloSolution best = soln.makeClone(env);
```

Now we enter an optimization loop, where we perform up to 100 iterations:

```
IloInt iter = 100;
do {
  while (--iter > 0 && solver.solve(move)) {
    if (soln.getObjectiveValue() < best.getObjectiveValue())
      best.copy(soln);
  }
} while (iter > 0 && !simAnn.complete());
```

The inner `while` loop can exit under two conditions: the first is that the iteration limit has been exhausted. In this case, the outer loop also terminates immediately. The inner loop can also terminate when `solver.solve(move)` returns `IloFalse`. This is the case when the move goal fails, that is, when all moves from the neighborhood are rejected and none can be chosen. When this happens for a metaheuristic, Solver provides an API for offering the metaheuristic the chance to change its filtering so that moves can be re-taken in the future. The `complete()` method of the metaheuristics performs this action. If `IloFalse` is returned, it means that the metaheuristic wishes to continue working, whereas if it returns `IloTrue`, it indicates that there is no way for it to relax its filtering, and so it would like to terminate the search.

For the simulated annealing metaheuristic, the temperature is *divided* by the ratio specified when the metaheuristic was constructed (`0.99` here), slightly increasing the temperature. This is done in the hope that an increased temperature can allow search to progress more easily. The return value of `complete()` is `IloFalse` for the simulated annealing metaheuristic, indicating that it never wants to stop search: it is up to the user to impose a limit.

After the optimization loop has terminated, we can restore the best solution found:

```
solver.solve(IloRestoreSolution(env, best));
```

Other metaheuristics

Other metaheuristics are provided by Solver. A brief overview of these is given in the following sections.

IloImprove

This metaheuristic is not really a metaheuristic in the usual sense. However, it fits well into the metaheuristic framework, and so is considered as one. It implements a greedy search where all cost degrading moves are rejected.

IloTabuSearch

Tabu search is a popular metaheuristic. The idea of tabu search in its simplest form is to first, make the lowest cost move at each stage in the search, and, second, to accept cost degrading moves. This means that in a local minimum, tabu search will move out, since it can accept cost degrading moves. However, unless prevented, it could immediately *fall back* to the local minimum it left by making the *opposite* if such an opposite move exists (as is common) and such a move is now the best one that can be made (which is also common).

To counteract this *cycling effect*, tabu search employs various mechanisms, the typical one, and that implemented in Solver, being the *tabu list*. This list holds information on the solutions recently visited and forbids the search from moving back to a solution which is the same as (or shares some features or attributes with) one it has recently visited. The length of the list is usually important to the performance of the metaheuristic. Typically this metaheuristic performs better than simulated annealing.

Constructing moves

Sometimes the construction of a goal which makes a single move can be a little laborious. To that end, Solver provides a way of creating such *standard* goals via the function `IloSingleMove`. One of the synopses of `IloSingleMove` is as follows:

```
IloGoal IloSingleMove(IloEnv env,
                      IloSolution solution,
                      IloNHood nhood,
                      IloMetaHeuristic mh,
                      IloSearchSelector sel);
```

The parameters comprise the elements needed to build up a goal that makes a single move. We supply a solution, a neighborhood, a metaheuristic, and a search selector that selects a move from the neighborhood. The generated goal automatically generates an `IloScanNHood` goal and automatically notifies the neighborhood and metaheuristic that a move is about to be taken. `IloSingleMove` also stores the current solution before succeeding.

In our simulated annealing example, given we had defined the solution `soln`, we could have replaced:

```
IloNeighborIdentifier nid(env);
IloGoal scan = IloScanNHood(env, nh, nid, soln);
IloGoal scanSA = IloApplyMetaHeuristic(env, soln, simAnn, scan);
IloGoal choose = IloSelectSearch(env, scanSA, IloFirstSolution(env));
IloGoal notify = IloNotify(env, nh, nid) && IloNotify(env, simAnn, nid);
IloGoal move = choose && notify && store;
```

with:

```
IloGoal move = IloSingleMove(env, soln, nh, simAnn, IloFirstSolution(env));
```

The use of `IloSingleMove` usually makes the use of `IloScanNHood`, `IloScanDeltas`, `IloNotify`, and `IloApplyMetaHeuristic` unnecessary, unless more complex goals need to be built which require the use of these lower level functions. Therefore, you should try to write your local search algorithms using `IloSingleMove`, using the other functions where `IloSingleMove` does not suffice.

The map coloring example revisited

The map coloring example can also be greatly simplified by the use of `IloSingleMove` and `IloImprove`. The local search part of the code can now be written as follows:

```
IloNHood nhood = IloFlip(env, AllVars);
IloGoal move = IloSingleMove(env, soln, nhood, IloImprove(env));
while (solver.solve(move)) {
  solver.out() << "Move made:    Objective value = "
               << soln.getObjectiveValue() << endl;
}
```

For reference, the entire example using `IloSingleMove` is given below:

```
#include <ilsolver/iimls.h>

ILOSTLBEGIN

typedef enum { COLOR_DIRECT, COLOR_DELTAS,
               COLOR_NHOOD, COLOR_SINGLEMOVE } ColorType;

const char* Names[] = {"blue", "white"};

void print(IloSolver solver, const char* name, IloIntVar var) {
  solver.out() << name << Names[(IloInt)solver.getValue(var)] << endl;
}

int main() {
  IloEnv env;
  try {
    IloModel model(env);
```

```
    IloIntVar Belgium(env, 0, 1), Denmark(env, 0, 1),
              France(env, 0, 1), Germany(env, 0, 1),
              Netherlands(env, 0, 1), Luxembourg(env, 0, 1);
    IloIntVarArray AllVars(env);
    AllVars.add(Belgium);
    AllVars.add(Denmark);
    AllVars.add(France);
    AllVars.add(Germany);
    AllVars.add(Netherlands);
    AllVars.add(Luxembourg);
    model.add(AllVars);
    model.add(France != Belgium);
    model.add(France != Germany);
    model.add(Belgium != Netherlands);
    model.add(Germany != Netherlands);
    model.add(Germany != Denmark);
    IloIntVar quality(env, 0, 20000);
    model.add(quality == 257 * (France != Luxembourg)
                      + 9043 * (Luxembourg != Germany)
                      +  568 * (Luxembourg != Belgium));
    IloObjective obj = IloMaximize(env, quality);
    IloSolver solver(model);
    IloGoal generate = IloGenerate(env, AllVars);
    if (!solver.solve(generate)) {
      throw "Could not find an initial solution";
    }
    IloSolution soln(env, "Colors");
    soln.add(AllVars);
    soln.add(obj);
    soln.store(solver);
    solver.out() << "1st solution: Objective value = "
                 << soln.getObjectiveValue() << endl;
    IloNHood nhood = IloFlip(env, AllVars);
    IloGoal move = IloSingleMove(env, soln, nhood, IloImprove(env));
    while (solver.solve(move)) {
      solver.out() << "Move made:    Objective value = "
                   << soln.getObjectiveValue() << endl;
    }
    solver.solve(IloRestoreSolution(env, soln));
    print(solver, "Belgium:     ", Belgium);
    print(solver, "Denmark:     ", Denmark);
    print(solver, "France:      ", France);
    print(solver, "Germany:     ", Germany);
    print(solver, "Netherlands: ", Netherlands);
    print(solver, "Luxembourg:  ", Luxembourg);
  }
  catch(IloException &ex) {
    cout << "Caught : " << ex << endl;
  }
  catch(const char *ex) {
    cout << "Caught : " << ex << endl;
  }
  env.end();
  return 0;
}
```

More about neighborhoods

We have been introduced to one basic neighborhood (`IloFlip`) and one neighborhood modifier (`IloRandomize`). Other basic neighborhood and modifiers exist which are useful for solving problems using local search. A few of the more useful ones are summarized in the following sections.

Basic neighborhoods

The following code generates a neighborhood which has a number of neighbors equal to the size of `vars`. Each neighbor sets one element of `vars` to the value `value`.

```
IloNHood IloSetToValue(IloEnv env,
                       IloNumVarArray vars,
                       IloInt value,
                       const char *name = 0);
```

The following code generates a neighborhood which has a number of neighbors equal to the size of `vars` times `max - min + 1`. The neighbors try to assign each variable an integer value ranging from `min` to `max`.

```
IloNHood IloChangeValue(IloEnv env,
                        IloNumVarArray vars,
                        IloInt min,
                        IloInt max,
                        const char *name = 0);
```

The following code generates a neighborhood which swaps the values of all pairs of variables in `vars`. Given that n is the size of the array `vars`, the size of this neighborhood is $n * (n - 1) / 2$.

```
IloNHood IloSwap(IloEnv env,
                 IloNumVarArray vars,
                 const char *name = 0);
```

Neighborhood modifiers

The following code generates a concatenation of neighborhoods. The set of neighbors of the new neighborhood is the union of the neighbors of `nhoods` or `nhood1` and `nhood2`, depending on the signature used.

```
IloNHood IloConcatenate(IloEnv env,
                        IloNHoodArray nhoods,
                        const char *name = 0);
IloNHood operator + (IloNHood nhood1, IloNHood nhood2);
```

The following code generates a neighborhood which behaves exactly like `nhood`, except that the generated neighborhood adjusts the indices passed to `nhood`. The adjustment is based on the previously accepted move. If the index (as passed to `nhood`) of the previous move

accepted was *k*, then the indices passed to `nhood` will begin at *k + offset + 1* modulo the size of the neighborhood for the next move.

```
IloNHood IloContinue(IloEnv env,
                     IloNHood nhood,
                     IloInt offset = 0,
                     const char *name = 0);
```

CHAPTER

24

Writing a Neighborhood

In this lesson, you will learn how to:

- write a neighborhood
- use the class `IloNHoodI`
- write a neighborhood modifier

Writing a new neighborhood

It is possible to write your own local search neighborhoods using Solver by subclassing the `IloNHoodI` class. This section shows you how to do this. The basic behavior of the neighborhood class is to generate possible moves from a *current solution* for local search procedures. Moves are described by the difference between the current solution and the new one. This difference is installed in a mini-solution or *delta* that contains just the variables that change and their new values.

Why would you want to write your own neighborhood? Solver provides certain standard neighborhoods which can be used for many common problems. However, challenging aspects of a problem may require some specialization, or integration of domain knowledge into the search procedure. One way of doing this is to write a neighborhood that generates neighbors dedicated to the problem at hand.

Here, we describe a small example that uses local search to sort a sequence of numbers, using a “Bubble Sort” type neighborhood. It should be noted that this is not an efficient way of sorting: the example is used here merely for its illustrative value.

Model

The model of the sorting problem is as follows. Assume *array* is the array of numbers to be sorted. The sorted array is represented by a permutation of the original array. Given *N* numbers to sort, a permutation array *perm* of size *N* is set up with each variable in this array having domain *0..N-1*. So, for instance, element *i* in the permuted array corresponds to the original array element *perm*[*i*]. The goal is to produce a *perm* array where, for all *i* in *0..n-2*: *array*[*perm*[*i*]] <= *array*[*perm*[*i + 1*]], that is, the ordered array is non-decreasing.

To use local search to find the sorted array, we need an appropriate cost function. A cost function which results in a simple search space associates a cost of 0 or 1 with each pair of values in the array to be sorted. The cost is 0 if they are in the correct order (lower before higher, or both values equal), and 1 if they are not. By reducing this cost to zero, the array can be sorted. This cost function also has the other desirable feature that when any pair of elements are swapped from an incorrect order to a correct order, the cost is reduced. This allows us to use a simple greedy improvement scheme to find the minimum.

The IloNHoodI class

We shall define a neighborhood specifically tailored to sorting. The idea of this neighborhood is to swap elements of the array which are adjacent, as in the classic “Bubble Sort” algorithm. We shall also include some other features which demonstrate how more complex behavior can be achieved.

Features of the “Bubble Sort” neighborhood

Our “Bubble Sort” neighborhood has three main features:

- It generates swaps of adjacent members of the permutation array. If the permutation array is of size *n*, then the neighborhood will be of size *n - 2*. Neighbor number *i* will swap elements *i* and *i + 1*. This is the minimal requirement for sorting the set of numbers. However, we also add some other features which are useful demonstrations of techniques which can be used when you write your own neighborhoods.
- When a swap is made, we do not start the next move by exploring swaps beginning with elements 0 and 1, but continue from where we left off. This is similar to the behavior of `IloContinue` and places priority on looking at unexamined swaps.
- When calculating the delta, the “Bubble Sort” neighborhood also calculates the difference in “sorting cost.” We use this difference, together with the previous cost, to compute the new cost of the neighbor. If the cost variable is added to the delta with a new value as computed, this speeds up local search, as deltas can be more readily rejected if they do not meet the required cost criteria. Note that such a technique needs to have

domain knowledge of the cost function, but can accelerate search. Although we set the *value* of the new cost in the delta, in general all that is needed is an upper or lower bound, depending on whether one is minimizing or maximizing.

Virtual functions of IloNHoodI

The virtual functions of `IloNHoodI`, with the exception of `reset`, are all called from the `IloScanNHood` goal (or via `IloScanNHood` by `IloSingleMove`) and are passed an instance of `IloSolver`. This instance of `IloSolver` is the one executing `IloScanNHood`.

To define a neighborhood, the following virtual functions are available:

```
virtual void start(IloSolver solver, IloSolution soln)
```

The `IloNHoodI::start` method is called to signify to the neighborhood that neighbors will be requested of it, the root of these neighbors being the solution `soln`.

For our “Bubble Sort” neighborhood, we need to store a reference to the solution so that we can generate neighbors with it as a reference.

```
virtual IloInt getSize(IloSolver solver) const = 0
```

The `IloNHoodI::getSize` method asks the neighborhood to return the number of neighbors of which it is comprised. The neighborhood should be prepared to define neighbors with indices from 0 to $s-1$ where s is the value returned.

Since our “Bubble Sort” neighborhood exchanges only two adjacent cells, its size will be $n-1$ where n is the size of the array to be sorted.

```
virtual IloSolution define(IloSolver solver, IloInt i) = 0
```

The `IloNHoodI::define` method is called to retrieve a particular neighbor of the neighborhood. The index `i` represents the index of the neighbor being requested, where neighbors are indexed from 0 upwards.

For our “Bubble Sort” neighborhood the definition of a neighbor mentions two adjacent variables of the permutation array, with values swapped from their values in the previous solution. As mentioned previously, we also add cost bounds to increase efficiency.

```
virtual void notify(IloSolver solver, IloInt i)
```

When a neighborhood move is about to be taken, the `IloNHoodI::notify()` method is called. This function supplies the index `i` to tell the neighborhood which neighbor is accepted.

> ***Note:** A neighborhood may not receive a notify call, even if a move is being taken, as the successful move may have been defined by a different neighborhood. This is the case, for example, when neighborhoods are added via the + operator.*

For our “Bubble Sort” neighborhood, we can use the `IloNHoodI::notify` member function to identify the previous move made, and use it in the definition of subsequent moves such that a different part of the neighborhood is explored. It is also an ideal place to capture the cost of the current instantiation so that we can compute cost differences from it in subsequent moves.

```
virtual void notifyOther(IloSolver solver, IloSolution delta)
```

When a move is about to be taken, but this neighborhood did not generate it, `IloNHoodI::notifyOther` is called on the neighborhood. This call is generated from `IloNHoodI::notify()` method of the `IloConcatenate` neighborhood (same as operator `+`) for the neighborhoods that did not define the move being taken. `delta` is the solution delta of the move.

Here, as in `notify`, we can capture the cost of the current instantiation so that we can compute cost differences from it in subsequent moves. In the context of the example, where the “Bubble Sort” neighborhood is the only one, `notifyOther` is never called, as no other neighborhood could perform the move. However, defining `notifyOther` here is good design practice, so the neighborhood could be used in a different context later.

```
virtual void reset();
```

This method can be called by the user (indirectly through the handle class `IloNHood`) to reset any internal state of the neighborhood to that at its creation.

For the “Bubble Sort” neighborhood, two items of internal state need to be reset: the cost of the previous solution, and the index of the last neighbor accepted.

Defining the “Bubble Sort” neighborhood

The following function displays the permuted array and its cost. The values of `perm` are retrieved from the algorithm via `solver.getValue`.

```
void Display(IloSolver solver,
             IloIntVarArray perm,
             IloIntArray values,
             IloIntVar cost) {
  cout << solver.getIntVarArray(perm) << endl;
  cout << solver.getIntVar(cost) << endl;
  for (IloInt i = 0; i < values.getSize(); i++) {
    cout << values[solver.getValue(perm[i])] << " ";
  }
  cout << endl;
}
```

The class BubbleI

Here is the synopsis of the class `BubbleI`, which is subclassed from `IloNHoodI`:

```
class BubbleI : public IloNHoodI {
private:
  IloSolution    _soln;
  IloIntVarArray _perm;
  IloIntArray    _values;
  IloIntVar      _cost;
  IloInt         _costVal;
  IloInt         _startPoint;
public:
  BubbleI(IloEnv env,
          IloIntVarArray perm,
          IloIntArray values,
          IloIntVar cost,
          const char *name = 0);
  void start(IloSolver solver, IloSolution soln);
  IloSolution define(IloSolver solver, IloInt i);
  IloInt getSize(IloSolver solver) const;
  void notify(IloSolver solver, IloInt i);
  void notifyOther(IloSolver solver, IloSolution delta);
  void reset();
};
```

The class BubbleI—local variables

The `BubbleI` class is subclassed from `IloNHoodI` and implements our “Bubble Sort” neighborhood. The local variables have the following functions:

`_soln`

This is the solution passed in the last call to `start()`.

`_perm`

This is the permutation array which holds the decision variables of the local search process. These are the variables which will be mentioned in the neighbors generated.

`_values`

This is the original array of values to sort.

`_cost`

This is the cost variable of the sorting problem. We pass this here so that we can specify cost bounds in generated deltas. This is better than taking the objective variable from `_soln` as the local search could be optimizing something different from just the cost of the sorting problem. For example, it could be optimizing a problem where cost is only a part.

`_costVal`

This is the value of the cost variable the last time the solution was saved. A sentinel value of `IloIntMax` means that no move has been made as yet. No cost bounds will be imposed in the first iteration of the search. This is not usually a significant performance problem.

`_startPoint`

This is the index in the `_perm` array that the neighbor indexed zero corresponds to. Definition starts from that point.

The class BubbleI—constructor

The constructor sets up the base class and initializes its data members. Note that `_costVal` is set to `IloIntMax` and `_startPoint` to zero. These are the variables which represent the *state* of the neighborhood.

```
BubbleI::BubbleI(IloEnv env,
                 IloIntVarArray perm,
                 IloIntArray values,
                 IloIntVar cost,
                 const char *name)
: IloNHoodI(env, name),
  _soln(),
  _perm(perm),
  _values(values),
  _cost(cost),
  _costVal(IloIntMax),
  _startPoint(0) { }
```

The class BubbleI—member functions

The `reset` member function resets the variables that represent the *state* of the neighborhood, `_costVal` and `_startPoint`, to their original values.

```
void BubbleI::reset() {
  _costVal = IloIntMax;
  _startPoint = 0;
}
```

The `start` member function simply keeps a reference to the current solution.

```
void BubbleI::start(IloSolver, IloSolution soln) {
  _soln = soln;
}
```

The `define` member function returns a delta which swaps the values of two adjacent variables from the `_perm` array. The index is first offset by `_startPoint`.

```
IloSolution BubbleI::define(IloSolver solver, IloInt i) {
  i = (i + _startPoint) % (_perm.getSize() - 1);
```

Now the move is created by adding the variables which will change, together with their new values, to a delta. One point should be borne in mind when you return a delta from `define`. `IloScanNHood` automatically destroys this delta when it has no more use for it—you do not have to handle the memory management yourself. This also means that you should not keep any references to deltas returned from `define`, as these are not guaranteed to be valid thereafter. It is also legal to return an empty handle from `define` if you do not wish a neighbor to be defined for this index.

After the delta is created, the two adjacent variables are added. Their values are set to those in `_soln` (the old values), but swapped over.

```
  IloSolution delta(solver.getEnv());
  delta.add(_perm[i]);
  delta.add(_perm[i + 1]);
  IloInt oldLeft = _soln.getValue(_perm[i]);
  IloInt oldRight = _soln.getValue(_perm[i + 1]);
  delta.setValue(_perm[i], oldRight);
  delta.setValue(_perm[i + 1], oldLeft);
```

Finally, if `_costVal` does not hold its sentinel value, we compute the new cost of the move. Simply, the cost increases by one if the move puts the values in the wrong order when they were in the correct order before, and decreases by one in the other situation.

```
  if (_costVal != IloIntMax) {
    IloInt diff = (_values[oldLeft] < _values[oldRight])
                - (_values[oldLeft] > _values[oldRight]);
    delta.add(_cost);
    delta.setValue(_cost, _costVal + diff);
  }

  return delta;
}
```

> ***Note:*** *It may be tempting to simply return an empty delta if the cost would increase. This is a poor decision for two reasons. First, the neighborhood may not necessarily be used in a context which requires the cost to be reduced at each step (for example, when used with tabu search). Second, the global optimization criterion may not be to minimize* `_cost`*, and so such a rejection of the neighbor would most likely be wrong.*

The size of the neighborhood is one less than the number of values to sort, as adjacent pairs are examined from pair {0, 1} up to pair {`_perm.getSize()-2, _perm.getSize()-1`}.

```
IloInt BubbleI::getSize(IloSolver) const {
  return _perm.getSize() - 1;
}
```

In `notify()` we store the new starting point of the neighborhood as the one after the neighbor being notified. We reduce this value modulo the size of the neighborhood. We also store the cost of the current sorting configuration.

```
void BubbleI::notify(IloSolver solver, IloInt i) {
  _startPoint = (i + 1 + _startPoint) % (_perm.getSize() - 1);
  _costVal = (IloInt)solver.getValue(_cost);
}
```

We store the cost of the current sorting configuration in `notifyOther`.

```
void BubbleI::notifyOther(IloSolver solver, IloSolution) {
  _costVal = (IloInt)solver.getValue(_cost);
}
```

The following function returns the "Bubble Sort" neighborhood. The new neighborhood is allocated on the same environment as is passed to it. This is important as the delete operator of `IloNHoodI` (called via `IloNHood::end`) frees the allocated memory from the `env`. If you allocate your subclass of `IloNHoodI` on a different heap, you must define a delete operator in your subclass.

```
IloNHood Bubble(IloEnv env,
                IloIntVarArray perm,
                IloIntArray values,
                IloIntVar cost,
                const char *name = 0) {
  return new (env) BubbleI(env, perm, values, cost, name);
}
```

Main program

To begin the program, we define an environment, a solver, a model, and a random number generator. The latter will be used to generate the values to sort.

```
int main()
{
  IloEnv env;
  try {
    IloSolver solver(env);
    IloModel m(env);
    IloRandom r(env, 54321);
```

We now generate twenty values to sort, displaying them after they have been generated.

```
    IloInt n = 20;
    IloIntArray values(env, n);
    IloInt i;
    for (i = 0; i < n; i++)
      values[i] = r.getInt(1000);
    cout << values << endl;
```

The permutation array is created, as well as an array to hold the 0-1 cost of each pair of variables in the `perm` array. An all-different constraint is added to the model to ensure that the values of perm really are a permutation (no repeated values). The total cost is formed from the sum of the 0-1 cost variables.

```
IloIntVarArray perm(env, n, 0, n - 1);
IloConstraint allDiff = IloAllDiff(env, perm);
m.add(allDiff);
IloIntVarArray costs(env, n * (n - 1) / 2, 0, 1);
IloInt k = 0;
for (i = 0; i < n - 1; i++) {
  for (IloInt j = i + 1; j < n; j++) {
    m.add(costs[k] == (values(perm[i]) > values(perm[j])));
    k++;
  }
}
IloIntVar cost(env, 0, n * (n - 1) / 2);
m.add(cost == IloSum(costs));
```

The initial solution to the sorting problem is created along with the solution to be used during local search. We add all the variables in `perm` to the solution, and the objective. We extract the model and find a first solution using the `IloGenerate` goal. We then store the solution and display it. The all-different constraint is then removed from the model as it is no longer needed. The “Bubble Sort” neighborhood only swaps values and so cannot violate this constraint.

```
solver.extract(m);
IloGoal g = IloGenerate(env, perm);
IloSolution soln(env);
soln.add(perm);
soln.add(IloMinimize(env, cost));
solver.solve(g);
soln.store(solver);
cout << "---" << endl;
Display(solver, perm, values, cost);
m.remove(allDiff);
```

The neighborhood is created along with the goal which will perform a single move. Then a typical local search optimization loop is entered, continuing until no move can be made, or until the cost value is reduced to zero.

```
IloNHood nhood = Bubble(env, perm, values, cost);
IloGoal move = IloSingleMove(env, soln, nhood, IloImprove(env));
while (soln.getObjectiveValue() != 0 && solver.solve(move))
;
```

The final solution is restored and displayed.

```
    solver.solve(IloRestoreSolution(env, soln));
    cout << "---" << endl;
    Display(solver, perm, values, cost);
  } catch (IloException e) {
    cout << "Caught: " << e << endl;
  }
  env.end();
  return 0;
}
```

Writing a new neighborhood modifier

It is possible to write your own neighborhood modifier using Solver by subclassing the `IloNHoodModifierI` class. This abstract implementation class is used to describe a neighborhood structure that depends on other “child” neighborhood structures. Solver neighborhoods such as `IloConcatenate`, `IloRandomize`, and `IloContinue` belong to this category of neighborhood. Such neighborhoods do not generate neighbors in their own right, but pass back neighbors produced by their children. The main job of a neighborhood modifier is to maintain a mapping between the indices of neighbors for the neighborhood modifier and the corresponding child neighborhoods and indices within these child neighborhoods. The member function `mapIndex` performs this maintenance. Additional behaviors can be added. One example is to manipulate normally produced deltas in some way, for instance by adding additional variables, before they are returned.

The following class forms the concatenation of two neighborhoods.

```
class IloAddNHoodsI : public IloNHoodModifierI {
public:
  IloAddNHoodsI(IloEnv env, IloNHood nhood0, IloNHood nhood1,
                const char *name = 0);
  void mapIndex(IloSolver solver, IloInt i,
                IloNHood &nhood, IloInt &offset) const;
};
```

The constructor builds the base class.

```
IloAddNHoodsI::IloAddNHoodsI(IloEnv env,
                             IloNHood nhood0,
                             IloNHood nhood1,
                             const char *name)
: IloNHoodModifierI(env, nhood0, nhood1, name) { }
```

The correct neighborhood is determined by comparing the index against the size of the first neighborhood. If we map to the first neighborhood, the index is unaffected. Otherwise, it must be reduced by the size of the first neighborhood.

```
void IloAddNHoodsI::mapIndex(IloSolver solver, IloInt i,
                             IloNHood &nhood, IloInt &offset) const {
  IloInt size0 = getNHoodSize(0);
  if (i < size0) { nhood = getNHood(0); offset = i; }
  else           { nhood = getNHood(1); offset = i - size0; }
}
```

Example

The following small example shows how to code a neighborhood which has a side effect on another neighborhood. We assume that the delta will be adjusted in the call to `define`. First, we define the class which performs a null mapping between neighbors of the modifier and neighbors of the child neighborhood.

```
class IloAdjustNeighborsI : public IloNHoodModifierI {
public:
  IloAdjustNeighborsI(IloEnv env, IloNHood nhood, const char *name = 0);
  void mapIndex(IloSolver, IloInt i, IloNHood &nhood, IloInt &offset) const {
    nhood = getNHood(0); offset = i;
  }
  IloSolution define(IloSolver solver, IloInt i);
};
```

The constructor builds the base class.

```
IloAdjustNeighborsI::IloAdjustNeighborsI(IloEnv env,
                                         IloNHood nhood,
                                         const char *name)
: IloNHoodModifierI(env, nhood, name) { }
```

The definition of the delta should be done by a call to the base class. Shortcuts, such as `getNHood(0).define(solver, i)` are not recommended, as they make assumptions about both the implementation of `IloNHoodModifierI::define` and `mapIndex`.

```
IloSolution IloAdjustNeighborsI::define(IloSolver solver, IloInt i) {
  IloSolution delta = IloNHoodModifierI::define(solver, i);
  //
  // Adjust delta here.
  //
  return delta;
}
```

C H A P T E R

25

Writing a Metaheuristic

In this lesson, you will learn how to:

- write a metaheuristic
- use the class `IloMetaHeuristicI`
- implement a "limited tabu search" metaheuristic

Writing a new metaheuristic

It is possible to write your own local search metaheuristic using Solver by subclassing the `IloMetaHeuristicI` class. This section shows you how to do this. The basic behavior of the metaheuristic class is to filter out proposed moves according to the metaheuristic rule. For instance, a greedy metaheuristic filters out all moves which do not improve the cost.

Here, you will find the description of a limited tabu search metaheuristic. Tabu search is a technique that typically makes the best move at each step in the search process, even if that move degrades the search cost. However, in order to avoid the problem of the search being caught in a cycle of making a non-improving move, then the opposite improving one, a *tabu status* is given to the last few moves. Thus, once a move is made some other moves (typically those which would result in a previously visited point) are made illegal (***tabu***). This is to avoid the cycling problem mentioned. The illegality of a tabu move persists for a number of moves after it is first made tabu, this length of time being termed the *tenure*.

In the limited tabu search metaheuristic described here, the tenure is fixed at 1. The tabu metaheuristic is meant to be used in a local search where the value of only a single variable is changed at each move. Then, at each move, the variable that changed its value during the last move is forbidden from changing its value at this iteration. This prevents cycles of length 2, where the search moves continually between two neighboring states. To prevent longer cycles, longer tabu lists can be used, such as in the class `IloTabuSearch` provided with Solver.

To add a little extra complexity, the metaheuristic described also includes a rule which says that the solution must not stray too far (in terms of cost) from the best solution found so far. The maximum *gap* between the current and best solution is specified to the metaheuristic.

Virtual functions of IloMetaHeuristicI

The virtual functions of `IloMetaHeuristicI`, with the exception of `reset` and `complete`, are called from various Solver goals such as `IloStart` and `IloNotify`, and are passed an instance of `IloSolver`. This instance of `IloSolver` is the one executing the Solver goal.

Up to six basic virtual member functions should be defined to implement a metaheuristic. A description of the member functions follows:

```
IloBool start(IloSolver solver, IloSolution soln);
```

The `start` method is called to signify to the metaheuristic that `soln` is the solution from which moves will be generated. By returning `IloFalse` or causing a failure inside this method, you signify that the metaheuristic forbids beginning from `soln`.

From the standpoint of our limited tabu metaheuristic, we want to add constraints to ensure that the variable which changed last time keeps its value and that the cost does not vary too much from the best cost value found so far.

```
IloBool isFeasible(IloSolver solver, IloSolution delta) const;
```

The `isFeasible` method is used as a fast pre-filter. The suggested change to the solution is delivered in `delta`. The move `delta` is a move that has been defined by an instance of `IloNHood`. If the metaheuristic can ascertain that the application of delta to the solution passed in `start` will lead to a solution which is disallowed by the metaheuristic, `IloFalse` should be returned. When this happens the neighborhood move can be discarded without the need to instantiate variables of the model.

From the standpoint of our limited tabu metaheuristic, we can look to see if the delta will change the value of our tabu variable. If it will, we can reject this delta by returning `IloFalse`.

```
IloBool test(IloSolver solver, IloSolution delta);
```

Whereas `isFeasible` is a pre-filter, the `test` member function implements a definitive test of the current move with respect to the metaheuristic. When `test` is called, all model variables are instantiated, and the metaheuristic can use the solution passed in `start` together with the values in the model variables to compute differences.

Alternatively, delta can be used to give the maximum scope of the difference. Not every variable mentioned in delta has necessarily changed value. This can happen when a neighborhood defines a value for a variable in the delta which is the same as the current one.

Be aware that if the metaheuristic is used with the `IloTest` goal, which does not take a neighborhood identifier, this method should be robust to `delta` being an empty handle.

From the standpoint of our limited tabu metaheuristic, we need only return `IloTrue` here, as any tabu or over-costly neighbor will have been rejected by the constraints added in `start` Since the default behavior of `test` returns `IloTrue`, a redefinition of this virtual function is not necessary here.

```
void notify(IloSolver solver, IloSolution delta);
```

The `notify` method is called when a move is about to be accepted, but before the solution passed in `start` is actually stored. This call makes it possible for the metaheuristic to update some internal data structures that will change its behavior at the next move.

Be aware that if the metaheuristic is used with the `IloNotify` goal which does not take a solution reference, this method should be robust to `delta` being an empty handle.

From the standpoint of our limited tabu metaheuristic, we store the variable whose value has changed, so that it can be made tabu at the next move.

```
IloBool complete();
```

When no moves could be taken the last time the neighborhood was explored, the `complete` member function is called. Its job is to decide whether the metaheuristic should indicate that it wants to give up search (by returning `IloTrue`), or keep going (by returning `IloFalse`).

From the standpoint of our limited tabu metaheuristic, we could try relaxing the tabu condition to see if this would allow a move on the next iteration, and return `IloFalse`. If the condition is already relaxed, we could return `IloTrue`.

```
void reset();
```

Subclasses of `IloMetaHeuristicI` often have some memory so that their behavior can be modified from move to move. The `reset` member function should reset any such internal structure. From the standpoint of our limited tabu metaheuristic, we remove the tabu condition.

The class LimitedTabuI

Here is the synopsis of the class LimitedTabuI:

```
class LimitedTabuI: public IloMetaHeuristicI {
private:
  IloNumVar   _previousVar;
  IloNum      _previousValue;
  IloNum      _gap;
  IloNum      _bestCost;
  IloSolution _soln;
public:
  LimitedTabuI(IloEnv env, IloNum gap, const char *name = 0);
  IloBool complete();
  IloBool isFeasible(IloSolver, IloSolution delta) const;
  void notify(IloSolver solver, IloSolution delta);
  void reset();
  IloBool start(IloSolver solver, IloSolution soln);
};
```

The class LimitedTabuI—local variables

The `LimitedTabuI` class is subclassed from `IloMetaHeuricticI` and implements our "limited tabu search" metaheuristic. The local variables have the following functions:

`_previousVar`

This variable holds the variable which was changed on the last move and which cannot change at this move. If this variable is an empty handle, it indicates that either the metaheuristic has just been created or reset, or that no variable in the solution was changed the last time `notify()` was called.

`_previousValue`

This variable holds the value that `previousVar` changed to on the previous move. This is the value it must have for the current move as it cannot change at this iteration.

`_gap`

This is the maximum difference in cost between the current solution and the best solution found.

`_bestCost`

This is the cost of the best solution given to the metaheuristic through `start()`.

`_soln`

This is the solution given to the metaheuristic the last time `start()` was called.

The class LimitedTabuI—constructor

This constructor creates the limited tabu search class.

```
LimitedTabuI::LimitedTabuI(IloEnv env, IloNum gap, const char *name)
  : IloMetaHeuristicI(env, name), _previousVar(), _gap(gap), _soln() {
  assert(_gap >= 0);
}
```

The class LimitedTabuI—member functions

The `complete` member function decides if the metaheuristic should indicate that it would like to terminate the search. If no tabu restriction was present, we end. Otherwise, we remove the tabu restriction and continue.

```
IloBool LimitedTabuI::complete() {
  IloBool done = (_previousVar.getImpl() == 0);
  _previousVar = IloNumVar();
  return done;
}
```

The `isFeasible` member function decides the feasibility of the delta with respect to the tabu restriction. If a variable is tabu and the delta assigns this value to a different value from the current one, the delta can be rejected.

```
IloBool LimitedTabuI::isFeasible(IloSolver, IloSolution delta) const {
  if (_previousVar.getImpl()) {
    if (delta.contains(_previousVar))
      return delta.getValue(_previousVar) != _previousValue;
  }
  return IloTrue;
}
```

The `notify` member function finds the changed variable and its value, and stores them in `previousVar` and `previousValue`. If `delta` is not an empty handle, an optimization can be performed. The only variables we need to examine are those in the delta. In the absence

of this information, we need to scan all variables in the solution to discover the changed variable.

```
void LimitedTabuI::notify(IloSolver solver, IloSolution delta) {
  // Update previousVar and previousValue
  _previousVar = IloNumVar();
  IloSolutionIterator<IloNumVar> it(_soln);
  if (delta.getImpl())
    it = IloSolutionIterator<IloNumVar>(delta);
  while (it.ok()) {
    IloNumVar v = *it;
    if (solver.getValue(v) != _soln.getValue(v)) {
      _previousVar = v;
      _previousValue = solver.getValue(v);
      break;
    }
    ++it;
  }
}
```

The `reset` member function resets to creation state.

```
void LimitedTabuI::reset() {
  _soln = IloSolution();
  _previousVar = IloNumVar();
}
```

The `start` member function ensures that the solution has an objective which is a simple variable (so that we can map from the `IloNumVar` to Solver variables). It stores the best cost, if it is better or if there is no cost stored. It also adds constraints to ensure the cost does not vary too far from the best cost, and that the value of `previousVar` does not change. (This last is done via a `setValue` rather than a constraint.)

```
IloBool LimitedTabuI::start(IloSolver solver, IloSolution solution) {
  if (solution.isObjectiveSet()) {
    IloNumVar obj = solution.getObjectiveVar();
    if (obj.getImpl()) {
      if (solution.getObjective().getSense() == IloObjective::Minimize) {
        if (_soln.getImpl() == 0 || solution.getObjectiveValue() < _bestCost)
          _bestCost = solution.getObjectiveValue();
        if (obj.getType() == ILOINT)
          solver.add(solver.getIntVar(obj) <= _bestCost + _gap);
        else
          solver.add(solver.getFloatVar(obj) <= _bestCost + _gap);
      }
      else {
        if (_soln.getImpl() == 0 || solution.getObjectiveValue() > _bestCost)
          _bestCost = solution.getObjectiveValue();
        if (obj.getType() == ILOINT)
          solver.add(solver.getIntVar(obj) >= _bestCost - _gap);
        else
          solver.add(solver.getFloatVar(obj) >= _bestCost - _gap);
      }
      if (_previousVar.getImpl())
```

```
        solver.setValue(_previousVar, _previousValue);
    }
    else {
      throw "LimitedTabuI::start - "
            "Solution objective must be a simple variable";
    }
  }
  else {
    throw "LimitedTabuI::start - Solution has no objective";
  }
  _soln = solution;
  return IloTrue;
}
```

The following function returns the limited tabu metaheuristic. The new metaheuristic is allocated on the same environment as is passed to it. This is important as the `delete` operator of `IloMetaHeuristicI` (called via `IloMetaHeuristic::end`) frees the allocated memory from the `env`. If you allocate your subclass of `IloMetaHeuristicI` on a different heap, you must define a delete operator in your subclass.

```
IloMetaHeuristic LimitedTabu(IloEnv env, IloNum gap, const char *name = 0) {
  return new (env) LimitedTabuI(env, gap, name);
}
```

C H A P T E R

26

Combining Complete and Local Search: Locating Warehouses

In this lesson, you will learn more about how to:

- combine complete and local search
- use custom goals
- store solutions

This chapter shows how complete search and local search can be used together to obtain the optimal solution to a complex problem.

Describe

In this problem we must make a decision about how many warehouses to build and where to build them. We can choose from several potential warehouses, each of which has a specific limited quantity of goods available (capacity) and a specific cost, `buildCosts`, to build. In addition, we have a number of clients that must each be served by one of the warehouses we build. Each of these clients demands a specific quantity of goods. The total quantity demanded by all clients served by a warehouse cannot exceed its capacity.

Each warehouse also has several costs, `serveCosts`, that represent respectively the cost if each client was served from that warehouse. The cost for a warehouse is the sum of the

service costs of all clients served from that warehouse and the cost to build the warehouse. The total cost of the solution is the sum of all the warehouse costs. The goal of the problem is to provide service to all clients within the constraint of warehouse capacity while minimizing total cost.

Model

We represent the capacities of the warehouses, the demands of the clients, the building costs and service costs using numerical arrays.

Create the model

We create an environment to handle input/output, memory allocation, and other general services for the variables of our problem. We then create a model to hold the variables that define the problem:

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

Read problem data from a file

We define a function `readProblem` to read the problem data from a file. We define each of the problem variables: the number of potential warehouses (`nbWhouses`), the total number of clients (`nbClients`), the array of capacities of the warehouses (`capacities`), the array of the demands of each of the clients (`demands`), the array of the costs to build each of the

warehouses (`buildCosts`), and the two-dimensional array of the costs for each potential warehouse to service each of the clients (`serveCosts`).

```
// Function to read the problem from a file

void readProblem(IloEnv env,
                 istream &is,
                 IloInt &nbClients,
                 IloInt &nbWhouses,
                 IloIntArray &capacities,
                 IloIntArray &demands,
                 IloIntArray &buildCosts,
                 IloArray<IloIntArray> &serveCosts) {
  is >> nbWhouses >> nbClients;
  capacities = IloIntArray(env, nbWhouses);
  demands = IloIntArray(env, nbClients);
  buildCosts = IloIntArray(env, nbWhouses);
  serveCosts = IloArray<IloIntArray>(env, nbClients);
  env.out() << nbClients << " clients, " << nbWhouses << " warehouses" << endl;
  IloInt i;
  for (i = 0; i < nbWhouses; i++) {
    is >> capacities[i];
    is >> buildCosts[i];
  }
  env.out() << "Capacities: " << capacities << endl;
  env.out() << "Build costs: " << buildCosts << endl;
  for (i = 0; i < nbClients; i++) {
    serveCosts[i] = IloIntArray(env, nbWhouses);
    is >> demands[i];
    for (IloInt j = 0; j < nbWhouses; j++) is >> serveCosts[i][j];
  }
  env.out() << "Demands: " << demands << endl;
}
```

The following code opens the input file; determines whether to perform local search, complete (proof) search, or both (the default is to perform both); defines the variable arrays;

and uses the function `readProblem` to read the problem data into those arrays. The choice of which type of search to perform can also be made on the command line.

```
// Open input file.
ifstream infile;
const char *fname;
if (argc > 1) fname = argv[1];
else          fname = "../../../examples/data/cap71.txt";
infile.open(fname);
if (!infile.good()) {
  env.out() << "Could not open " << fname << endl;
  env.end();
  return 0;
}
// Decide on local search, proof, or local search, then proof.
const char *lp = "lp";
if (argc > 2) lp = argv[2];
IloBool local = IloFalse;
IloBool proof = IloFalse;
if (strchr(lp, 'l')) local = IloTrue;
if (strchr(lp, 'p')) proof = IloTrue;

// Read in the problem
IloInt nbWhouses, nbClients;
IloIntArray capacities(env);
IloIntArray demands(env);
IloIntArray buildCosts(env);
IloArray<IloIntArray> serveCosts;
readProblem(env, infile, nbClients, nbWhouses,
            capacities, demands, buildCosts, serveCosts);
infile.close();
```

Build constraints and variables

We build the problem variables using the classes and expressions we have just defined. The numerical array `offer` is used to store the number of the warehouse to which each client is assigned. The numerical array `transCost` is used to store the cost of serving a client from the chosen warehouse. The 0-1 variable array `open` is used to store the values that indicate whether a warehouse is built or not. The numerical array `load` is used to store the sum of the demands of the clients served by each warehouse.

```
// Build all the constraints.
IloIntVarArray offer(env, nbClients, 0, nbWhouses - 1);
IloIntVarArray transCost(env, nbClients, 0, IloIntMax);
IloIntVarArray open(env, nbWhouses, 0, 1);
IloIntVarArray load(env, nbWhouses);
```

We add constraints to ensure that the load of customers assigned to a warehouse do not exceed the warehouse capacity.

```
IlcInt i;
for (i = 0; i < nbWhouses; i++)
  load[i] = IloIntVar(env, 0, capacities[i]);
m.add(IloPack(env, load, offer, demands));
```

We add a constraint that specifies that a warehouse is built (`open`) if and only if it has assigned clients to serve.

```
// Only open if customers are served.
for (i = 0; i < nbWhouses; i++)
  m.add(open[i] == (load[i] != 0));
m.add(IloScalProd(open, capacities) >= IloSum(demands));
```

We also add a variable, `cost`, and set it as equal to the sum of the total client service costs for each warehouse (`transCost`) plus the total building costs (`buildCosts`) for all open warehouses.

```
// Total cost is sum of shipment and build costs
IloIntVar cost(env, 0, IloIntMax);
cost.setName("Cost\t");
m.add(cost == IloSum(transCost) + IloScalProd(open, buildCosts));
```

Finally, we add the element constraint that links the client service costs for a warehouse to the customers served by that warehouse.

```
// Element constraints.
for (i = 0; i < nbClients; i++) {
  IloIntArray costs(env, nbWhouses);
  for (IlcInt j = 0; j < nbWhouses; j++)
    costs[j] = serveCosts[i][j];
  IloIntVar dRes(env, costs);
  m.add(IloTableConstraint(env, dRes, costs, offer[i]));
  m.add(transCost[i] == dRes);
}
```

Define search functions

We first define a function, `TabuSearch`, to perform a simple tabu search with a short-term memory. We define a neighborhood using `IloFlip` and randomize it using `IloRandomize`. We define a goal, `move`, that makes a single move and, if the solution is improved, stores it in the solution object `whole`. The function `IloSingleMove` scans the randomized

neighborhood `nh` using `sol` as the current solution. The metaheuristic `mh` uses tabu search to filter moves, and the search selector `selectMove` selects a move that minimizes the cost.

```
// Function to perform simple tabu search with a short term memory

void TabuSearch(IloSolver solver,
                IloIntVarArray open,
                IloSolution sol,
                IloSolution whole,
                IloGoal subGoal,
                IloInt iter = IloIntMax) {
  IloEnv env = solver.getEnv();
  IloRandom rand(env);
  IloNHood nh = IloRandomize(env, IloFlip(env, open), rand);
  IloMetaHeuristic mh = IloTabuSearch(env, 2);
  IloIntVar cost = sol.getObjectiveVar();
  IloSearchSelector selectMove = IloMinimizeVar(env, cost, 1.0);

  // Goal makes a move, and updates the (whole) best solution if better.
  IloGoal move = IloSingleMove(env, sol, nh, mh, selectMove, subGoal) &&
                 IloStoreBestSolution(env, whole);
```

Finally, we define a standard optimization loop that continues the search until all iterations have been completed or the tabu search is complete.

```
  // Standard optimization loop.
  do {
    IloInt i = 0;
    while (--iter >= 0 && solver.solve(move)) {
      solver.out() << "Iter = " << ++i << ", "
                   << sol.getObjectiveValue() << " : ";
      for (IloInt j = 0; j < open.getSize(); j++)
        solver.out() << solver.getValue(open[j]);
      solver.out() << " (Best = " << whole.getObjectiveValue() << ")" << endl;
    }
  } while (iter > 0 && !mh.complete());
}
```

We define a goal, `IloSetOffer`, that chooses a client with the smallest number of available open warehouses (an open warehouse is available if it has sufficient remaining capacity to

satisfy the client's demand), and assign the client to the available open warehouse with the lowest cost.

```
// Goal to assign customers to specific warehouses

ILCGOAL4(IlcSetOffer, IlcIntVarArray, offer, IlcIntVarArray, open,
                      IloArray<IloIntArray>, costs, IloIntArray, buildCosts) {
  IlcInt i = IlcChooseMinSizeInt(offer);
  if (i >= 0) {
    // Choose closest warehouse first.
    IlcInt best = -1;
    IlcInt lowest = IlcIntMax;
    for (IlcIntExpIterator it(offer[i]); it.ok(); ++it) {
      IloIntArray row = costs[i];
      IlcInt cost = row[*it] + buildCosts[*it] * (open[*it].getMin() == 0);
      if (cost <= lowest) { lowest = cost; best = *it; }
    }
    return IlcAnd(IlcOr(IlcSetValue(offer[i], best),
                        IlcRemoveValue(offer[i], best)),
                  this);
  }
  return 0;
}
```

Solve

We use local search to decide whether or not to open a specific warehouse. At each move, we use complete search to assign clients to warehouses, thus producing a hybrid search method.

Create the solution object

We create two solution objects. The first, `sol`, holds only the variables `open`, and is used to perform a local search over which warehouses are open (built) and which are closed (not built). The second solution `whole` maintains the assignment of a warehouse to each customer and so represents a proper solution to the problem. It is only used to store solutions as they are found and not used actively in the local search process. As mentioned previously,

local search decides which warehouses will be open or closed, but the assignment of warehouses to customers is performed by a complete search goal, executed after each move.

```
// Set up two solutions.
// sol: Holds only the 0-1 variables which defining the open warehouses
// whole: Holds the variables which assign a warehouse to each client
// NOTE: whole defines a solution, whereas sol only defines which
//       warehouses will be used.  The clients must be then be assigned
//       a specific warehouse each for a complete solution.
//       We perform local search over the openness of warehouses,
//       and at each move assign the customers to warehouses.
IloSolution sol(env);
IloObjective obj = IloMinimize(env, cost);
sol.add(obj);
for (i = 0; i < nbWhouses; i++) {
  char name[10];
  sprintf(name, "O-%ld", i);
  open[i].setName(name);
  sol.add(open[i]);
}
IloSolution whole(env);
whole.add(obj);
whole.add(offer);
```

Use local search

We create a goal to generate an initial solution, using the goal `IloSetOffer`. After execution of this goal we store the initial solution in both our solution instances.

```
IloGoal g;
IloInt upperBound = IloIntMax;
IloSolver solver(env);

// Inital solution
// Complete search goal to assign clients to warehouses
IloGoal generateOffer =
        IloSetOffer(env, offer, open, serveCosts, buildCosts);
g = generateOffer &&
    IloStoreSolution(env, sol) &&
    IloStoreSolution(env, whole);
solver.extract(m);
if (solver.solve(g)) {
  DisplaySolution(solver, "Initial Solution", cost, offer);
}
else {
  solver.out() << "No initial solution" << endl;
  env.end();
  return 0;
}
```

We perform the local search if desired. We build the subgoal to pass to the `TabuSearch` function. This subgoal is an instance of `IloSetOffer` and will open warehouses to

customers at each move. When local search is complete, we take the value of the best solution as an upper bound and store it.

```
if (local) {
  // Local Search

  // The sub-goal to assign clients to open warehouses, at minimum cost
  IloGoal subGoal = IloSelectSearch(env, generateOffer,
                                    IloMinimizeVar(env, cost, 1));

  // Run the tabu search
  TabuSearch(solver, open, sol, whole, subGoal, 30);
}
// Get the upper bound
upperBound = (IloInt)whole.getObjectiveValue();
```

Use complete search for optimization

If the option to use complete search has been selected, we first set the upper bound for the search. If local search has been run first, the best objective value obtained in local search is used as the upper bound. Thus, local search provides a method for pruning the search tree used by the complete search method.

We create a standard optimization goal to solve the problem. If local search has not been performed, we extract the model for the solver. We also add code to output information about how the solution is found:

```
if (proof) {
  cost.setUb(upperBound - 1);
  g = IloGenerate(env, open) &&
      generateOffer &&
      IloStoreSolution(env, whole);
  g = IloSelectSearch(env, g, IloMinimizeVar(env, cost, 1));

  // Optimization loop
  if (!local) solver.extract(m);
  if (solver.solve(g)) {
    solver.out() << "Complete search found solution of cost "
                 << solver.getValue(cost) << endl;
    solver.out() << solver.getIntVarArray(offer) << endl;
    if (local) {
      solver.out() << "Better solution found by complete search" << endl;
      solver.out() << "Local search was "
                   << 100.0 * (upperBound / whole.getObjectiveValue() - 1)
                   << "% above optimal" << endl;
    }
  }
  else {
    if (local) solver.out() << "No better solution found" << endl;
    else       solver.out() << "No solution" << endl;
  }
}
```

After the search is complete, the best solution is restored and displayed.

Displaying information about a solution

We define the function `DisplaySolution` to display the solution:

```
// Function to display the solution

void DisplaySolution(IloSolver solver,
                     const char *phrase,
                     IloIntVar cost,
                     IloIntVarArray offer) {
  solver.out() << endl << phrase << endl
               << "[" << solver.getValue(cost) << "]" << endl
               << solver.getIntVarArray(offer) << endl;
}
```

Complete program

The complete program follows. You can also view the entire program online in the file `YourSolverHome/examples/src/lswhouse.cpp`.

```
#include <ilsolver/iimls.h>

ILOSTLBEGIN


// Function to read the problem from a file

void readProblem(IloEnv env,
                 istream &is,
                 IloInt &nbClients,
                 IloInt &nbWhouses,
                 IloIntArray &capacities,
                 IloIntArray &demands,
                 IloIntArray &buildCosts,
                 IloArray<IloIntArray> &serveCosts) {
  is >> nbWhouses >> nbClients;
  capacities = IloIntArray(env, nbWhouses);
  demands = IloIntArray(env, nbClients);
  buildCosts = IloIntArray(env, nbWhouses);
  serveCosts = IloArray<IloIntArray>(env, nbClients);
  env.out() << nbClients << " clients, " << nbWhouses << " warehouses" << endl;
  IloInt i;
  for (i = 0; i < nbWhouses; i++) {
    is >> capacities[i];
    is >> buildCosts[i];
  }
  env.out() << "Capacities: " << capacities << endl;
  env.out() << "Build costs: " << buildCosts << endl;
  for (i = 0; i < nbClients; i++) {
```

```
    serveCosts[i] = IloIntArray(env, nbWhouses);
    is >> demands[i];
    for (IloInt j = 0; j < nbWhouses; j++) is >> serveCosts[i][j];
  }
  env.out() << "Demands: " << demands << endl;
}

// Function to display the solution

void DisplaySolution(IloSolver solver,
                     const char *phrase,
                     IloIntVar cost,
                     IloIntVarArray offer) {
  solver.out() << endl << phrase << endl
               << "[" << solver.getValue(cost) << "]" << endl
               << solver.getIntVarArray(offer) << endl;
}

// Function to perform simple tabu search with a short term memory

void TabuSearch(IloSolver solver,
                IloIntVarArray open,
                IloSolution sol,
                IloSolution whole,
                IloGoal subGoal,
                IloInt iter = IloIntMax) {
  IloEnv env = solver.getEnv();
  IloRandom rand(env);
  IloNHood nh = IloRandomize(env, IloFlip(env, open), rand);
  IloMetaHeuristic mh = IloTabuSearch(env, 2);
  IloIntVar cost = sol.getObjectiveVar();
  IloSearchSelector selectMove = IloMinimizeVar(env, cost, 1.0);

  // Goal makes a move, and updates the (whole) best solution if better.
  IloGoal move = IloSingleMove(env, sol, nh, mh, selectMove, subGoal) &&
                 IloStoreBestSolution(env, whole);
  // Standard optimization loop.
  do {
    IloInt i = 0;
    while (--iter >= 0 && solver.solve(move)) {
      solver.out() << "Iter = " << ++i << ", "
                   << sol.getObjectiveValue() << " : ";
      for (IloInt j = 0; j < open.getSize(); j++)
        solver.out() << solver.getValue(open[j]);
      solver.out() << " (Best = " << whole.getObjectiveValue() << ")" << endl;
    }
  } while (iter > 0 && !mh.complete());
}

// Goal to assign customers to specific warehouses

ILCGOAL4(IlcSetOffer, IlcIntVarArray, offer, IlcIntVarArray, open,
                      IloArray<IloIntArray>, costs, IloIntArray, buildCosts) {
  IlcInt i = IlcChooseMinSizeInt(offer);
  if (i >= 0) {
    // Choose closest warehouse first.
    IlcInt best = -1;
    IlcInt lowest = IlcIntMax;
```

```
    for (IlcIntExpIterator it(offer[i]); it.ok(); ++it) {
      IloIntArray row = costs[i];
      IlcInt cost = row[*it] + buildCosts[*it] * (open[*it].getMin() == 0);
      if (cost <= lowest) { lowest = cost; best = *it; }
    }
    return IlcAnd(IlcOr(IlcSetValue(offer[i], best),
                        IlcRemoveValue(offer[i], best)),
                  this);
  }
  return 0;
}

ILOCPGOALWRAPPER4(IloSetOffer, solver, IloIntVarArray, offer, IloIntVarArray, 
open,
                  IloArray<IloIntArray>, costs, IloIntArray, buildCosts) {
  return IlcSetOffer(solver,
                     solver.getIntVarArray(offer),
                     solver.getIntVarArray(open),
                     costs,
                     buildCosts);
}

// Main program

int main(int argc, char *argv[]){
  IloEnv env;
  try {
    IloModel m(env);
    // Open input file.
    ifstream infile;
    const char *fname;
    if (argc > 1) fname = argv[1];
    else          fname = "../../../examples/data/cap71.txt";
    infile.open(fname);
    if (!infile.good()) {
      env.out() << "Could not open " << fname << endl;
      env.end();
      return 0;
    }
    // Decide on local search, proof, or local search, then proof.
    const char *lp = "lp";
    if (argc > 2) lp = argv[2];
    IloBool local = IloFalse;
    IloBool proof = IloFalse;
    if (strchr(lp, 'l')) local = IloTrue;
    if (strchr(lp, 'p')) proof = IloTrue;

    // Read in the problem
    IloInt nbWhouses, nbClients;
    IloIntArray capacities(env);
    IloIntArray demands(env);
    IloIntArray buildCosts(env);
    IloArray<IloIntArray> serveCosts;
    readProblem(env, infile, nbClients, nbWhouses,
                capacities, demands, buildCosts, serveCosts);
    infile.close();
    // Build all the constraints.
    IloIntVarArray offer(env, nbClients, 0, nbWhouses - 1);
```

```
IloIntVarArray transCost(env, nbClients, 0, IloIntMax);
IloIntVarArray open(env, nbWhouses, 0, 1);
IloIntVarArray load(env, nbWhouses);
IlcInt i;
for (i = 0; i < nbWhouses; i++)
  load[i] = IloIntVar(env, 0, capacities[i]);
m.add(IloPack(env, load, offer, demands));

// Only open if customers are served.
for (i = 0; i < nbWhouses; i++)
  m.add(open[i] == (load[i] != 0));
m.add(IloScalProd(open, capacities) >= IloSum(demands));

// Total cost is sum of shipment and build costs
IloIntVar cost(env, 0, IloIntMax);
cost.setName("Cost\t");
m.add(cost == IloSum(transCost) + IloScalProd(open, buildCosts));

// Element constraints.
for (i = 0; i < nbClients; i++) {
  IloIntArray costs(env, nbWhouses);
  for (IlcInt j = 0; j < nbWhouses; j++)
    costs[j] = serveCosts[i][j];
  IloIntVar dRes(env, costs);
  m.add(IloTableConstraint(env, dRes, costs, offer[i]));
  m.add(transCost[i] == dRes);
}

// Set up two solutions.
// sol: Holds only the 0-1 variables which defining the open warehouses
// whole: Holds the variables which assign a warehouse to each client
// NOTE: whole defines a solution, whereas sol only defines which
//       warehouses will be used.  The clients must be then be assigned
//       a specific warehouse each for a complete solution.
//       We perform local search over the openness of warehouses,
//       and at each move assign the customers to warehouses.
IloSolution sol(env);
IloObjective obj = IloMinimize(env, cost);
sol.add(obj);
for (i = 0; i < nbWhouses; i++) {
  char name[10];
  sprintf(name, "O-%ld", i);
  open[i].setName(name);
  sol.add(open[i]);
}
IloSolution whole(env);
whole.add(obj);
whole.add(offer);
IloGoal g;
IloInt upperBound = IloIntMax;
IloSolver solver(env);

// Inital solution
// Complete search goal to assign clients to warehouses
IloGoal generateOffer =
        IloSetOffer(env, offer, open, serveCosts, buildCosts);
g = generateOffer &&
    IloStoreSolution(env, sol) &&
```

```
        IloStoreSolution(env, whole);
    solver.extract(m);
    if (solver.solve(g)) {
      DisplaySolution(solver, "Initial Solution", cost, offer);
    }
    else {
      solver.out() << "No initial solution" << endl;
      env.end();
      return 0;
    }
    if (local) {
      // Local Search

      // The sub-goal to assign clients to open warehouses, at minimum cost
      IloGoal subGoal = IloSelectSearch(env, generateOffer,
                                        IloMinimizeVar(env, cost, 1));

      // Run the tabu search
      TabuSearch(solver, open, sol, whole, subGoal, 30);
    }
    // Get the upper bound
    upperBound = (IloInt)whole.getObjectiveValue();
    if (proof) {
      cost.setUb(upperBound - 1);
      g = IloGenerate(env, open) &&
          generateOffer &&
          IloStoreSolution(env, whole);
      g = IloSelectSearch(env, g, IloMinimizeVar(env, cost, 1));

      // Optimization loop
      if (!local) solver.extract(m);
      if (solver.solve(g)) {
        solver.out() << "Complete search found solution of cost "
                     << solver.getValue(cost) << endl;
        solver.out() << solver.getIntVarArray(offer) << endl;
        if (local) {
          solver.out() << "Better solution found by complete search" << endl;
          solver.out() << "Local search was "
                       << 100.0 * (upperBound / whole.getObjectiveValue() - 1)
                       << "% above optimal" << endl;
        }
      }
      else {
        if (local) solver.out() << "No better solution found" << endl;
        else       solver.out() << "No solution" << endl;
      }
    }
    // Reporting final solution
    cost.setUb(whole.getObjectiveValue());
    g = IloRestoreSolution(env, whole);
    solver.solve(g);
    DisplaySolution(solver, "Final Solution", cost, offer);
  } catch(IloException& ex) {
    cout << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Output

The program initially produces the following output. The member function `IloSolver::printInformation` displays details about how Solver solved the problem.

```
50 clients, 16 warehouses
Capacities: [58268, 58268, 58268, 58268, 58268, 58268, 58268, 58268, 58268,
58268, 58268, 58268, 58268, 58268, 58268, 58268]
Build costs: [7500, 7500, 7500, 7500, 7500, 7500, 7500, 7500, 7500, 7500, 0,
7500, 7500, 7500, 7500, 7500]
Demands: [146, 87, 672, 1337, 31, 559, 2370, 1089, 33, 32, 5495, 904, 1466,
143, 615, 564, 226, 3016, 253, 195, 38, 807, 551, 304, 814, 337, 4368, 577,
482, 495, 231, 322, 685, 12912, 325, 366, 3671, 2213, 705, 328, 1681, 1117,
275, 500, 2241, 733, 222, 49, 1464, 222]

Initial Solution
[965952]
IlcIntVarArrayI[[10] [10] [10] [10] [10] [10] [1] [10] [10] [10] [3] [10] [10]
[10] [10] [7] [3] [8] [3] [7] [3] [9] [10] [10] [10] [10] [12] [10] [10] [14]
[10] [10] [10] [2] [10] [3] [2] [10] [7] [10] [10] [3] [7] [9] [12] [7] [7] [7]
[10] [10]]
Iter = 1, 943477 : 0111010111101010 (Best = 943477)
Iter = 2, 939949 : 0111010111111010 (Best = 939949)
Iter = 3, 937269 : 1111010111111010 (Best = 937269)
Iter = 4, 934199 : 1111010111111000 (Best = 934199)
Iter = 5, 935152 : 1111010111111001 (Best = 934199)
Iter = 6, 935445 : 1111010110111001 (Best = 934199)
Iter = 7, 933569 : 1111011110111001 (Best = 933569)
Iter = 8, 932616 : 1111011110111000 (Best = 932616)
Iter = 9, 933876 : 1111011110101000 (Best = 932616)
Iter = 10, 935883 : 1111011110101100 (Best = 932616)
Iter = 11, 938953 : 1111011110101110 (Best = 932616)
Iter = 12, 937692 : 1111011110111110 (Best = 932616)
Iter = 13, 935686 : 1111011110111010 (Best = 932616)
Iter = 14, 932616 : 1111011110111000 (Best = 932616)
Iter = 15, 933569 : 1111011110111001 (Best = 932616)
Iter = 16, 934829 : 1111011110101001 (Best = 932616)
Iter = 17, 937899 : 1111011110101011 (Best = 932616)
Iter = 18, 936946 : 1111011110101010 (Best = 932616)
Iter = 19, 935686 : 1111011110111010 (Best = 932616)
Iter = 20, 932616 : 1111011110111000 (Best = 932616)
Iter = 21, 933569 : 1111011110111001 (Best = 932616)
Iter = 22, 934829 : 1111011110101001 (Best = 932616)
Iter = 23, 937899 : 1111011110101011 (Best = 932616)
Iter = 24, 936946 : 1111011110101010 (Best = 932616)
Iter = 25, 935686 : 1111011110111010 (Best = 932616)
Iter = 26, 932616 : 1111011110111000 (Best = 932616)
Iter = 27, 933569 : 1111011110111001 (Best = 932616)
Iter = 28, 934829 : 1111011110101001 (Best = 932616)
Iter = 29, 937899 : 1111011110101011 (Best = 932616)
Iter = 30, 936946 : 1111011110101010 (Best = 932616)
No better solution found
```

```
Final Solution
[932616]
IlcIntVarArrayI[[7] [11] [0] [5] [7] [0] [1] [2] [7] [7] [3] [10] [5] [0] [6]
[7] [3] [8] [3] [6] [3] [6] [10] [0] [11] [10] [12] [10] [10] [0] [0] [10] [0]
[2] [11] [11] [5] [5] [7] [5] [10] [3] [7] [6] [12] [7] [7] [6] [5] [11]]
```

Here is how to interpret this program output:

The initial output displays the input problem data: the number of clients, the number of warehouses, the array of capacities of the warehouses, the array of building costs of the warehouses, and the array of the demands of the customers.

The initial solution uses nine warehouses and has a total cost of 965952.

After 30 iterations, the best solution found by local search is passed to the complete search, which is unable to find a better solution than that found by local search. The final solution uses eleven warehouses and has a total cost of 932616.

The `IlcIntVarArray` indicates the warehouse to which each of the 50 customers is assigned.

CHAPTER

27

Minimizing Talent Wait Cost Using LNS

This lesson introduces the concept of *large neighborhood search* (LNS). LNS is a search technique which is a hybrid of traditional neighborhood search techniques and constraint programming. The basic idea is simple: at each step of the local search method, a subset of the decision variables is chosen (invariably in a randomized fashion) and referred to as the *fragment*. All decision variables *not* in the fragment must keep their values as in the current solution. All variables in the fragment are free to take their values from their initial domains (subject to constraint propagation). A constraint programming goal is then used to complete the assignment by instantiating the variables in the fragment. This goal is usually subject to some acceptance conditions or constraints, for instance that any new assignment is better than the current one in the case of a greedy search. The completion goal is also normally limited by a fail limit to avoid the exponential run-times that can be seen using complete search. Regardless of whether a better solution was found during completion, the process is repeated from the selection of a new fragment. As in traditional local search procedures, a number of iterations or a time limit usually terminates the search.

LNS is a successful technique which can be applied to a range of problems. The crucial part is the selection of a fragment. One method is to select a fragment completely at random. While this can work in some cases, normally there is a better way to select a fragment that can be determined by some consideration of the problem. This example introduces you to the LNS framework provided in Solver IIM and addresses such key points as fragment generation at the end of the chapter.

Make a copy of the example file `YourSolverHome/examples/src/tutorial/lstalent_partial.cpp` and open this copy 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 section. At the end of the section, you will have completed the example and you can compile and run the program.

Problem description

This lesson addresses a problem known as the *talent scheduling problem*, which involves deciding on the best order in which to shoot different film scenes in order to minimize actor waiting costs.

In this problem, there are *m* actors required to shoot *n* scenes. Each scene *s* has a filming duration *d(s)* and each actor *a* has a cost per unit duration (pay rate) of *c(a)*. Not all actors are required for all scenes, and a predicate *r(a,s)* determines if actor *a* is required in scene *s*.

The scenes are shot in a certain order, and once an actor arrives on set, he will remain there until his last scene has been shot. All actors are paid for their time on set, including any time waiting for scenes to be shot which do not involve them. The objective of the search is to decide on the best *filming order* of scenes such that the total actor waiting cost is minimized.

For example, see the following 3 actor, 5 scene problem:

Table 27.1 *A 3-actor, 5-scene problem*

Scene1	**1**	**2**	**3**	**4**	**5**	**Cost**
Actor 1	1	0	1	0	0	5
Actor 2	0	1	1	0	1	7
Actor 3	0	1	1	1	0	2
Duration	**3**	**1**	**4**	**2**	**2**	

This problem has five scenes of filming durations 3, 1, 4, 2 and 2, and three actors who are paid 5, 7 and 2 units of currency per time unit. The central section of the table indicates which actors are required for each scene, a 1 meaning that the actor is required for the corresponding scene, and a 0 meaning that he is not.

If the scenes were filmed in the numerical order given, then actor 1 would have to wait during scene 2 at a cost of 5 units, actor 2 would have to wait during scene 4 at a cost of 14 units, and actor 3 would not wait, incurring no additional cost. The total cost of this filming schedule would be 5+14=19.

Optimal solutions for this problem have a cost of 4. Such a solution is given below:

Table 27.2 *An optimal solution of the 3-actor, 5-scene problem*

Scene1	**1**	**3**	**2**	**5**	**4**	**Cost**
Actor 1	1	1	0	0	0	5
Actor 2	0	1	1	1	0	7
Actor 3	0	1	1	0	1	2
Duration	**3**	**4**	**1**	**2**	**2**	

Here, only actor 3 waits for 2 time units during the filming of scene 5.

Problem representation: Reading the problem data

In this example, the problem data will be read from a file. The data to be read are the numbers of actors and scenes, the pay rate of the actors, and the filming durations of the scenes. In addition, the presence (or otherwise) of an actor in any particular scene is read. The following initial part of the program has been provided for you:

```
int main(int argc, const char * argv[]) {
  IloEnv env;
  try {
    const char * inputFile = "../../../examples/data/rehearsal.txt";
    IloInt seed = 1;
    if (argc > 1)
      inputFile = argv[1];
    if (argc > 2)
      seed = atoi(argv[2]);

    IloIntArray actorPay(env);
    IloIntArray sceneDuration(env);
    IloArray<IloIntSet> actorInScene(env);
    IloBool ok = ReadData(inputFile, actorPay, sceneDuration, actorInScene);
```

This code accepts as command line arguments first, the filename from which the problem data should be read, and second, a random seed with which to seed Solver's random number generator. Varying the seed will vary the behavior of the resulting search procedure.

Three empty arrays are created which will be filled when the data file is read. `actorPay` holds the pay per unit time for each actor. `sceneDuration` holds the filming duration of each scene. Finally, `actorInScene` holds, for each actor, the set of scenes in which he appears.

Thereafter, the problem data are read by filling the three aforementioned arrays. The function `ReadData` carries out this process, returning a true value if the data were read without error and false otherwise.

The `ReadData` function is provided for you and can be simply understood from the code itself.

Problem representation: Model

The next step is to create a Concert Technology model of the problem from the data that have been read. The following code is provided for you.

```
// Create the decision variables, cost, and the model
IloInt numScenes = sceneDuration.getSize();
IloInt numActors = actorPay.getSize();
IloIntVarArray scene(env, numScenes, 0, numScenes - 1);
IloIntVar idleCost(env, 0, IloIntMax);
IloModel model = BuildModel(
  scene, idleCost, actorPay, sceneDuration, actorInScene
);
```

This code creates the main decision variables of the problem `scene`. Element *i* of this array will specify which scene is to be filmed in slot *i*. The code also creates the cost variable `idleCost`. Then a call to the function `BuildModel` is made which creates the model which calculates the idle cost of a particular filming order, given actor pay and scene involvement, as well as the scene filming durations.

Now you create the model of the talent scheduling problem. Go to the function `BuildModel` in your file. The head of the function is provided for you. It simply creates the Concert Technology model object and retrieves the number of actors and scenes to film.

```
// Build the talent scheduling model
IloModel BuildModel(IloIntVarArray scene,
                    IloIntVar idleCost,
                    IloIntArray actorCost,
                    IloIntArray sceneDuration,
                    IloArray<IloIntSet> actorInScene) {
  IloEnv env = scene.getEnv();
  IloInt numScenes = scene.getSize();
  IloInt numActors = actorCost.getSize();
  IloModel model(env);
```

The decision variables of the problem are, for each time slot, which scene to film in that particular time slot, and indeed this model is the most intuitive one for most people. However, it turns out that the model is easier to specify if a secondary model is used. Instead of having one variable per time slot, indicating the scene, the dual model has one variable per scene, indicating in which time slot that scene is filmed.

Here, you will create this secondary model. The two models are connected together using the *inverse* constraint which ensures that for two arrays of constrained variables x and y of equal size, for any index i, $y[x[i]] = x[y[i]] = i$.

This constraint also has the virtue of ensuring that all variables in x must take on different values, as must all those in y; if this were not the case, the inversion would not be possible.

You will now create the secondary model.

Step 1 Create the secondary model

Add the following code after the comment

```
// Make the slot-based secondary model
```

```
IloIntVarArray slot(env, numScenes, 0, numScenes - 1);
model.add(IloInverse(env, scene, slot));
```

You will now build the cost function which is the sum of the waiting cost of each actor. The basic loop is already provided for you:

```
  // Expression representing the global cost
  IloIntExpr cost(env);

  // Loop over all actors, building cost
  for (IloInt a = 0; a < numActors; a++) {
    // Expression for the waiting time for this actor
    IloIntExpr actorWait(env);


    // Calculate the first and last slots where this actor plays


    // If an actor is not in a scene,
    // he waits if he is on set when the scene is filmed


    // Accumulate the cost of waiting time for this actor
    cost += actorCost[a] * actorWait;
  }

  model.add(idleCost == cost);
  return model;

}
```

You will write the code which will calculate the expression `actorWait` for each actor. There are three points to note here:

- An actor is on set between his first and last scenes.
- An actor will only wait for scenes filmed while he is on set.

◆ An actor can only wait for scenes he does not appear in.

First of all, you will work out when an actor is on set by creating variables which represent the first and last time slots where the actor is needed. To do this, you first create an array of variables which contains all the slots where the actor plays.

Step 2 Create the slots in which the actor plays

Insert the following code after the comment

```
// Calculate the first and last slots where this actor plays
```

```
IloIntVarArray position(env);
for (IloIntSet::Iterator it(actorInScene[a]); it.ok(); ++it)
  position.add(slot[*it]);
IloIntExpr firstSlot = IloMin(position);
IloIntExpr lastSlot = IloMax(position);
```

This code iterates over all scenes which the current actor plays in, and adds the appropriate slot variables for those scenes to an array of variables `position`. Then, two expressions are built: the first slot that the actor plays in is the minimum of these slot variables and the last slot that he plays in is their maximum.

`firstSlot` and `lastSlot` define when the actor is on set; he will be present for every scene filmed between `firstSlot` and `lastSlot` inclusively.

You are now ready to build the part of the model which will evaluate the waiting time of the current actor *a* in the loop.

Step 3 Create the waiting time for an actor

Insert the following code after the comment `// If an actor is not in a scene,`

```
// he waits if he is on set when the scene is filmed
```

```
for (IloInt s = 0; s < numScenes; s++) {
  if (!actorInScene[a].contains(s)) { // not in scene
    IloIntExpr wait = (firstSlot <= slot[s] && slot[s] <= lastSlot);
    actorWait += sceneDuration[s] * wait;
  }
}
```

Here, you loop over all scenes. For any scene in which the actor does not appear, you add a waiting time if and only if the slot in which the scene is filmed is a slot for which the actor is on set (in other words, a slot between the arrival and departure of the actor). This is done by creating a 0-1 expression from the Boolean condition of being on set. This 0-1 expression is then multiplied by the filming duration of the scene in question to arrive at the correct waiting time for that scene. Notice that the waiting time for the scene is either 0 or the filming duration.

The remainder of the code of the `BuildModel` function which is provided for you works out the waiting cost of the actor to be his waiting time multiplied by his pay rate, and accumulates this cost. Then the total waiting cost is constrained to be equal to the cost variable passed as parameter, and the model is returned.

Problem solving: Finding an initial solution

Now you create an initial solution to the problem which will serve as a starting point for the LNS which will follow. Before you do this, you need to create an instance of `IloSolver` which will be used for search.

Step 4 Create a solver

Insert the following code after the comment

```
// Create the solver, and seed its random number generator
```

```
IloSolver solver(model);
solver.getRandom().reSeed(seed);
```

This code creates the solver from the model and afterwards sets this solver's random number seed to that specified on the command line. In this way, different seeds can be specified to observe different search trajectories.

You can now create the initial solution.

Step 5 Generate an initial solution

Insert the following code after the comment

```
// Create an initial solution to the problem
```

```
IloGoal goal = IloGenerate(env, scene);
solver.solve(goal);
```

Here, you use the basic `IloGenerate` goal on the `scene` variables to produce the initial solution.

To perform any local search using Solver IIM, you need to have an instance of an `IloSolution` object which represents the current solution. You will now create this solution object.

Step 6 Create the solution object

Insert the following code after the comment `// Create the solution object`

```
IloSolution solution(env);
solution.add(scene);
IloObjective objective = IloMinimize(env, idleCost);
solution.add(objective);
solution.store(solver);
```

Here you add the decision variables `scene` to the solution object, as well as the objective which is to minimize total idle time. Addition of the objective allows improvement search to compare the current solution with the new proposed solution. Finally, you store the current values of the decision variables in the solution via `solution.store(solver)`.

Problem solving: Large neighborhood search

As outlined in the introduction, there are two essential components to LNS. The first is the choice of "fragment"; the part of the solution which can change its value. The second is the completion method, which instantiates the fragment to some new value. Solver IIM uses goals for the latter task (instances of `IloGoal`), and for the former uses neighborhoods (instances of `IloNHood`), although the neighborhoods are quite particular in nature.

Normally, this neighborhood has only one neighbor which might be termed a "meta-neighbor" as it defines the fragment and thus the search space for the completion goal. There is a special subclass of `IloNHoodI`, `IloLargeNHoodI` which is used to define large neighborhoods; full details can be found in the *IBM ILOG Solver Reference Manual*. However, for now you will not need to understand any details, or even how to subclass `IloLargeNHoodI`. A simple interface is provided via the macro `ILODEFINELNSFRAGMENT1` (and its variants) which you will use here to define an LNS fragment for the talent scheduling problem.

You will now create the instance of `IloNHood` which defines a fragment.

Step 7 Write the fragment-defining neighborhood

Insert the following code after the comment

```
// Define an LNS fragment which is a random segment
```

```
ILODEFINELNSFRAGMENT1(SegmentLNSNHood, solver, IloIntVarArray, x) {
  IlcRandom r = solver.getRandom();
  IloInt a = r.getInt(x.getSize());
  IloInt b = r.getInt(x.getSize());
  if (a > b) { IloInt tmp = a; a = b; b = tmp; }
  for (IlcInt i = a; i <= b; i++)
    addToFragment(solver, x[i]);
}
```

This code chooses as a fragment a random segment in the schedule. That is, you will authorize changes to variables from indices `a` to `b` where `a` and `b` are randomly chosen. It is *essential* that the fragment is chosen with some random element. If not, the same fragment will be chosen time and time again, and the search will stagnate very quickly with no improvements possible. In the macro `ILODEFINELNSFRAGMENT`, you must call the service function `addToFragment` for every variable that you wish to include in the part of the solution that can change its value. The macro takes only two mandatory parameters: the name of the neighborhood function to create, and the name of an instance of `IloSolver` which will be received. Depending on the particular instance of the macro chosen, optional parameters may then be added in the usual way by pairs of type and variable name. The macro also has other services available like retrieving the current solution: see the documentation of `IloLargeNHoodI` in the *IBM ILOG Solver Reference Manual*.

In this fragment, two indices are chosen randomly using the solver's random number generator. These indices are interchanged to make sure that `a` is no greater than `b`. Finally, all variables from `a` to `b` are added to the fragment.

Now that you have created the fragment-defining neighborhood, you can move on to the large neighborhood search itself, which is essentially the same a performing a standard local search. You begin by creating the large neighborhood.

Step 8 Create the neighborhood

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

```
IloNHood nhood = SegmentLNSNHood(env, scene);
```

This code creates the large neighborhood to be used in the search. This takes care of the first part of LNS: the definition of a fragment. The second part is the completion method. For this (and as is common in LNS), you will use the initial solution goal to complete the search. This goal will, however, be limited in search capacity so as to avoid long search times in fragments which may have no improving solutions. Experience has shown that this

limitation is essential to LNS, although the best limit can vary by problem and quality of the heuristic used in the completion goal (good heuristics are likely to require less search in the completion goal than poorer ones).

Step 9 Create the completion goal

Add the following code after the comment

```
// Create the LNS completion goal
```

```
IloGoal complete = IloLimitSearch(env, goal, IloFailLimit(env, 100));
```

As LNS can explore a much larger neighborhood than is typical of standard neighborhood searches, normally there is less reliance on a metaheuristic to guide the search, and as such greedy search is often employed. That is the technique that you will use here.

Step 10 Create the single move goal

You will now create the local search move goal by inserting the following code after the comment `// Create a greedy local search move goal`

```
IloGoal move = IloSingleMove(
  env, solution, nhood, IloImprove(env), complete
);
```

Next, you perform the local search loop. This is just as for other greedy local searches, except that the stopping condition is altered. As the fragment is generated randomly, a failed solve is not an indication that the search should stop, as success depends on the particular randomized fragment generated. Thus, the search is stopped only after a certain number of tries are made without success. The total number of choice points used is totalled to give an indication of total search effort. The following code is provided for you:

```
// Enter the local search loop
IloInt choicePoints = 0;
IloInt noMove = 0;
IloInt movesTried = 0;
while (noMove < 100) {
  movesTried++;
  if (solver.solve(move)) {
    noMove = 0;
    cout << movesTried << ": Cost = "
         << solver.getValue(idleCost) << endl;
  }
  else
    noMove++;
  choicePoints += solver.getNumberOfChoicePoints();
}
cout << "Total choice points = " << choicePoints << endl;
```

Displaying the solution

Having terminated the LNS, it is now time to display the best solution found. To do this, you will first restore the best solution so that solver's variables are instantiated in line with them. Then, a simple loop displays the order of the scenes.

Step 11 Display the scene filming order

Insert the following display code after the comment

```
// Display the order of scene filming
```

```
solver.solve(IloRestoreSolution(env, solution));
cout << "Solution of idle cost " << solver.getValue(idleCost) << endl;
cout << "Order:";
for (IloInt s = 0; s < numScenes; s++)
  cout << " " << 1 + solver.getValue(scene[s]);
cout << endl;
```

A more complicated display is provided for you which displays the time units at which each actor acts. The number of units of waiting time can be more clearly seen for each actor in this case. The code is given below:

```
// Give more detailed information on the schedule
for (IloInt a = 0; a < numActors; a++) {
  cout << "|";
  for (IloInt s = 0; s < numScenes; s++) {
    IloInt sc = solver.getIntValue(scene[s]);
    for (IloInt d = 0; d < sceneDuration[sc]; d++) {
      if (actorInScene[a].contains(sc))
        cout << "X";
      else
        cout << ".";
    }
    cout << "|";
  }
  cout << "  (Rate = " << actorPay[a] << ")" << endl;
}
```

Complete program

The complete program follows and can be viewed online in `YourSolverHome\examples\src\lstalent.cpp`:

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

```
#include <ilsolver/iim.h>

ILOSTLBEGIN
// Read talent scheduling problem data from a file
IloBool ReadData(const char * filename,
                 IloIntArray actorPay,
                 IloIntArray sceneDuration,
                 IloArray<IloIntSet> actorInScene) {
  IloEnv env = actorPay.getEnv();
  ifstream in(filename);
  if (!in.good())
    return IloFalse;

  IloInt numActors, numScenes, a, s;

  in >> numActors;
  for (a = 0; a < numActors; a++) {
    IloInt pay;
    in >> pay;
    actorPay.add(pay);
  }

  in >> numScenes;
  for (s = 0; s < numScenes; s++) {
    IloInt duration;
    in >> duration;
    sceneDuration.add(duration);
  }

  for (a = 0; a < numActors; a++) {
    actorInScene.add(IloIntSet(env));
    for (s = 0; s < numScenes; s++) {
      IloBool inScene;
      in >> inScene;
      if (inScene)
        actorInScene[a].add(s);
    }
  }
  if (!in.good())
    return IloFalse;

  in.close();
  return IloTrue;
}

// Build the talent scheduling model
IloModel BuildModel(IloIntVarArray scene,
                    IloIntVar idleCost,
                    IloIntArray actorCost,
                    IloIntArray sceneDuration,
                    IloArray<IloIntSet> actorInScene) {
  IloEnv env = scene.getEnv();
  IloInt numScenes = scene.getSize();
  IloInt numActors = actorCost.getSize();
  IloModel model(env);

  // Make the slot-based secondary model
  IloIntVarArray slot(env, numScenes, 0, numScenes - 1);
```

```
  model.add(IloInverse(env, scene, slot));

  // Expression representing the global cost
  IloIntExpr cost(env);

  // Loop over all actors, building cost
  for (IloInt a = 0; a < numActors; a++) {
    // Expression for the waiting time for this actor
    IloIntExpr actorWait(env);

    // Calculate the first and last slots where this actor plays
    IloIntVarArray position(env);
    for (IloIntSet::Iterator it(actorInScene[a]); it.ok(); ++it)
      position.add(slot[*it]);
    IloIntExpr firstSlot = IloMin(position);
    IloIntExpr lastSlot = IloMax(position);

    // If an actor is not in a scene,
    // he waits if he is on set when the scene is filmed
    for (IloInt s = 0; s < numScenes; s++) {
      if (!actorInScene[a].contains(s)) { // not in scene
        IloIntExpr wait = (firstSlot <= slot[s] && slot[s] <= lastSlot);
        actorWait += sceneDuration[s] * wait;
      }
    }

    // Accumulate the cost of waiting time for this actor
    cost += actorCost[a] * actorWait;
  }
  model.add(idleCost == cost);
  return model;
}

// Define an LNS fragment which is a random segment
ILODEFINELNSFRAGMENT1(SegmentLNSNHood, solver, IloIntVarArray, x) {
  IlcRandom r = solver.getRandom();
  IloInt a = r.getInt(x.getSize());
  IloInt b = r.getInt(x.getSize());
  if (a > b) { IloInt tmp = a; a = b; b = tmp; }
  for (IlcInt i = a; i <= b; i++)
    addToFragment(solver, x[i]);
}

int main(int argc, const char * argv[]) {
  IloEnv env;
  try {
    const char * inputFile = "../../../examples/data/rehearsal.txt";
    IloInt seed = 1;
    if (argc > 1)
      inputFile = argv[1];
    if (argc > 2)
      seed = atoi(argv[2]);

    IloIntArray actorPay(env);
    IloIntArray sceneDuration(env);
    IloArray<IloIntSet> actorInScene(env);
    IloBool ok = ReadData(inputFile, actorPay, sceneDuration, actorInScene);
    if (!ok) {
```

```
    env.out() << "Error reading " << inputFile << endl;
  }
  else {
    // Create the decision variables, cost, and the model
    IloInt numScenes = sceneDuration.getSize();
    IloInt numActors = actorPay.getSize();
    IloIntVarArray scene(env, numScenes, 0, numScenes - 1);
    IloIntVar idleCost(env, 0, IloIntMax);
    IloModel model = BuildModel(
      scene, idleCost, actorPay, sceneDuration, actorInScene
    );

    // Create the solver, and seed its random number generator
    IloSolver solver(model);
    solver.getRandom().reSeed(seed);

    // Create an initial solution to the problem
    IloGoal goal = IloGenerate(env, scene);
    solver.solve(goal);

    // Create the solution object
    IloSolution solution(env);
    solution.add(scene);
    IloObjective objective = IloMinimize(env, idleCost);
    solution.add(objective);
    solution.store(solver);

    // Create the LNS neighborhood
    IloNHood nhood = SegmentLNSNHood(env, scene);

    // Create the LNS completion goal
    IloGoal complete = IloLimitSearch(env, goal, IloFailLimit(env, 100));

    // Create a greedy local search move goal
    IloGoal move = IloSingleMove(
      env, solution, nhood, IloImprove(env), complete
    );

    // Enter the local search loop
    IloInt choicePoints = 0;
    IloInt noMove = 0;
    IloInt movesTried = 0;
    while (noMove < 100) {
      movesTried++;
      if (solver.solve(move)) {
        noMove = 0;
        cout << movesTried << ": Cost = "
             << solver.getValue(idleCost) << endl;
      }
      else
        noMove++;
      choicePoints += solver.getNumberOfChoicePoints();
    }
    cout << "Total choice points = " << choicePoints << endl;

    // Display the order of scene filming
    solver.solve(IloRestoreSolution(env, solution));
    cout << "Solution of idle cost " << solver.getValue(idleCost) << endl;
```

```
      cout << "Order:";
      for (IloInt s = 0; s < numScenes; s++)
        cout << " " << 1 + solver.getValue(scene[s]);
      cout << endl;

      // Give more detailed information on the schedule
      for (IloInt a = 0; a < numActors; a++) {
        cout << "|";
        for (IloInt s = 0; s < numScenes; s++) {
          IloInt sc = solver.getIntValue(scene[s]);
          for (IloInt d = 0; d < sceneDuration[sc]; d++) {
            if (actorInScene[a].contains(sc))
              cout << "X";
            else
              cout << ".";
          }
          cout << "|";
        }
        cout << "  (Rate = " << actorPay[a] << ")" << endl;
      }
      solver.end();
    }
  } catch (IloException & ex) {
    cout << "Caught: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Output

Execution of the program results in the following output:

```
2: Cost = 45
3: Cost = 40
5: Cost = 37
6: Cost = 36
8: Cost = 34
10: Cost = 33
22: Cost = 30
24: Cost = 23
29: Cost = 19
30: Cost = 17
Total choice points = 1807
Solution of idle cost 17
Order: 3 8 2 7 1 5 6 4 9
|.|.......|XXXX|XXXXX|XX|...|XX|XXX|XXXXXX|  (Rate = 1)
|.|XXXXXXX|XXXX|.....|XX|XXX|XX|XXX|......|  (Rate = 1)
|.|XXXXXXX|XXXX|XXXXX|XX|...|..|...|......|  (Rate = 1)
|.|.......|....|.....|XX|XXX|XX|...|XXXXXX|  (Rate = 1)
|X|XXXXXXX|....|XXXXX|..|XXX|XX|...|......|  (Rate = 1)
```

About this problem

An excellent overview of this problem can be found in Barbara Smith's tutorial paper: "Constraint Programming in Practice: Scheduling a Rehearsal", research report APES-67-2003. This problem is also present in the Constraint Programming library (`http://www.csplib.org`) as problem number 39: *the rehearsal problem.* Here you may find the problem data supplied with Solver as well as references to Smith's tutorial and other information.

The original problem originated from Lancaster University in the 1970s where the objective was to minimize the waiting time of players during an evening rehearsal of several musical pieces. The application to the *talent scheduling problem* where each actor (or player) has a different rate of pay was described in T. C. E. Cheng, J. Diamond, and B. M. T. Lin: "Optimal scheduling in film production to minimize talent hold cost". *Journal of Optimization Theory and Applications*. 79:197-206, 1993.

Tips on using Large Neighborhood Search

The two most important aspects of successfully applying LNS to a new problem are the choices of the *completion goal* and of the method for generation of the LNS *fragment.* The completion goal has less complications, and so it shall be described first.

The completion goal

◆ **Use the first solution goal for completion.**

The simplest way to use a completion goal in LNS is simply to use the initial solution goal as a completion goal. The idea here is that if the initial solution goal is to considered as a goal which is well adapted to providing good solutions to the problem as whole, then we can also suppose that it will produce good solutions to problem fragments.

◆ **Limit the completion goal.**

One disadvantage of using the initial solution goal "as is" to complete the fragment is that if the fragment is large, the completion goal can take a long time to search the entire search space of the fragment. This exponential search time is what you would like to avoid in using a local search method. So, in practice any completion goal used in an LNS context is limited (normally by a fail limit imposed using `IloFailLimit`). This allows LNS to examine many more fragments in the same time frame, avoiding becoming bogged down in very long searches of fragments which have no improving solutions.

◆ **Use a good heuristic.**

When you use pure tree search to find solutions to a constraint programming problem, normally you try to find a good *heuristic* for the problem, that is a variable and value selection rule which will tend to lead to better solutions. This is also true for the completion goal, but will come for free if you use the same goal to complete the fragment as you do to find the initial solution.

A good heuristic is also particularly important in fragment completion as the search goal will be limited (see above) and so will have much less explorative power that a complete goal. As such, it is useful if the search can find good solutions without making many mistakes *i.e.* a good heuristic is desirable. That also means that for better heuristics, you can probably make do with a lower fail limit in the completion goal, and conversely for a poor heuristic, the fail limit will need to be higher, and search time per iteration will increase.

◆ **Use randomization.**

If you don't have an idea of what a good heuristic may be, or it seems too difficult to come up with a good goal, one technique which can increase robustness of the completion goal is to add some randomization to the completion goal (see for example `IloRandomPerturbation` in the *IBM ILOG Solver Reference Manual*). This will distribute the assignments made more evenly between the available values, meaning that the completion goal will not produce fragment completions in the same vein each time, but will vary them.

The choice of fragment

◆ **Randomization.**

The basic rule is that fragments should be chosen with some random element. Doing so will mean an large variety of fragments. This does not mean, however, that fragments should be chosen completely at random. Often, from looking at the problem, you can determine which types of fragments are likely to be better for finding better solutions. You can then bias your randomized process towards certain types of fragment.

◆ **Fragment size.**

One important question is: "How large should a fragment be?" The advantages of a small fragment are that the fragment can be searched more quickly and it is more likely that the completion goal will find an improving solution in a smaller fragment if one exists. The advantage of a larger fragment is that it is more likely that an improving solution does indeed exist in the fragment.

The ideal size will depend on the problem and on the stage of the search. For example, if there are many improving solutions, a small fragment can suffice as there will be a good probability that even small fragments will contain improving solutions. As search progresses and improving solutions become more scarce, then a larger fragment size may be needed. However, this must be traded with the fact that for larger fragments, the chance of finding any improving solution will decrease with the size of the fragment. So, ideally the fragment

should be just large enough to have a reasonable chance of containing an improving solution, and no larger.

Some more complex schemes try to learn the best size of fragment by increasing it after a certain number of iterations with no improvement, and decreasing it if improving solutions are being found easily. However, one very simple technique which removes the burden of choice is to choose the size of the fragment randomly (perhaps in some prescribed range) each time a fragment is generated. Certainly, this approach is not optimal, and it could be (or will be) that a good deal of fragments are generated which are too large or too small. However, equally true is that a good deal of fragments will be generated which are in the acceptable range of sizes, and so provide good candidates for finding improved solutions. This simple technique has the advantage of being simple and robust to changes in problem data which may call for larger or smaller fragments than is typical.

◆ **Random choice.**

One simple way to decide what to place in the fragment is to decide completely at random. That is, each variable has the same probability of being added to the fragment as any other variable. You can try this method for the lesson that you have just completed (add each `scene` variable with probability 0.5, for example). You should see that although it does not perform as well as the method given, it does not perform very badly. Thus, it can be admissible, in the absence of any other good ideas, to generate the fragment completely at random. However, you will see below how you can do better when the problem admits some structure.

◆ **Good and bad parts.**

Quite often, for a given solution, it is possible to identify “good” and “bad” parts of the solution. For example, in a technician dispatching application, a measure of quality of the schedule of a technician might be the total time spent working at a customer sites divided by the total travelling time between customers. In this case, you would want to favor the inclusion of the poorest parts of the solution in the fragment. This does not mean that the “best” parts of the solution should never be added to the fragment; indeed their inclusion may be necessary (with poorer parts) in order to produce a better solution mixing the elements of both parts. The idea is merely that the poorer parts of the solution should be easier to optimize and thus a bias to include them in the fragment with higher probability should lead to an improved solution more quickly.

◆ **Related parts of solutions.**

The previous point gave a criterion that can be used for preferring to include certain variables in the fragment (those involved in “bad” parts of the current solution). However, often a judgment about what variables are desirable to add to the fragment depends upon the variables which have already been added. The basic idea is that the set of variables added to the fragment should have some interdependence. Thus, when a variable in the fragment changes its value from that in the current solution, if we are to respect problem constraints and/or improve the objective, then some other part of the fragment will need to change value

too. If this property holds, the fragment is said to be *compact*. If the aforementioned inter-dependence were not present, then a smaller fragment could have been chosen, leaving out the parts which have little or no relationship to all the others as even if they were in the fragment, they are unlikely to need to change value.

As an example, suppose you have a selection of objects of different sizes, and you need to decide which subset of those objects should go in the container with the objective of minimizing the amount of free space left in the container. This problem can be modeled by a 0-1 variable for each object indicating whether that object is in the container or not. For most of the search, solutions found will have very little free space (except perhaps at the beginning, when it is easy to produce a better solution). Now, clearly it is absurd to relax only variables which have value 1 (objects in the container) or of value 0 (objects not in the container). In the former case, the only option would be to reduce the number of objects in the container by setting variables formerly at 1 to 0. This must make the solution worse. In the latter case, the only option is to add objects to the container (without removing others) by setting variables formerly at 0 to 1. However, this is unlikely to be possible as we suppose that for most of the search the free space in the container is limited. After consideration, you can see that you must have both variables of value 0 and of value 1 in the fragment so that some objects can be removed from the container and others added in their place. This leads to a simple rule which says that when the fragment already contains a surplus of variables at one value, the probability of adding a new variable with the same value should be diminished.

In the sequel, some different possibilities for the structure of the fragment depending on the problem type are described.

- **Sequence-based problems.**

Sequence-based problems, like the one presented in this chapter, or the car sequencing problem described in Chapter 6, *Using the Distribute Constraint: Car Sequencing*, often have strong interactions (in terms of constraints or contributions to the objective function) between neighbors in the sequence. Therefore, often in this case a good fragment is one which involves a contiguous segment of the sequence (as was used in this lesson), as this allows the greatest flexibility for changing contiguous cells in the sequence.

- **Geographic problems.**

In geographical problems (where you have a notion of some distance between objects), often one has to minimize some total distance traveled between objects in the solution as one makes single or multiple “tours”. In this case, poorer parts of the solution can be seen as objects which are visited consecutively but are far apart. As far as related objects go, a typical method of producing a compact fragment is to add objects to the fragment which are close together. This rationale comes from the observation that if one swaps the positions of two distance objects in some route (or two distinct routes), this will increase the total distance considerably in any reasonable solution.

- **Row/column problems.**

Problems such as time tabling can have a row- and column-based structure. These problems invariably have constraints on the rows as well as on the columns. Suppose that for such a problem each cell in the matrix corresponds to a variable. Such problems benefit from fragments which include variables from the same column or the same row, and so effective fragment generation methods often include such bias. This can be done by weighting probabilities so that rows and columns already mentioned in the fragment are favored. Another method is to choose a smaller (random) number of rows and columns and then select variables only from these. This ensures a spread of the fragment over only a few rows and columns.

- **Scheduling problems.**

Scheduling problems are best treated via the LNS support in IBM® ILOG® Scheduler. However, to give a flavor of the type of fragment which is useful, one can observe that scheduling is both time and resource constrained. This would indicate that activities forming a compact fragment would start at approximately the same time, or require the same resource or resources. Thus, an effective technique can be to add all variables corresponding to activities lying in a certain time window or on a certain resource. The logic is very similar to that of row/column problems where a resource can be likened to a row and a time window to a column.

- **Packing and resource allocation problems.**

In packing and resource allocation problems, there is a limited resource and tasks or objects need to be allocated to resources to minimize resource number or cost. This is simply a more general form of the “container” example presented earlier. Poorer parts of the solution can usually be identified as resources which are bad value for money, for example which are costly and/or under-utilized. This provides one criterion for including variables in the fragment. Another criterion based on how parts of the solution are related may also be employed in order to create a more compact fragment. For example, if the goal is to reduce the number of resources used, then from time to time *all* tasks allocated to a particular resource must be added to the fragment in order to give a chance of liberating the resource in question. This technique can also be useful for combining the tasks allocated on three resources onto only two, for example.

Making the most of neighborhood search

LNS is implemented as a neighborhood search technique in Solver IIM. In this way, you can make the most of the facilities provided by neighborhood search to improve the way in which you use LNS. For example, LNS is a more computationally intensive technique than more traditional neighborhood searches, but does have the advantage that it is less likely to become stuck in local optima. One way to increase the efficiency of your search is to run a typical neighborhood search (using, depending on your problem, `IloFlip`, `IloSwap`, and so on) before switching to LNS. This can pay dividends on larger problem instances.

Another way in which neighborhood search can be used is via the application of metaheuristics. In this lesson, only greedy search was shown, but tabu search and simulated annealing apply equally well to LNS. You should, however, recognize that such searches will require longer running times.

Finally, some problems have a dual aspect, for instance multiple or hierarchical objectives, or strongly competing different constraint types. In this case, you might envisage two or more different types of large neighborhood search fragment that you can generate, each one dedicated to working on a different aspect of the problem. You could create one single fragment definition neighborhood which performs one of the fragment types at random, but a more modular approach is to use neighborhood addition as below. In this way, all types of fragment generation methods can be used which keeping their codes separate.

```
// Fragment generators
ILODEFINELNSFRAGMENT1(GenFrag1, solver, IloIntVarArray, x) {
  // ... code of generator 1 ...
}

ILODEFINELNSFRAGMENT1(GenFrag2, solver, IloIntVarArray, x) {
  // ... code of generator 2 ...
}

ILODEFINELNSFRAGMENT1(GenFrag3, solver, IloIntVarArray, x) {
  // ... code of generator 3 ...
}

int main() {
  // ...

  // Decision Variables.
  IloIntVarArray x(env, size);
  // ...

  // ...
  IloNHood nhood = GenFrag1(env, x) + GenFrag2(env, x) + GenFrag3(env, x);
  // ...
}
```

Part VI

Evolutionary Algorithms

This part consists of the following lessons:

- Chapter 28, *Introduction and Basic Concepts*
- Chapter 29, *Modeling and Solving a Basic EA Problem*
- Chapter 30, *Using More Advanced EA Features*
- Chapter 31, *Bin Packing Using EA*
- Chapter 32, *Car Sequencing Using EA*

C H A P T E R

28

Introduction and Basic Concepts

This part of the *IBM ILOG Solver User's Manual* describes the IBM® ILOG® Solver EA (Evolutionary Algorithm) Framework, part of IBM ILOG Solver IIM (Iterative Improvement Methods). This part will give you an introduction to the ideas behind Evolutionary Algorithms and will present the EA object support available via examples. The examples are presented as working lessons that move from a code skeleton to a finished program that you compile and execute.

This documentation does not provide an introduction to evolutionary or genetic algorithms in general, and it is assumed that the reader is familiar with the basic genetic algorithm concepts. An understanding of simple genetic algorithm operations such as mutation and combination (crossover) is assumed, although detailed knowledge of the construction of genetic algorithms is not assumed.

In this lesson, certain sections are labeled "Advanced Topic." These are topics which can be skipped on first reading.

Overview of concepts

Evolutionary Algorithms (EA) are optimization algorithms whose design has been influenced by the observation of the adaptation of living organisms. This covers rather a broad range of algorithms including those which are influenced by Darwinian evolution, the food-finding abilities of ants and the behavior of immune systems. Such algorithms typically

deal with a “population” of solutions and use adaptation, competition and communication to produce better solutions.

The class of algorithms known as Genetic Algorithms (GAs) more specifically simulates the processes of natural selection, genetic mutation and genetic combination. A GA starts with a population of “individuals” (which is a set of solutions to the problem), then continually selects, mutates, breeds, and replaces members of the population. How the GA carries out these processes depends on the type of GA and the qualities of the individuals. A GA must maintain a balance of diversity and quality in the population. Concentration on diversity at the expense of quality will result in diverse, but generally poor solutions, as exploitation of good solution features is too weak. The inverse can result in the population *converging* too quickly, usually well before the solutions can be sufficiently optimized. (Convergence is the term used to mean that all individuals in the population represent more or less the same solution; from this point there is little hope of further evolution.)

At the present time, the IBM® ILOG® Solver EA framework is primarily geared towards facilitating the implementation of GAs. The genetic operators provided in the framework work *with* the constraint programming provided by Solver. That is, genetic operators make use of the constraint model and constraint propagation to avoid generation of illegal solutions. A mutation or a combination operator is a constraint programming search for a new solution that encodes a maximal amount of genetic material from the parent(s) while respecting the problem constraints.

If you are familiar with genetic algorithms, you may notice that some of the concepts and class names used do not directly follow the parlance of the genetic algorithm community. Normally, the class names used are more abstract in keeping with the notion of generality of the library. For instance, “solution pools” are general objects which serve as holders for the population, offspring, and any solutions selected for replacement. Also, the concept of a “pool processor” encompasses objects that perform selection, mutation, combination, and replacement. These naming conventions are largely superficial and will become clearer in the forthcoming sections.

Genetic algorithms and constraint programming

The use of Solver as a basis for the implementation of genetic algorithms offers advantages over what may be considered a more virgin approach. Constraint programming enhances the creation of genetic algorithms in a variety of ways, including, but not limited to:

- Guarantee of feasibility

  In traditional GA systems, genetic operators such as mutation or combination are often quite blind to the constraints of the problem. This can lead to many generated solutions being illegal with respect to the problem constraints. This is normally circumvented in one of two ways. First, by using an *indirect* representation. (This will be fully discussed in the next section.) Second, problem constraints are relaxed and their violation penalized.

In contrast to the second method, the use of constraint programming means that constraints have a central role to play and need not be avoided or softened; the constraint programming engine ensures that constraints will not be violated by new solutions that are produced.

◆ Initial population generation

Generating an initial population of solutions is an essential part of genetic algorithms. Without constraint programming, often one has to begin with an entirely random population, perhaps softening important constraints in the process.

Using constraint programming, the generation of initial solutions can be performed by the execution of straight-forward constraint programming goals, just as in the standard Solver examples. Goals can be executed repeatedly, each time being differently randomized to result in a population which respects problem constraints. Quality can be traded for diversity by adjusting the amount of randomization applied to the goal. This topic will be more fully described later.

◆ Flexibility

Constraint programming carries with it inherent flexibility in the type of operations that can be carried out. One example is that by simple addition of a constraint, it can be stipulated that any new solution generated must be better in quality than the worst member of the population. Such a constraint will be actively used in the genetic operator to restrict the set of solutions examined to those satisfying the aforementioned property. Another example of the flexibility fostered by constraint programming is the ability to mix what may be termed “pure” constraint programming (tree search) with genetic operators. For example, one might fill 3/4 of a new solution with information from parents, allowing the missing 1/4 to be supplied using a more standard tree search. Such techniques typically work well in practice.

Representation

The type of representation chosen for the solution to a problem can have a great effect both on the genetic operators used and the overall efficiency of the resulting genetic algorithm. Representations fall into two broad categories: direct and indirect. A *direct* representation is the one that you are most used to dealing with in Solver; it might be referred to as the “natural” representation, and has the benefit that it is easy to evaluate. An *indirect* representation is one which sits alongside the direct representation and is not complete in itself. In order to evaluate an instantiation in the indirect representation, a *decoder* or mapping function is invoked which transforms the solution in the indirect representation into one in the direct representation. In the EA framework, the way of doing this is with a constraint programming goal which uses the values of variables in the indirect representation to influence its instantiation of the direct representation. Note that in general there is no one-to-one correspondence between solutions in the two representations. Many solutions in the indirect representation may decode to the same solution in the direct representation. Equally,

some directly represented solutions may be impossible to obtain as there may be no solution in the indirect representation which maps to them.

An example of an indirect representation of scheduling is a unique priority for each activity. The description of a possible decoding goal could then be stated as follows. The decoder schedules activities one by one until all activities are scheduled. At each step, among the activities which can be scheduled, the decoder chooses the one with highest priority. This activity is then placed as early as possible in the schedule. Importantly, using constraint programming, this last decision can be backtracked in the case of a failure, and alternative assignments tried. This, combined with constraint propagation, makes decoders written using constraint programming more robust. This will be elaborated on below.

Figures below show the use of a direct and indirect representation in the EA framework.

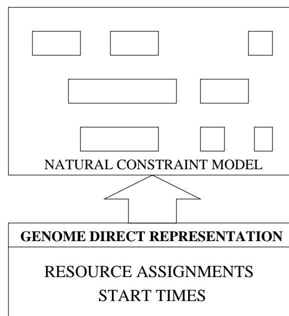

Figure 28.1 *Direct Representation*

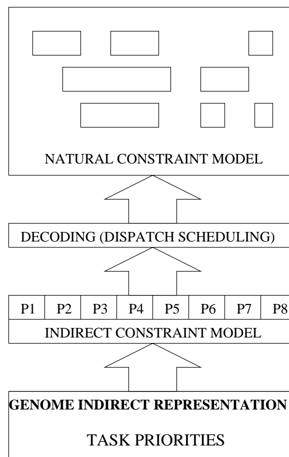

Figure 28.2 *Indirect Representation*

Given its additional complexity, what are the advantages of the indirect representation? First, the indirect representation is normally free of constraints, or has constraints which are easy to satisfy. For example, there may be only “range” constraints on a set of priorities for activities. Thus, matters can be arranged such that any genetic operators applied to a solution in the indirect representation result in a new valid solution. This moves the burden of maintaining feasibility of the direct representation to the decoder. The decoder is then responsible for interpreting the indirect representation, and instantiating the direct representation such that the constraints therein (that is, the original problem constraints) are respected.

At this point an additional question might be posed. Does the indirect representation have any merit when constraint programming is being used? After all, one of the benefits is that constraints in the direct representation are handled directly. This is true, but depending on the problem being addressed, the performance of genetic operators on the direct representation can vary. The problem of finding a solution for a mutation or combination operator on the direct representation is combinatorial in nature. The problem of decoding the indirect representation under constraints is also combinatorial in nature, but importantly is normally easier in practice. Moreover, constraint programming delivers the advantage over traditional methods that additional constraints can be added to the problem without changing the decoder. Constraint propagation and the ability to backtrack bad decisions greatly increase the robustness of decoding.

So, when should you use a direct representation, and when should you use an indirect one? This is not always the easiest question to answer, but some general advice can be given. There are two main reasons for using an indirect representation. The first one is that the problem is *strongly coupled*. What do we mean by this? One way to think about it is by asking yourself the following question. If I have a solution to my problem (in the direct/natural representation), and I change a small part of this solution, how much of the rest of the solution would I need to change to stay compatible with the initial change? If the answer is “a lot,” the problem is strongly coupled. Often the best way to imagine the initial change is as one which needs to be made because a machine breaks down, a time slot becomes unavailable, a road is closed, a delivery is delayed, and so on. You may then ask, “How much of my solution will this change affect?” For example, imagine a scheduling problem where the objective is to minimize the total length of the schedule. Good solutions to such problems tend to be tightly packed and have little “slack.” In these types of problems, if some input (say some raw material) arrives late, then this can have huge implications on the schedule; there is seldom a “small fix” which will maintain feasibility. Such problems can be so tightly coupled that small mutations on a direct representation have little hope of working. Even when using constraint programming, the closest legal schedule may be far from what is proposed by the genetic operator, and thus may take an unacceptably long time to find. Contrast this with a schedule where the objective is to minimize labor costs. Such a problem tends to be much more loosely coupled. In this case, if some goods arrive late, then we can normally change the schedule slightly to account for this, for instance by authorizing overtime for a period until the schedule is back on track. The cost will be higher than in the original schedule, but importantly, the schedule only had to be slightly modified. Here, small

mutations of the schedule have a much higher probability of resulting in a legal schedule, even if at increased cost.

The second reason why an indirect representation can be preferred to a direct one is probably even more important than the first. Indirect representations are often much more standardized than direct ones. Direct representations are necessarily specific to the problem class, whereas indirect ones tend to center around themes. A common theme is to give a priority to each assignable object in the problem. A decoder then decides what to do with these objects *in priority order*. The use of standardized indirect representations allow simple (and often built-in) genetic operators to be used. This still means that a decoder needs to be written for the problem class at hand, but this is usually relatively simple. With direct representations on the other hand, genetic operators usually need to be written for the problem class being addressed; normally a much more sophisticated task.

It should be mentioned that with good design of genetic operators made for the problem class at hand, direct representations can be effective even for problems which are quite strongly coupled. Performance in such cases is often significantly better than that of indirect representations as no decoding phase is present. However, the design of hand-crafted operators can be a difficult and painstaking endeavor, and as such, we recommend using an indirect representation as a first step either in the case where no built-in operators match the direct representation of the problem, or where the problem is strongly coupled.

Features of the library

This section outlines the features of the IBM® ILOG® Solver EA Framework.

Solutions and solution pools

When Solver is used in the standard manner, via tree search, little consideration is given to multiple solutions unless these are the progression of improving solutions found during a tree search process. Since these solutions do not need to co-exist (you are normally only interested in the best solution found), then the use of a *solution object* is not normally necessary; the state of Solver's constrained variables provide a solution without the need for additional information.

By contrast, solution objects have a central and fundamental role when engaging in genetic or evolutionary search processes as a number of solutions have to co-exist. Thus, in the EA Framework, extensive use is made of the `IloSolution` class (see the *IBM ILOG Solver Reference Manual*) which can represent a solution in its entirety. Objects of type `IloSolution` form the basis of communication between the basic elements of the framework.

In order to make it easier for you to manipulate populations, or groups of solutions, the class `IloSolutionPool` is provided. This class is nothing more than a bag of solutions, although

some additional facilities make it easier for you to get the best and worst members, sort the pool and iterate over it. Objects of type `IloSolutionPool` are the principal form of communication between the various types of genetic operators available. Typically, a genetic operator will receive a solution pool, which in traditional genetic algorithm terminology might be termed the *parents*, and will generate another pool of solutions that would traditionally be termed the *offspring*. Despite the fact that the `IloSolutionPool` class plays all roles (as well as that of the role of the population), it is quite normal to use the terms “population,” “parents,” and “offspring” depending on the exact role of the pool.

The following code shows a simple manipulation of a solution pool:

```
IloSolutionPool pool(env);
IloSolution s1(env, "SOLUTION 1"), s2(env, "SOLUTION 2");
pool.add(s1);
pool.add(s2);
for (IloSolutionPool::Iterator it(pool); it.ok(); ++it)
  env.out() << *it << endl;
pool.remove(s1);
env.out() << pool << endl;
pool.end();
```

Generation of the initial population

Any genetic algorithm begins by the generation of a suitable initial population. Our goal in this initial process is to create a set of solutions which have sufficient diversity. If you have a representation which is unconstrained (for instance, an indirect representation), then it is simple to create a completely random population. However, if this is not the case, the generation of the initial population becomes more subtle.

In Solver, the standard way of producing a solution is through the execution of a goal which instantiates the decision variables. In the EA Framework, the same technique can be applied repeatedly to generate an initial population. However, for the EA Framework, we need to be aware that Solver goals are completely deterministic; executing the same goal twice will produce exactly the same result. To avoid generating an initial population of identical solutions, we need to modify the behavior of the solution goal for each new population member.

The EA Framework provides a way of perturbing the normal behavior of a goal by altering its standard choices with a certain probability. This new goal can then be executed repeatedly to generate different solutions. A great advantage of this approach is that the probability can be *controlled*. By controlling the perturbation probability, the make-up of the population can be altered. When this probability is low, the solutions will resemble that generated by the unperturbed goal, but will be of reasonable quality if we assume that the standard goal is good. When high, solutions generated will be much more random in appearance, bearing little resemblance to the one generated by the standard goal. The resulting population is also likely to be lower in quality due to the large random element. Between the two extremes, a balance can be struck between diversity and quality.

Below, you will find two codes: one to generate a random solution pool and another to generate a solution pool from a randomly perturbed goal. Note that these code samples make use of a “prototype” solution. This prototype defines the structure for all solutions to be created; it has the correct decision variables and objective within it. The solutions to be included in the population are created by cloning the prototype. The notion of a solution prototype will also be used later.

Random solution:

```
IloIntVarArray vars = ...;
IloObjective obj = ...;
IloSolution prototype(env);
prototype.add(vars);
prototype.add(obj);

IloSolutionPool population(env);
IloRandom r = env.getRandom();
for (IloInt i = 0; i < popSize; i++) {
  IloSolution newMember = prototype.makeClone(env);
  for (IloInt j = 0; j < vars.getSize(); j++)
    newMember.setValue(vars[j], r.getInt(2));
  solver.solve(IloRestoreSolution(env, newMember)); // calculate objective
  newMember.store(solver);
  population.add(newMember);
}
```

Solution from goal:

```
IloIntVarArray vars = ...;
IloObjective obj = ...;
IloSolution prototype(env);
prototype.add(vars);
prototype.add(obj);
IloGoal generate = IloRandomPerturbation(env, stdGoal, 0.05);

IloSolutionPool population(env);
for (IloInt i = 0; i < popSize; i++) {
  while (!solver.solve(generate)) { }
  IloSolution newMember = prototype.makeClone(env);
  newMember.store(solver);
  population.add(newMember);
}
```

The goal modifier `IloRandomPerturbation` randomly modifies the branching decisions taken by `stdGoal`, such that (in this case) at each branch point, there is a 5% chance that the right branch will be taken before the left one, resulting in a slightly diversified population.

Evaluating, Comparing, and Selecting Solutions

During the operation of EAs, evaluation, comparison and selection of solutions are fundamental operations. In Solver IIM’s EA framework, these operations are carried out by

specializations of the template classes `IloEvaluator`, `IloComparator` and `IloSelector`. Selection of solutions is essential for the implementation of EAs as normally better quality solutions are favored for breeding as parents, and likewise poorer solutions tend to be selected as candidates for replacement. The operations of solution comparison and evaluation are often sub-ordinate to those of selection. Evaluators can give the quality of solutions in various different ways, whereas comparators can make pair-wise comparisons of solutions based on one or more different evaluations.

The most obvious evaluation of a solution, is simply the value of the objective function, but other equally simple evaluations are possible, such as the value of a particular variable. The functions `IloSolutionEvaluator` return different types of `IloEvaluator<IloSolution>` which evaluate either the objective value of a solution, or the value of one of its variables. More complex evaluators can then be built using the available arithmetic operators on evaluators. You can also write a particular custom evaluation from scratch using the macro `ILOEVALUATOR`. Below is a code which creates an evaluator which evaluates the profit of a particular solution:

```
IloIntVar profit(env), income(env), cost(env);

// Omitted: Build full model...

model.add(income == CalculateIncome(decisionVars));
model.add(cost == CalculateCost(decisionVars));
model.add(profit == income - cost);

IloEvaluator<IloSolution> eval = IloSolutionEvaluator(env, profit);

IloSolution solution(env);
solution.add(decisionVars);
solution.add(profit);
solution.add(income);
solution.add(cost);
IloSolver solver(model);
solver.solve(IloGenerate(env, decisionVars));
solution.store(solver);
IloInt profitValue = eval(solution);
```

Evaluations can be used to compare two solution objects, but a more general way exists which does not depend on the two solutions being compared having a unique evaluation. This second method uses a *comparator* object, and more specifically, an instance of `IloComparator<IloSolution>`. Comparators are the basic objects used to sort solution pools, as well as perform certain EA selection operations such as *tournament selection,* which will be covered later in this chapter. The following code sample shows how to sort a population in *decreasing* order of profit. The member function `makeGreaterThanComparator` is used to convert an evaluator into a comparator which prefers *higher* evaluations.

```
population.sort(eval.makeGreaterThanComparator());
IloSolution bestSolution = population.getSolution(0);
```

The population could have been sorted in increasing order of profit by making use of `makeLessThanComparator`. Alternatively, solution pools can store the manner in which they are to be sorted, and so the following is a (nearly) equivalent code. The difference is that afterwards, the population can be sorted in decreasing profit order via a simple call to `IloSolutionPool::sort()`.

```
population.setSortComparator(eval.makeGreaterThanComparator());
population.sort();
IloSolution bestSolution = population.getSolution(0);
```

More complicated comparisons can be created by composing simpler comparisons. For example, assume that, for equivalent profit, you prefer a solution which has a lower cost. This can be achieved by using a *lexicographic* comparison, when compares two objects on ordered criteria. If a particular criterion cannot distinguish if one solution is better than another, than this job falls to the next criterion on the list. The following code shows you how to create a comparator which prefers solutions of higher profit, but when profits are equal, those of lower cost.

```
IloEvaluator<IloSolution> evalProfit = IloSolutionEvaluator(env, profit);
IloEvaluator<IloSolution> evalCost = IloSolutionEvaluator(env, cost);
IloComparator<IloSolution> cmpProfit = evalProfit.makeGreaterThanComparator();
IloComparator<IloSolution> cmpCost = evalCost.makeLessThanComparator();
IloComparator<IloSolution> cmp = IloComposeLexical(cmpProfit, cmpCost);
```

Both solution evaluations and comparisons can be used as a basis for solution selection, which is an important part of evolutionary algorithms. The two most common places where you will select solutions are first, to choose parent solutions, and second to choose solutions that the new offspring will replace in the population. Some common selection methods supplied with the Solver IIM EA framework are *random selection, tournament selection* and *roulette wheel selection.* The first method, as the name suggests, selects solutions in an unbiased random fashion. The second uses a comparator in order to select solutions which tend to be better (or worse) than average. The third uses an evaluator to weight the probability of selection using the evaluation. To selector a random solution from a solution pool, you would create a random selector as below.

```
IloRandomSelector<IloSolution, IloSolutionPool> selector(env);
```

You could then use this selector to select a solution as follows:

```
IloSolution selectedSolution;
IloBool ok = selector.select(selectedSolution, population);
```

Although the usage described above is completely valid, normally you will delegate the call of the selector to other objects called *solution pool processors*, which will be discussed soon. At the same time, you will also become familiar with more complex selectors, such as the tournament selector.

Advanced Topic: Solution Pool (or Multiple) Evaluators

From time to time, it can be useful to evaluate all solutions in a solution pool *at once*. For instance, one might consider as evaluation of a solution its *rank* within the population. That its, the best solution would have rank 1, the next best rank 2, *etc.* In order to perform such an evaluation, you need to have access to the entire solution pool: each solution cannot be evaluated in isolation. To do this, in Solver IIM, there is the notion of a multiple evaluator which can be used to evaluate entire solution pools at once. You will find more information on evaluating solution pools in Chapter 30, *Using More Advanced EA Features*.

Evolution of the population

With the population initialization complete, a GA can start to work on its principal job: evolving the population. Evolution of the population normally has a very precise goal: production of the best solution possible in the time given. To attain this goal, the genetic algorithm uses three main operations. These are, the selection of a *parent* or *parents* to undergo some genetic operation, the production of *offspring* by mutating and/or combining the parent(s) and merging the offspring and current population into a new one. Each "round" of these operations is termed a *generation*. Generations are repeated as long as is deemed necessary (usually up to a certain time limit) with the goal of improving the average quality of the solutions in the population. (Improving average quality normally leads to an improvement of the highest quality solution in the population—the ultimate goal). Hereafter, each of the three basic phases of a single generation is described from the standpoint of the EA framework. However, before this, we introduce a concept which is common to all of these phases; that of solution pool processors.

Solution pool processors

Solution pool processors are objects which perform quite a general role, and form the umbrella under which the various selection, replacement, and genetic operators reside. Solution pool processors are objects which take a solution pool as input and produce another solution pool as output. This quite general mechanism can be used to select solutions from the input pool to place on the output pool, or produce entirely new solutions on the output pool.

Processors can be chained so that the output of one processor becomes the input of another using the `>>` operator. Thus, pipelines of processors can be built up which embody quite complex algorithms. Below is a code which creates a processor which `proc` takes a solution pool, `population`, selects the three best solutions according to the value of the variable

profit, then one solution randomly amongst those. The resulting solution is then placed in the pool selected.

```
IloSolutionPool selected(env);
IloEvaluator<IloSolution> evalProfit = IloSolutionEvaluator(env, profit);
IloComparator<IloSolution> cmpProfit = evalProfit.makeGreaterThanComparator();

IloBestSelector<IloSolution,IloSolutionPool> selBest(cmpProfit);
IloPoolProc selBestProc = IloSelectSolutions(env, selBest);

IloRandomSelector<IloSolution,IloSolutionPool> selRnd(env);
IloPoolProc selRndProc = IloSelectSolutions(env, selRnd);

IloPoolProc proc = population >> selBestProc(3) >> selRndProc >> selected;
```

Note that in the context of the >> operator, solution pools (in this case population and selected) can be used as processors. If the pool has no processors supplying it with solutions (population), it produces the specified pool on its output. If the pool appears elsewhere, then it passes on the solutions it receives (if there is another processor to the right) as well as placing them in the pool.

If a processor performs a genetic operation to produce a new solution, Solver's search and propagation is used to do this. One example of a processor which performs selection, then mutation, might be described as follows:

```
IloPoolProc select = IloSelectSolutions(
  env, IloRandomSelector<IloSolution,IloSolutionPool>(env)
);
IloEAOperatorFactory factory(env, vars);
IloPoolProc mutate = factory.mutate(probability);
IloPoolProc produce = population >> select >> mutate;
```

Note the use of the class IloEAOperatorFactory used to produce processors which perform genetic operations. This class will be fully introduced later.

Processors are executed by an instance of IloSolver using the IloExecuteProcessor goal. When executed, a process similar to extraction transforms the processors into Solver-based Ilc objects which perform the actual calculations. The operation of the processors is demand-based. To execute a pool processor, you ask the processor to produce a number of solutions, which is not specified will default to the natural number for the processor (usually one). When processors are in a chain, this request goes directly to the right-most processor of the chain. This processor in turn will calculate the number of solutions it needs as input to fulfill the request. The processor then asks the processor to its left in the chain for the number of solutions it needs before carrying out its work and producing the solutions required. If any processor fails to provide the number of solutions desired, it is repeatedly

invoked until the number of solutions desired is attained. The number of solutions to supply can be specified using the parenthesis operator:

```
IloPoolProc select = IloSelectSolutions(
  env, IloRandomSelector<IloSolution,IloSolutionPool>(env)
);
IloEAOperatorFactory factory(env, vars);
IloPoolProc mutate = factory.mutate(probability);
IloPoolProc produce = population >> select >> mutate;
IloSolutionPool offspring(env);
solver.solve(IloExecuteProcessor(env, produce(30) >> offspring));
```

Here, the processor `produce` is asked to produce 30 solutions; the result is placed in the `offspring` solution pool.

Choice of parents

The choice of parent or parents which will participate in a genetic operation in the EA framework is made by a solution processor. Normally, a processor playing the role of a selector takes the population as input, and generates a requested number of solutions. (The number to be requested will depend on the exact type of genetic operator which requests the solutions.) Although a pool processor performs the selection of solutions, it is really only a shell which encapsulates an `IloSelector<IloSolution,IloSolutionPool>` object which does the main work. The function `IloSelectSolutions` produces a pool processor from such a selector. The `IloSelectSolutions` function also takes a boolean parameter which dictates if solutions should be selected without duplicates. That is, is it valid for the same solution to appear more than once in the selection?

An example of a selector which is commonly used is the tournament selector, characterized by its tournament size *k*. This selector selects one solution from the population as follows. First, *k* different solutions are selected randomly, and without bias, from the input pool. Then, the highest quality solution from these *k* solution is chosen as the final selected solution. This process is repeated to produce the desired number of selections. When *k=1*, solutions are selected randomly. As the value of *k* is increased, the chances are increased that better solutions are selected. However, a balance should be struck, as a value of *k* that is too high will result in poorer quality solutions rarely being selected and in the context of a genetic algorithm could lead to limited diversity of generated solutions. Below is an example of the use of the tournament selector with tournament size 3.

```
IloTournamentSelector<IloSolution,IloSolutionPool> tsel(
  env, 3, IloBestSolutionComparator(env)
);
IloPoolProc select = population >> IloSelectSolutions(env, tsel);
```

How the tournament selector compares solutions is determined by a *solution comparator*. A solution comparator is a simple object which determines if one solution is better than another according to some criterion. `IloBestSolutionComparator` uses the

IloSolution::isBetterThan member function to compare solutions. You could equally have the following:

```
IloComparator<IloSolution> compare = IloWorstSolutionComparator(env);
IloTournamentSelector<IloSolution,IloSolutionPool> tsel(env, 3, compare);
IloPoolProc select = IloSelectSolutions(env, tsel);
```

This would tend to select poorer solutions with greater probability. You may inquire as to the usefulness of such a selector as it surely selects poor quality solutions. We will return to this in a following section on solution replacement.

Genetic operations

After selection of parents, a GA then typically applies genetic operators to those parents to produce new solutions: the offspring. The EA framework comes with various built-in mutation and combination operators, operating on arrays of constrained integer variables. All genetic operators provided with the EA framework are randomized and you should follow this tenet if you write your own operators. During the execution of a genetic algorithm, the same operator may be applied to the same solution or solutions more than once. If operators where not randomized, search would quickly stagnate, unable to provide new different solutions.

All genetic operators are represented by the class IloPoolOperator, which can be seen as a special type of pool processor; in fact an automatic cast exists between a pool operator and pool processor. An operator depends on an instance of IloSolver to produce its solutions and works as follows:

1. The operator asks for the solutions it needs to perform the operation. Typically, it asks the processor immediately “upstream” of a >> operator.
2. The operator instantiates the constrained variables in the solver according to the definition of the operator, and the values of the input solutions. For example, imagine that the operator was to mutate 0-1 variables uniformly. To take advantage of constraint programming, the operator could launch a goal search where, at each node, a choice point (IlcOr) is created which instantiates the variable in question to either 0 or 1. With high probability, the left branch would concur with the value in the input solution, and with low probability (the mutation probability), the branches would be reversed.
3. The operator looks for a prototype solution. This can be given explicitly by the user to the operator, but if not, one of the input solutions is used as a prototype.
4. The prototype is cloned, is stored using solution.store(solver) and transferred to the output of the operator.

Steps 1, 3, and 4 are automated and are common to all operators. They comprise the demand for input, the creation of a new solution, the transfer of solution values from the solver to the solution, and the transfer of the new solution to the output of the processor. Only Step 2 need

vary from operator to operator, and it is this step that you will be responsible for if you want to write your own genetic operator. (Some information for Step 1 is also required—the number of solutions that the operator needs as input to properly function.)

The principle of Step 2 of the above process is to place the decision variables of the problem in some desired state congruent with the definition of the operator. A goal can also be seen as performing this job. As such, it is possible to produce an operator from a standard goal, so long as a solution prototype is given explicitly. This approach can be used for creating initial populations as follows:

```
IloIntVarArray vars = ...;
IloObjective obj = ...;
IloSolution prototype(env);
prototype.add(vars);
prototype.add(obj);
IloGoal generate = IloRandomPerturbation(env, stdGoal, 0.05);

IloSolutionPool population(env);
IloPoolProc genProc = IloSource(env, generate, prototype)(50) >> population;
solver.solve(IloExecuteProcessor(env, genProc));
```

The `IloSource` operator requests no inputs, runs goal `generate`, clones `prototype`, stores the resulting solution, and places it in the output pool. Setting the output size to 50 ensures that this process is repeated until `IloSource` has gathered 50 solutions in its output pool.

The genetic operators provided in the EA framework are generated by a factory class `IloEAOperatorFactory`. This class allows built-in operators to be generated as well as configured by solution prototypes and search limits. For example, the following code creates a factory which will generate operators with a search limit of 100 fails in which to generate a solution and a prototype of `prototype`:

```
IloSolutionPool population(env);
IloEAOperatorFactory fact(env, vars);

// Prototype
fact.setPrototype(prototype);

// All operators generated by the factor will have this limit applied.
fact.setSearchLimit(IloFailLimit(env, 100));

// Generate an initial random population of 30 solutions
solver.solve(IloExecuteProcessor(env, fact.randomize()(30) >> population));

// Mutation operator, with 1% probability.
IloPoolProc mut = fact.mutate(0.01);
```

Note that if the operator could not produce a solution in the number of fails given as a limit, it will be repeatedly called until it produces such a solution. Each time this happens, new input solutions are requested before the operator is re-applied.

The IIM EA framework also supports dynamic pool processor selection. That is, a pool processor can be defined to be a dynamic selection among a set of such processors.

This is an elegant way of specifying composite processors. Imagine a processor which when invoked, behaves as one of *n* subordinate processors, the choice being made at random. This behavior can be specified quite simply in the EA framework as follows:

```
IloPoolProc op1 = ...;
IloPoolProc op2 = ...;
IloTournamentSelector<IloSolution,IloSolutionPool> tsel(
  env, 3, IloBestSolutionComparator(env)
);
IloPoolProc tourSelect = IloSelectSolutions(env, tsel);
IloPoolProc rndSelect = IloSelectSolutions(
  env, IloRandomSelector<IloSolution,IloSolutionPool>(env)
);

// Build an array of processors.
IloPoolProcArray ops(env);
ops.add(op1);
ops.add(op2);

// A composite operator which executes a randomly selected operator.
IloPoolProc composite = IloSelectProcessor(
  env, ops, IloRandomSelector<IloPoolProc,IloPoolProcArray>(env)
);

// Generate 30 offspring.
IloSolutionPool offspring(env);
IloPoolProc breed = population >> tourSelect >> composite(30) >> offspring;
solver.solve(IloExecuteProcessor(env, breed));
```

Notice the function `IloSelectProcessor` which uses a selection of a pool processor from an array of pool processors to pick the “sub-processor” to execute each time processor `composite` is executed.

The result is that the `offspring` pool will be filled by solutions which have been generated by a non-deterministic interleaving of the executions of `op1` and `op2`. One additional point to note in the above is that `op1` and `op2` need not be consistent in the numbers of solutions that they require as input. The tournament selector `tourSelect` is only asked for the correct number of solutions to supply *after* `composite` has decided on the exact subprocessor to execute. Thus, every time `composite` is invoked, it may demand a different number of solutions from the tournament selector.

Solution Replacement

The final step of a genetic algorithm generation is to combine the intermediate offspring with the current population to form a new population. For convenience, a “steady-state” model is often used, where the size of the population does not change from generation to generation. This is not imposed by the EA framework, but does simplify discussion and genetic algorithm design. In such a steady-state system, we refer to the above combination

process as “solution replacement.” That is, we replace some of the members of the population with offspring.

Replacement processors, like selectors, but unlike operators, are basically raw pool processors. Normally, they receive the offspring as input, and the current population as a parameter. They then modify the population as desired, typically keeping its size fixed, and generate as output the solutions which are to be discarded. The pool processor `IloReplaceSolutions` can be used to show how this procedure functions:

```
IloComparator<IloSolution> compareWorst = IloWorstSolutionComparator(env);
IloSolutionPoolSelector deadSelector =
IloTournamentSelector<IloSolution,IloSolutionPool>(env, 3, compareWorst);
IloPoolProc replace = IloReplaceSolutions(env, population, deadSelector);
IloPoolProc combine = offspring >> replace >> IloDestroyAll(env);
```

Above, a tournament selector `deadSelector` is created which tends to select poorer solutions (via use of the comparator `IloWorstSolutionComparator`). This selector is passed to the pool processor `IloReplaceSolutions` which works as follows. First, the population passed as parameter is fed as input to `deadSelector`. Next, the selector is asked to select as many solutions from the population as there are solutions in the input pool (*i.e.* the number of offspring). Finally, the selected solutions are removed from the population and placed on the output pool, after which the solutions in the input pool (the offspring) are added to the population.

In the EA framework, destruction of solutions which have been permanently removed from the population is explicit. The processor `IloDestroyAll` calls the `end` method on all solutions in its input pool. This destruction phase completes the combination operation.

Advanced Topic: Monitoring and listeners

The EA framework provides a system for monitoring the operation of any genetic algorithm created via a “listener” framework. Listeners can be added to pools of solutions or genetic operators and can be invoked when elements add or leave the pool, or when new solutions are produced. This makes it easy, for example, to monitor the selection of parents or the selection of solutions to replace. Statistics on successful and unsuccessful operators can also be gathered for simple reporting, or for influencing the future operation of the GA. Monitoring is fully introduced in the examples delivered with the framework.

29

Modeling and Solving a Basic EA Problem

In this lesson, you will use the Solver IIM Evolutionary Algorithm (EA) Framework to solve the "one max" problem, a basic binary digit (bit) optimization problem. In this problem, you try to find the solution that maximizes the number of times the digit 1 appears within a given set of 0-1 variables.

To understand how this problem works, consider an initial population of 4 solutions, each a string of 6 bits:

001101

010111

101000

110010

You use evolutionary algorithm techniques to create new generations until one of these generations contains the desired solution:

111111

In the example that follows, your initial population is 50 randomly generated solutions, each of which is 50 bits long. You use tournament selection, mutation, and replacement of parents by offspring to search for the solution that maximizes the objective function. The objective function is the number of times the digit 1 appears. The optimal solution is one where the digit 1 appears 50 times.

While this is not a difficult optimization problem, this example allows you to become familiar with the evolutionary algorithm concepts without too many modeling concerns. The solving process is split into two steps:

1. Create an initial population using random generation.
2. Write a basic generation loop:
   - Create the reproduction goal, which uses tournament selection to choose which parents will reproduce and then uses the mutation operator reproduce.
   - Create the replacement goal. In this example, the parent population will be entirely replaced by the offspring population.
   - For each generation, solve the problem using the generation goal (which combines the reproduction and replacement goals) and display the generation statistics. Break the loop when the solution that equals the optimum value (the problem size) is found or a certain number of generations is reached.

The first step in this procedure is discussed in the section "Creating an initial solution pool" on page 510. The second step is discussed in the section "Writing a simple generation loop" on page 514.

Creating an initial solution pool

In this section, you will learn how to:

- build a solution prototype
- create the objective function
- create an operator factory to generate predefined genetic operators
- create a solution pool
- perform a goal which fills the solution pool with random solutions

Make a copy of the example file `YourSolverHome/examples/src/tutorial/ea1max_init_partial.cpp` and open this copy 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 section. At the end of the section, you will have completed the example and you can compile and run the program.

The main routine starts by parsing command line parameters for the problem size and the population size. You declare an `IloEnv` object used for allocation followed by a "try/catch" block to trap possible exceptions which may occur within the function body execution.

Step 1 Parse command line parameters

Add the following code after the comment
`//Parse command line parameters`

```
IloInt problemSize = 50;     // Bits per solution
IloInt populationSize = 50;  // Number of solutions in population
if (argc > 1)
  problemSize = atoi(argv[1]);
if (argc > 2)
  populationSize = atoi(argv[2]);
env.out() << "Problem size: " << problemSize << endl;
env.out() << "Population size: " << populationSize << endl;
```

Next, you build a Concert model of the "one max" problem. You declare an instance of `IloModel`. Then, you declare an `IloBoolVarArray` of bits. You constrain the `count` integer variable to be the sum of the binary digits in the `bits` array. The `count` variable has a lower bound of 0 for an array composed entirely of 0's and an upper bound of 50 for an array composed entirely of 1's.

Step 2 Create the model

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

```
IloModel model(env);
IloBoolVarArray bits(env, problemSize);     // The bits to count
for (IloInt v = 0; v < problemSize; v++) {
  char name[10];
  sprintf(name, "%02ld", v + 1);
  bits[v].setName(name);                         // Name the variable
}
IloIntVar count(env, 0, problemSize, "count");   // The bit count
model.add(count == IloSum(bits));                // Count the bits
```

As all generated solutions must contain the variable array you just modeled, you define a solution prototype. This prototype is a template for all solutions that will be placed in the initial population. Internally, unique solutions will be generated by cloning this prototype and storing the solver's state. You also define the objective function, which is to maximize the `count` variable. This addition is useful when solutions are to be compared or sorted.

Step 3 Create the solution prototype

Add the following code after the comment //Create the solution prototype

```
IloSolution prototype(env);
prototype.add(bits);
IloObjective obj = IloMaximize(env, count, "Obj");
prototype.setObjective(obj);
```

You will make use of a factory of operators which operate over an array of decision variables specified to the factory. This factory process simplifies the definition of standard operators as the factory has access to certain common parameters, for example, the solution prototype.

Here, you use an operator which randomizes the values of the variables specified in the constructor of the factory, in order to generate a random population. This operator will be called until the number of solutions is equal to the desired population size. Solver verifies the legality of each solution by generating the solutions in a goal search which is backtracked each time a solution is produced. The fail limit specified is applied to the generation of each solution, meaning that the effort to generate each solution is limited. If a solution is not generated within this limit, this small subsearch is abandoned, and the generation loop reiterated.

Step 4 Initialize the population

Add the following code after the comment //Initialize the population

```
IloSolver solver(model);
IloSolutionPool population(env);
IloEAOperatorFactory factory(env, bits);
factory.setPrototype(prototype);
factory.setSearchLimit(IloFailLimit(env, problemSize));
solver.solve(IloExecuteProcessor(
  env, factory.randomize("rand")(populationSize) >> population
));
```

You can then display the solution pool's content.

Step 5 Display the population

Add the following code after the comment //Display the population

```
env.out() << "INITIAL POPULATION " << population << endl;
```

Finally, the routine's "try/catch" block as well as its body are closed. This code is provided for you:

```
  } catch(IloException ex) {
    env.out() << "Caught: " << ex << endl;
  }
  env.end();
  return 0;
}
```

You have now completed the model of the "one max" problem and created the initial solution pool. In the next section, you will learn how to write a simple generation loop.

Step 6 Compile and run the program

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

```
Problem size: 50
Population size: 50
INITIAL POPULATION IloSolutionPoolI (no name)
    [0] IloSolution[ 01[0] 02[1] 03[1] 04[1] 05[0] 06[0] 07[1] 08[1] 09[1]
10[0] 11[1] 12[1] 13[0] 14[1] 15[0] 16[0] 17[1] 18[1] 19[1] 20[1] 21[0] 22[1]
23[1] 24[1] 25[0] 26[0] 27[1] 28[0] 29[1] 30[1] 31[1] 32[1] 33[1] 34[1] 35[1]
36[0] 37[0] 38[1] 39[0] 40[1] 41[1] 42[1] 43[1] 44[1] 45[1] 46[0] 47[1] 48[1]
49[0] 50[1] Obj{Max} [34] ] (Obj = 34)
    [1] IloSolution[ 01[0] 02[1] 03[1] 04[1] 05[0] 06[0] 07[0] 08[1] 09[1]
10[1] 11[1] 12[1] 13[0] 14[1] 15[1] 16[1] 17[0] 18[0] 19[1] 20[1] 21[1] 22[0]
23[0] 24[1] 25[0] 26[1] 27[1] 28[0] 29[0] 30[0] 31[1] 32[1] 33[1] 34[0] 35[0]
36[1] 37[0] 38[0] 39[0] 40[1] 41[1] 42[0] 43[1] 44[0] 45[1] 46[0] 47[0] 48[0]
49[0] 50[1] Obj{Max} [26] ] (Obj = 26)
    [2] IloSolution[ 01[1] 02[1] 03[1] 04[0] 05[0] 06[1] 07[1] 08[1] 09[0]
10[0] 11[1] 12[0] 13[1] 14[0] 15[1] 16[0] 17[0] 18[1] 19[0] 20[0] 21[0] 22[0]
23[0] 24[1] 25[0] 26[1] 27[1] 28[0] 29[0] 30[0] 31[1] 32[1] 33[1] 34[1] 35[0]
36[1] 37[0] 38[1] 39[0] 40[0] 41[0] 42[0] 43[1] 44[0] 45[1] 46[0] 47[0] 48[0]
49[1] 50[0] Obj{Max} [22] ] (Obj = 22)
    [3] IloSolution[ 01[0] 02[0] 03[1] 04[0] 05[1] 06[1] 07[1] 08[0] 09[0]
10[0] 11[1] 12[0] 13[0] 14[1] 15[0] 16[1] 17[1] 18[0] 19[0] 20[1] 21[0] 22[0]
23[1] 24[1] 25[0] 26[0] 27[1] 28[0] 29[0] 30[1] 31[0] 32[1] 33[1] 34[0] 35[0]
36[1] 37[0] 38[1] 39[1] 40[1] 41[0] 42[1] 43[1] 44[0] 45[1] 46[1] 47[1] 48[1]
49[1] 50[1] Obj{Max} [27] ] (Obj = 27)
    [4] IloSolution[ 01[1] 02[0] 03[1] 04[0] 05[0] 06[0] 07[1] 08[1] 09[0]
10[0] 11[0] 12[1] 13[1] 14[0] 15[0] 16[1] 17[0] 18[1] 19[0] 20[1] 21[0] 22[1]
23[0] 24[1] 25[0] 26[1] 27[0] 28[1] 29[1] 30[1] 31[1] 32[0] 33[0] 34[1] 35[1]
36[1] 37[1] 38[0] 39[0] 40[0] 41[0] 42[0] 43[1] 44[0] 45[1] 46[0] 47[1] 48[0]
49[1] 50[0] Obj{Max} [24] ] (Obj = 24)

                              ( abridged )
```

The complete program is available in the `YourSolverHome/examples/src/ea1max_init.cpp` file.

Review Exercises

1. Iterate over the created solution pool and print each solution's objective value.
2. Retrieve the best, worst, and average solution objective value from the solution pool.

Suggested Answers

Exercise 1

For this exercise, you should use a solution pool iterator and the `getObjectiveValue` API of `IloSolution`.

```
for (IloSolutionPool::Iterator pi(population); pi.ok(); ++pi)
  env.out() << (*pi).getObjectiveValue() << " ";
env.out() << endl;
```

Exercise 2

The average is calculated by iterating and accumulating a sum which will eventually be divided by the population size to produce an average. The best and worse could also be maintained in the iterator loop by examination of the objectives of each solution. An alternative is to use the `getBestSolution/getWorstSolution` API of `IloSolutionPool`, as shown here:

```
IloNum sum = 0.0;
for (IloSolutionPool::Iterator pi(population); pi.ok(); ++pi)
  sum += (*pi).getObjectiveValue();
IloNum best = population.getBestSolution().getObjectiveValue();
IloNum worst = population.getWorstSolution().getObjectiveValue();
env.out() << "Best  = " << best << endl
          << "Worst = " << worst << endl
          << "Mean  = " << sum / population.getSize() << endl;
```

Writing a simple generation loop

In this section you will learn how to:

- use a predefined selector
- specify a genetic operator
- replace the existing population by offspring
- access the best and worst solutions for each generation

Make a copy of the example file `YourSolverHome/examples/src/tutorial/ea1max_gen_partial.cpp` and open the copy in your development environment. As in

the first section of this lesson, you will fill in the blanks in each step. At the end of the section, you will have completed the example and you can compile and run the program.

As in the previous example you parse command line arguments. In this example, there is another command line argument: the maximum number of generations. This code is provided for you:

```
IloInt problemSize = 50;     // Bits per solution
IloInt populationSize = 50;  // Number of solutions in population
IloInt maxGeneration = 60;   // Generation count limit
if (argc > 1)
  problemSize = atoi(argv[1]);
if (argc > 2)
  populationSize = atoi(argv[2]);
if (argc > 3)
  maxGeneration = atoi(argv[3]);
env.out() << "Problem size: " << problemSize << endl;
env.out() << "Population size: " << populationSize << endl;
env.out() << "Max generation: " << maxGeneration << endl;
```

To keep track of population evolution you write a function to display some statistical information:

- the best solution objective value
- the worst solution objective value
- the average solution objective value

This function is used after the population initialization and after the creation of each new generation.

Step 1 Display the generation statistics

Add the following code after the comment `//Display the generation statistics`

```
void DisplayGenerationStatistics(ostream& stream,
                                 IloInt generation,
                                 IloSolutionPool population) {
  IloNum sum = 0.0;
  for (IloSolutionPool::Iterator it(population); it.ok(); ++it)
    sum += (*it).getObjectiveValue();
  stream << "GENERATION " << generation
         << " WORST "
         << population.getWorstSolution().getObjectiveValue()
         << " BEST "
         << population.getBestSolution().getObjectiveValue()
         << " AVERAGE " << (sum / population.getSize())
         << endl;
}
```

Solutions processed by the operators will be chosen using tournament selection. The selector `IloTournamentSelector` will be used. A tournament selector operates by randomly selecting *n* (the tournament size) solutions from its input pool, and placing the best of these resulting solutions in the output pool. In this way, better solutions are favored for selection over poorer ones. Here a tournament size of 2 is used: two random solutions are chosen from the population, and the better is placed in the parent pool. You also specify via a Boolean parameter that when more than one solution is selected by the selector, the same solution cannot be selected more than once.

Step 2 Use tournament selection to choose parents

Add the following code after the comment
`//Use tournament selection to choose parents`

```
IloTournamentSelector<IloSolution, IloSolutionPool> tsel(
  env, 2, IloBestSolutionComparator(env)
);
IloPoolProc selector = IloSelectSolutions(env, tsel, IloTrue);
```

In this example, the solution will be processed using a single mutation operator. The mutation operator simply changes the value of each element of the solution to another of its values with a constant probability. Here, you choose this probability to be `1.0/problemSize` to ensure that, on average, one site is mutated each time. This is simply parameterization and not a necessary condition. The operator is supplied by a factory which returns an object of type `IloPoolOperator`. An operator takes an input pool of solutions and instantiates Solver's constrained variables according to its function. An object of type `IloPoolOperator` does not produce a solution object, and so cannot be used directly with pool processors. The operator is converted to a pool processor (type `IloPoolProc`) automatically. This conversion creates a pool processor which invokes the operator, and creates an output solution from the state of the Solver.

Step 3 Use mutation as a single operator

Add the following code after the comment `//Use mutation as a single operator`

```
IloPoolProc applyOp = factory.mutate(1.0 / problemSize, "mutate");
```

In this example, you will replace the entire population by the offspring at each generation. Thus, at each generation, `populationSize` offspring must be created. The reproduction goal accumulates solutions in the `offspring` pool until the pool's size reaches `populationSize`.

Step 4 Create the reproduction goal

Add the following code after the comment `//Create the reproduction goal`

```
IloSolutionPool offspring(env, "offspring");
IloGoal reproduceGoal = IloExecuteProcessor(
  env, population >> selector >> applyOp(populationSize) >> offspring
);
```

Then you create a replacement goal which:

- destroys existing solutions
- replaces the entire population with the newly generated offspring

Step 5 Replace the entire population with offspring

Add the following code after the comment
`//Replace the entire population with offspring`

```
IloGoal replacementGoal =
  IloExecuteProcessor(env, population >> IloDestroyAll(env)) &&
  IloExecuteProcessor(env, offspring >> population);
```

A single generation is created by chaining reproduction and replacement goals.

Step 6 Each generation reproduces and replaces solutions

Add the following code after the comment
`//Each generation reproduces and replaces solutions`

```
IloGoal generationGoal = reproduceGoal && replacementGoal;
```

You define the optimum value which will be used to stop the generation loop. Note that, for this example, since the optimal is known, such a determination is possible. In most real world cases, the optimal will not be known and the more usual terminating condition is on the number of iterations or a time limit.

Step 7 Set up the optimum value

Add the following code after the comment `//Set up the optimum value`

```
IloInt optimumValue = problemSize;
env.out() << "Optimum value: " << optimumValue << endl;
```

Finally, you create the loop to search for the best solution.

Step 8 The main generational loop

Add the following code after the comment `//The main generational loop`

```
for (generation++; generation < maxGeneration; generation++) {
  solver.solve(generationGoal);
  DisplayGenerationStatistics(env.out(), generation, population);
  if (population.getBestSolution().getObjectiveValue() == optimumValue) {
    env.out() << "OPTIMUM FOUND " << endl;
    break;
  }
}
```

You have now learned how to write a simple generation loop.

Step 9 Compile and run the program

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

```
Problem size: 50
Population size: 50
Max generation: 60
GENERATION 0 WORST 17 BEST 34 AVERAGE 25.06
Optimum value: 50
GENERATION 1 WORST 20 BEST 35 AVERAGE 27.4
GENERATION 2 WORST 24 BEST 35 AVERAGE 29.08
GENERATION 3 WORST 26 BEST 37 AVERAGE 31.02
GENERATION 4 WORST 29 BEST 38 AVERAGE 32.18
GENERATION 5 WORST 28 BEST 38 AVERAGE 32.68
GENERATION 6 WORST 29 BEST 39 AVERAGE 33.58
GENERATION 7 WORST 31 BEST 39 AVERAGE 34.86
GENERATION 8 WORST 31 BEST 38 AVERAGE 35.04
GENERATION 9 WORST 32 BEST 39 AVERAGE 35.38
GENERATION 10 WORST 32 BEST 39 AVERAGE 35.66
GENERATION 11 WORST 33 BEST 39 AVERAGE 36.2
GENERATION 12 WORST 33 BEST 39 AVERAGE 36.02
GENERATION 13 WORST 34 BEST 39 AVERAGE 36.2
GENERATION 14 WORST 33 BEST 38 AVERAGE 35.84
GENERATION 15 WORST 32 BEST 39 AVERAGE 36.18

                (abridged)
```

The complete program is available in the `YourSolverHome/examples/src/ea1max_gen.cpp` file.

Review Exercises

1. Increase the tournament size of the parent selector and note if the population converges faster.

2. Try other operators, such as uniform crossover. Use the member function `uniformXover` of the operator factory.

Suggested Answers

Exercise 1

Change the parameter “2” in `IloTournamentSelector` to “5” for example:

```
IloPoolProc selector = IloTournamentSelector(
  env, 5, IloBestSolutionComparator(env), IloTrue
);
```

You should see that the optimal is found within the default generation limit this time, whereas it was not before.

Exercise 2

Change the generation of the solution operator to:

```
IloPoolProc applyOp = factory.uniformXover(0.5, "uXover");
```

The `0.5` indicates that features of the child should probably be equally drawn from both parents of the crossover. With this operator, the optimum is found more quickly than when using mutation.

CHAPTER

30

Using More Advanced EA Features

This lesson uses the “one max” problem to demonstrate some of the more advanced features available in the Solver Evolutionary Algorithm (EA) Framework. In the “one max” problem, you try to find the solution that maximizes the number of times the digit 1 appears in a set or string of bits.

This lesson contains three sections:

- *Using custom selectors and multiple operators* on page 521
- *Using listeners and comparators* on page 525
- *Using pool evaluators and partial replacement* on page 533

All three sections are based on the model and basic generation loop explained in Chapter 29, *Modeling and Solving a Basic EA Problem*.

Using custom selectors and multiple operators

In this section, you will learn how to:

- define a customized selector for parents
- use a processor pool to manage multiple genetic operators
- instantiate a selector

Make a copy of the example file `YourSolverHome/examples/src/tutorial/ea1max_cust_partial.cpp` and open this copy 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 section. At the end of the section, you will have completed the example and you can compile and run the program.

You define the runtime custom selector that will be invoked at solve time. This is a tournament selector whose constructor takes one argument—the tournament size. Recall that a tournament selector takes the best of *n* solutions drawn randomly from the population, where *n* is the tournament size. In the custom operator, you add each selected solution to the selector's output solution pool. The use of `IloSolution::isBetterThan` facilitates the construction of this selector.

Step 1 The custom selector

Add the following code after the comment `//The custom selector`

```
ILOCTXSELECTOR1(CustomSelector, IloSolution, IloSolutionPool, in,
                IloSolver, solver, IlcInt, tournamentSize) {
  IloInt size = in.getSize();
  if (size != 0) {
    IlcRandom rnd = solver.getRandom();
    IloSolution candidate = in.getSolution(rnd.getInt(size));
    for (IlcInt tournament = 1; tournament < tournamentSize; tournament++) {
      IloSolution competition = in.getSolution(rnd.getInt(size));
      if (competition.isBetterThan(candidate))
        candidate = competition;
    }
    select(candidate);
  }
}
```

In the main code, instead of a predefined selector, you will instantiate your own selector, specifying the tournament size.

Step 2 Create the user-defined tournament selector

Add the following code after the comment
`//Create the user-defined tournament selector`

```
    IloPoolProc selector =
      IloSelectSolutions(env, CustomSelector(env, 3), IloTrue);
```

This creates a pool processor which uses `CustomSelector` to select solutions. Here a selector of tournament size 3 is created. The boolean parameter to `IloSelectSolutions` indicates whether the same solution can be selected more than once or when `selector` is asked for more than one solution. In this case, `IloTrue` is specified, meaning that all

solutions generated must be unique, and duplicates are forbidden. This ensures that later, a crossover cannot be performed between two instances of the same parent.

Now you will learn how to use more than one genetic operator to evolve the population. These alternative operators will be stored in a processor pool, which is similar to a solution pool.

Step 3 Create a processor pool to store operators

Add the following code after the comment
`//Create a processor array to store operators`

```
IloPoolProcArray ops(env);
```

You will keep the mutation operator in your operator set, and so you add the mutation operator to the processor pool.

Step 4 Add the mutation operator to the pool

Add the following code after the comment
`//Add the mutation operator to the pool`

```
ops.add(factory.mutate(1.0 / problemSize, "mutate"));
```

You also add the uniform crossover operator to the processor array in the same manner as for mutation.

Step 5 Add the crossover operator to the pool

Add the following code after the comment
`//Add the crossover operator to the pool`

```
ops.add(factory.uniformXover(0.5, "uXover"));
```

Now you produce a new composite operator which selects a random operator from the array and invokes it. You do this by:

- creating a random selector on the processor array
- using `IloSelectProcessor` to invoke the selector and execute the processor selected.

Step 6 Creates a goal selector which will choose the goal to apply

Add the following code after the comment
`//Creates a goal selector which will choose the goal to apply`

```
    IloPoolProc applyOp = IloSelectProcessor(
      env, ops, IloRandomSelector<IloPoolProc,IloPoolProcArray>(env)
    );
```

You have now learned how to write a custom selector and use multiple operators in a processor pool. In the next section, you will learn how to use listeners.

Step 7 Compile and run the program

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

```
Problem size: 50
Population size: 50
Max generation: 60
GENERATION 0 WORST 17 BEST 34 AVERAGE 25.06
Optimum value: 50
GENERATION 1 WORST 20 BEST 33 AVERAGE 27.7
GENERATION 2 WORST 24 BEST 34 AVERAGE 30.54
GENERATION 3 WORST 29 BEST 36 AVERAGE 32.34
GENERATION 4 WORST 29 BEST 38 AVERAGE 33.52
GENERATION 5 WORST 30 BEST 41 AVERAGE 34.76
GENERATION 6 WORST 32 BEST 41 AVERAGE 35.8
GENERATION 7 WORST 33 BEST 42 AVERAGE 37.02
GENERATION 8 WORST 34 BEST 42 AVERAGE 38.6
GENERATION 9 WORST 35 BEST 44 AVERAGE 39.28
GENERATION 10 WORST 37 BEST 46 AVERAGE 40.92
GENERATION 11 WORST 40 BEST 46 AVERAGE 42.36
GENERATION 12 WORST 40 BEST 47 AVERAGE 43.36
GENERATION 13 WORST 42 BEST 48 AVERAGE 44.44
GENERATION 14 WORST 42 BEST 49 AVERAGE 45.64
GENERATION 15 WORST 43 BEST 49 AVERAGE 46.38
GENERATION 16 WORST 43 BEST 50 AVERAGE 47.06
OPTIMUM FOUND
```

The complete program is available in the `YourSolverHome/examples/src/ea1max_cust.cpp` file.

Review Exercises

1. Adjust the selector to produce the worst of *n* solutions (*n* being the tournament size) and examine the behavior.

Suggested Answers

Exercise 1

Simply change the line:

```
if (competition.isBetterThan(candidate))
```

to:

```
if (competition.isWorseThan(candidate))
```

You should see that, as may be expected, the pressure of selecting poorer individuals results in a population that contains solutions with a very low number of 1's.

Using listeners and comparators

In this section, you will learn how to:

- to store operator invocation and improvement counts
- compare operators using lexicographic composition of comparators
- show operator statistics

You can monitor operator improvements by attaching listeners to operators. Such monitoring can be done in two ways:

- you can determine when an operator is invoked
- you can detect improvement by monitoring when an operator produces a solution

Make a copy of the example file `YourSolverHome/examples/src/tutorial/ea1max_listen_partial.cpp` and open this copy in your development environment. After filling in the blanks in each step in this section, you will have completed the example and you can compile and run the program.

In this example, you will show operator statistics; you define a small structure to hold these statistics for a particular operator.

Step 1 Stores invocation and improvement statistics for operators

Add the following code after the comment

`//Stores invocation and improvement statistics for operators`

```
struct OperatorStatistics {
  IloInt invocations;
  IloInt improvements;
  IloInt successes;
  OperatorStatistics() : invocations(0), improvements(0), successes(0) { }
};
```

The following procedure checks if the operator statistics record already exists and creates it if required. Each operator's statistics object are stored and retrieved using `setObject` and `getObject` respectively.

Step 2 Retrieve operator statistics (and create if necessary)

Add the following code after the comment

`//Retrieve operator statistics (and create if necessary)`

```
OperatorStatistics * GetOperatorStatistics(IloPoolOperator op) {
  OperatorStatistics * stat = (OperatorStatistics *)op.getObject();
  if (stat == 0) {
    stat = new (op.getEnv()) OperatorStatistics();
    op.setObject(stat);
  }
  return stat;
}
```

Objects of type `IloPoolOperator` define the basic genetic operators that will be used to modify the population, and it is upon these objects that monitoring will be done, and statistics gathered. However, to execute an operator it must be cast to a pool processor (`IloPoolProc`), albeit automatically most of the time. It is thus convenient to be able to have access to an operator which was used to create a processor. You perform this using a wrapping function and `setObject`. We also introduce a function which retrieves operator statistics give a processor.

Step 3 Set up processor to operator mapping

Add the following code after the comment

`// Setup the processor to operator mapping`

```
IloPoolProc BuildGAProcessor(IloPoolOperator op) {
  IloPoolProc proc(op);
  proc.setObject(op.getImpl());
  return proc;
}

OperatorStatistics * GetOperatorStatistics(IloPoolProc proc) {
  return GetOperatorStatistics((IloPoolOperatorI*)proc.getObject());
}
```

You are now in a position to define a listener object which will be called when a new solution is produced by an operator. In the listener, you update operator statistics:

- success—the operator produced a solution
- improvement—the operator produced a solution better than previous best solution

Listener objects can be defined via macros. You should pass dynamic information by reference or pointer so that you have the up-to-date information available when the listener is invoked; here the current generation.

Step 4 Defines a listener which records solution improvements

Add the following code after the comment

`//Defines a listener which records solution improvements`

```
ILOIIMLISTENER2(ImprovementListener, IloPoolOperator::SuccessEvent, event,
                IloSolution&, best,
                IloInt&, generation) {
  IloSolution newSolution = event.getSolution();
  IloPoolOperator op = event.getOperator();
  OperatorStatistics * stat = GetOperatorStatistics(op);
  stat->successes++;
  if (newSolution.isBetterThan(best)) {
    best.end();
    best = newSolution.makeClone(getEnv());
    cout << " IMPROVEMENT " << best.getObjectiveValue()
         << " BY " << op.getDisplayName()
         << " GENERATION " << generation
         << endl;
    stat->improvements++;
  }
}
```

The previous listener was to be called whenever a new solution was produced. You now create a listener which keeps track of when a particular genetic operator is invoked.

Step 5 Defines a listener which records operator invocations

Add the following code after the comment
`//Defines a listener which records operator invocations`

```
ILOIIMLISTENER0(OperatorListener, IloPoolOperator::InvocationEvent, event) {
  IloPoolOperator op = event.getOperator();
  OperatorStatistics * stat = GetOperatorStatistics(op);
  stat->invocations++;
}
```

It can be useful to rank or score genetic operators according to their past performance so that they can be more intelligently invoked in the future. For example, this can mean choosing more often operators which seem to work well. In this example, to sort genetic operators depending on their performance, you use two criteria:

- the improvement count
- the invocation count

To do so, you define a first an `IloComparator` which will compare operator improvement counts, followed by one to compare invocations. The final idea is to combine the two comparisons such that operators with higher improvement counts and lower invocation counts are preferred. You use the `ILOCOMPARATOR` macro to perform custom comparison. Two parameters, `left` and `right`, are always present and are the operators to be compared. This macro should return a true value if and only if `left` is strictly better than `right`. Here you see that an operator with more improvements is better than one with less.

Step 6 Define the operator improvement comparator

Add the following code after the comment
`//Define the operator improvement comparator`

```
ILOCOMPARATOR0(OperatorImprovementComparator, IloPoolProc, left, right) {
  OperatorStatistics * s1 = GetOperatorStatistics(left);
  OperatorStatistics * s2 = GetOperatorStatistics(right);
  return s1->improvements > s2->improvements;
}
```

You also define a second `IloComparator` devoted to comparing the numbers of operator invocations. Here, an operator which has been invoked less is deemed better than one which has been invoked more.

Step 7 Define the operator invocation comparator

Add the following code after the comment

```
//Define the operator invocation comparator

ILOCOMPARATOR0(OperatorInvocationComparator, IloPoolProc, left, right) {
  OperatorStatistics * s1 = GetOperatorStatistics(left);
  OperatorStatistics * s2 = GetOperatorStatistics(right);
  return s1->invocations < s2->invocations;
}
```

In the main routine you use the predefined tournament selector instead of the customized one.

Step 8 Use tournament selection to choose parents

Add the following code after the comment
`//Use tournament selection to choose parents`

```
    IloTournamentSelector<IloSolution, IloSolutionPool> tsel(
      env, 2, IloBestSolutionComparator(env)
    );
    IloPoolProc selector = IloSelectSolutions(env, tsel, IloTrue);
```

You attach the listeners before starting the generation loop. It is possible to attach listeners directly to the operator objects themselves. However, often a more convenient method is to add the listeners to the operator factory. These listeners will then be added to all operators produced by the factory after this point.

Step 9 Add the operator invocation listener

Add the following code after the comment
`// Add the operator invocation listener`

```
    factory.addListener(OperatorListener(env));
```

To record generated solutions, you add the improvement listener to the factory.

Step 10 Add the improvement listener

Add the following code after the comment `// Add the improvement listener`

```
    factory.addListener(ImprovementListener(env, best, generation));
```

This last listener will display a message each time the best solution found is improved upon.

Creation of the operators to use is done as before, with the exception that you make use of the `BuildGAProcessor` function to maintain a link from the processor at the operator on which it is based. This code is provided for you:

```
ops.add(BuildGAProcessor(factory.mutate(1.0 / problemSize, "mutate")));
ops.add(BuildGAProcessor(factory.uniformXover(0.5, "uXover")));
```

Once the generational loop exits, you display the goal statistics. The idea is to compare the performance of the operators according to two criteria. First, you compare the number of improvements which each operator provided. If the number of improvements is different for each operator, we know definitively which operator is considered better. If the number of improvements is the same, however, the number of invocations of each operator is examined. The operator which was invoked less is preferred. This type of hierarchical comparison is termed a lexicographical comparison and you will use it below.

Step 11 Create composite operator comparator

Add the following code after the comment
`//Create composite operator comparator`

```
IloComparator<IloPoolProc> opComparator = IloComposeLexical(
   OperatorImprovementComparator(env),
   OperatorInvocationComparator(env)
);
```

Using this comparator, you sort the operator pool.

Step 12 Sort the operator pool

Add the following code after the comment `//Sort the operator pool`

```
IloBool done;
do {
  done = IloTrue;
  for (IloInt i = 1; i < ops.getSize(); i++) {
    if (opComparator.isBetterThan(ops[i], ops[i-1])) {
      IloPoolProc tmp = ops[i-1]; ops[i-1] = ops[i]; ops[i] = tmp;
      done = IloFalse;
    }
  }
} while (!done);
```

Finally, you loop over the sorted processors in best-first manner.

Step 13 Loop over the processor pool and display statistics

Add the following code after the comment

`//Loop over the processor pool and display statistics`

```
    for (IloInt i = 0; i < ops.getSize(); i++) {
      IloPoolProc op = ops[i];
      OperatorStatistics* stat = GetOperatorStatistics(op);
      env.out() << " " << op.getDisplayName()
                << " INVOKE " << stat->invocations
                << " SUCCESS " << stat->successes
                << " IMPROVE " << stat->improvements
                << endl;
    }
```

You have now learned how to use listeners and comparators. In the next section, you will learn how to use pool evaluators.

Step 14 Compile and run the program

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

```
Problem size: 50
Population size: 50
Max generation: 60
GENERATION 0 WORST 17 BEST 34 AVERAGE 25.06
Optimum value: 50
 IMPROVEMENT 39 BY IloArrayUniformXover GENERATION 1
GENERATION 1 WORST 20 BEST 39 AVERAGE 27.18
GENERATION 2 WORST 23 BEST 35 AVERAGE 28.2
GENERATION 3 WORST 24 BEST 35 AVERAGE 29.42
GENERATION 4 WORST 25 BEST 38 AVERAGE 30.6
GENERATION 5 WORST 28 BEST 37 AVERAGE 32.06
GENERATION 6 WORST 28 BEST 39 AVERAGE 33.38
 IMPROVEMENT 40 BY IloArrayUniformXover GENERATION 7
GENERATION 7 WORST 26 BEST 40 AVERAGE 33.76
GENERATION 8 WORST 29 BEST 40 AVERAGE 34.86
 IMPROVEMENT 41 BY IloArrayUniformXover GENERATION 9
GENERATION 9 WORST 28 BEST 41 AVERAGE 35.44
GENERATION 10 WORST 31 BEST 40 AVERAGE 35.74
GENERATION 11 WORST 31 BEST 41 AVERAGE 36.76
 IMPROVEMENT 42 BY IloArrayUniformXover GENERATION 12
 IMPROVEMENT 43 BY IloArrayUniformXover GENERATION 12
GENERATION 12 WORST 30 BEST 43 AVERAGE 37.64
GENERATION 13 WORST 33 BEST 43 AVERAGE 38.64
GENERATION 14 WORST 36 BEST 43 AVERAGE 39.54
 IMPROVEMENT 45 BY IloArrayUniformXover GENERATION 15
GENERATION 15 WORST 36 BEST 45 AVERAGE 40.54
 IMPROVEMENT 47 BY IloArrayUniformXover GENERATION 16
GENERATION 16 WORST 37 BEST 47 AVERAGE 41.74
GENERATION 17 WORST 38 BEST 47 AVERAGE 42.42
GENERATION 18 WORST 40 BEST 47 AVERAGE 43.54
 IMPROVEMENT 48 BY IloArrayUniformXover GENERATION 19
GENERATION 19 WORST 41 BEST 48 AVERAGE 44.1
GENERATION 20 WORST 41 BEST 48 AVERAGE 44.72
GENERATION 21 WORST 40 BEST 48 AVERAGE 45.16
 IMPROVEMENT 49 BY IloArrayUniformXover GENERATION 22
GENERATION 22 WORST 42 BEST 49 AVERAGE 45.88
GENERATION 23 WORST 43 BEST 49 AVERAGE 46.32
GENERATION 24 WORST 43 BEST 48 AVERAGE 46.06
GENERATION 25 WORST 43 BEST 49 AVERAGE 46.56
GENERATION 26 WORST 42 BEST 49 AVERAGE 46.76
GENERATION 27 WORST 44 BEST 49 AVERAGE 46.96
 IMPROVEMENT 50 BY IloArrayUniformXover GENERATION 28
GENERATION 28 WORST 44 BEST 50 AVERAGE 47.44
OPTIMUM FOUND
 IloArrayUniformXover INVOKE 705 SUCCESS 705 IMPROVE 10
 IloArrayMutate INVOKE 695 SUCCESS 695 IMPROVE 0
```

The complete program is available in the `YourSolverHome/examples/src/ea1max_listen.cpp` file.

Review Exercises

1. Add an element to the lexicographical comparator so that preferred operators are ones with maximal improvement/invocation ratio, then maximum improvements, then minimum invocations. You will need to create a new comparator for the ratio and adjust the definition of the lexicographical comparator.

Suggested Answers

Exercise 1

A comparator is created as follows, taking care to avoid division by zero:

```
ILOCOMPARATOR0(OperatorImprovementRatioComparator, IloPoolProc, left, right) {
  OperatorStatistics * s1 = GetOperatorStatistics(left);
  OperatorStatistics * s2 = GetOperatorStatistics(right);

  // Avoid illegal computation of zero divided by zero
  if (s1->invocations == 0 || s2->invocations == 0)
    return IloFalse;

  return (IloNum)s1->improvements / s1->invocations >
         (IloNum)s2->improvements / s2->invocations;
}
```

The comparator is then used in the main body as shown below:

```
    IloComparator<IloPoolProc> opComparator = IloComposeLexical(
       OperatorImprovementRatioComparator(env),
       OperatorImprovementComparator(env),
       OperatorInvocationComparator(env)
    );
```

Using pool evaluators and partial replacement

In this section, you will learn how to:

- use a fitness-based parent selector
- alter parent values using a custom solution evaluator
- use a replacement goal to replace a fraction of the population with offspring

Make a copy of the example file `YourSolverHome/examples/src/tutorial/ea1max_values_partial.cpp` and open this copy in your development environment. After filling in the blanks in each step in this section, you will have completed the example.

In this example, you will replace only half of the population at each generation. You will also decide to select parents depending on solution objective values. To do this, you will use pool evaluators to do customized value computation.

You will follow the idea in this example that the probability of selecting a parent will be proportional to the square of the number of 1's (the objective value) of the parent. We introduce an evaluator which performs this function: given another evaluator as argument, it raises its evaluation to a specified power.

The following macro definition allows you to raise the values obtained from pools to a given power, and then normalizes these values so that they sum to unity. (Such a normalization can be useful for various purposes, not least displaying.) This evaluator will be used later to modify the evaluations of solutions. The evaluator calls the `update` method on the subordinate evaluator passed. Note that the macro receives two parameters, a subordinate evaluator, and a power. `ILOMULTIPLEEVALUATOR` must call the function `setEvaluation` for each solution of the input pool.

Step 1 Raise the result of an evaluator to a given power

Add the following code after the comment
`//Raise the result of an evaluator to a given power, and normalize`

```
ILOMULTIPLEEVALUATOR2(PowerEvaluator, IloSolution, IloSolutionPool, pool,
                      IloSolutionPoolEvaluator, eval,
                      IloNum, power) {
  eval.update(pool);
  IloNum sum = 0.0;
  for (IloSolutionPool::Iterator it1(pool); it1.ok(); ++it1)
    sum += pow(eval(*it1), power);
  for (IloSolutionPool::Iterator it2(pool); it2.ok(); ++it2)
    setEvaluation(*it2, pow(eval(*it2), power) / sum);
}
```

In the main body, when performing parent selection, we first declare an `IloSolutionPoolEvaluator` based on a simple `IloSolutionEvaluator` which collects objective values from solution pools.

Step 2 Evaluate objective values of solutions

Add the following code after the comment
`//Evaluate objective values of solutions`

```
    IloSolutionPoolEvaluator parentEvaluator(env, IloSolutionEvaluator(env));
```

This line constructs a *multiple evaluator* which evaluates each solution in an entire pool using the objective (which is evaluated using `IloSolutionEvaluator`). Then you use the `PowerEvaluator` constructor to square and normalize the objective value.

Step 3 Increases discrepancies between solutions

Add the following code after the comment
`//Increases discrepancies between solutions`

```
parentEvaluator = PowerEvaluator(env, parentEvaluator, 2.0);
```

You will now create a *parent selector* using `parentEvaluator`, and for this you will use a roulette wheel selector. In a roulette wheel selector, the probability for a given solution at index *i* to be selected is proportional to its evaluation. This selector is known as a roulette wheel selector as each parent evaluation can be viewed as a distance along the circumference of a roulette wheel which is spun to choose a parent.

Step 4 Selects parents depending on their probabilities

Add the following code after the comment
`//Selects parents depending on their probabilities`

```
IloRouletteWheelSelector<IloSolution,IloSolutionPool>
  rwsel(env, parentEvaluator);
IloPoolProc selector = IloSelectSolutions(env, rwsel, IloTrue);
```

The boolean parameter set to `IloTrue` indicates that when selecting more than one parent, the same solution cannot be selected more than once. You will see later that the content of this array will be computed using the `parentEvaluator` object as a preliminary step of the generation goal. Now, you will expand the repertoire of operators. Add one-point and two-point crossover operators, which exchange segments of parent solutions.

Step 5 Add 1 and 2 point crossover

Add the following code after the comment `//Add 1 and 2 point crossover`

```
ops.add(BuildGAProcessor(factory.onePointXover("1ptXover")));
ops.add(BuildGAProcessor(factory.twoPointXover("2ptXover")));
```

You also add operators which may not directly produce improvements but help maintain the population diversity by moving solution values around.

Step 6 Add transposition operators

Add the following code after the comment `//Add transposition operators`

```
ops.add(BuildGAProcessor(factory.swap("swap")));
ops.add(BuildGAProcessor(factory.translocate("transloc")));
```

In this example, you will replace only half of the current population with the offspring. You will thus generate only half the number of offspring as the size of the population, ensuring that the population will be kept constant in number.

Step 7 Create the reproduction goal

Add the following code after the comment `//Create the reproduction goal`

```
IloSolutionPool offspring(env, "offspring");
IloGoal reproduceGoal = IloExecuteProcessor(
  env, population >> selector >> applyOp(populationSize / 2) >> offspring
);
```

Here, the genetic operator is instructed to produce `populationSize / 2` solutions which are then placed in the offspring pool.

Since you use partial replacement instead of full replacement in this example, you need to define a way to choose which solutions get replaced or become “dead.” This can be done using a tournament selector (here, with tournament size 3) and using an `IloWorstSolutionComparator` which tends to choose bad solutions rather than good ones. The last argument of this selector's factory is a flag (in this case `IloTrue`) indicating that selected pool elements should appear only once in the selector's output pool (named `dead`). The selector is not allowed to select the same solution twice.

Step 8 Create the selector of individuals to remove

Add the following code after the comment
`//Create the selector of individuals to remove`

```
IloTournamentSelector<IloSolution,IloSolutionPool> deadSelector(
  env, 3, IloWorstSolutionComparator(env)
);
```

Now, you can create the replacement goal.

Step 9 Create the replacement goal

Add the following code after the comment `//Create the replacement goal`

```
IloGoal replacementGoal = IloExecuteProcessor(env,
  offspring >>
  IloReplaceSolutions(env, population, deadSelector) >>
  IloDestroyAll(env)
);
```

The `IloReplaceSolutions` processor takes its input (the offspring pool), and examines its size *n*. It then removes *n* solutions from population, using the `deadSelector` to select which ones and places them on the output. All solutions on the input are then added to the population. Finally, `IloDestroyAll` reclaims the memory used by the solutions in its input (that is, those selected for destruction by `IloReplaceSolutions`).

Finally, you have to add the code that will compute solution values used for selecting parents each generation. This is done by using the `IloUpdate` goal which will perform the computation specified by `PowerEvaluator` on values stored in solutions for the `count` solver variable.

Step 10 Compute parent probabilities before each generation

Add the following code after the comment
`//Compute parent probabilites before each generation`

```
generationGoal = IloUpdate(env, parentEvaluator, population)
              && generationGoal;
```

You have now learned how to use pool evaluators.

Step 11 Compile and run the program

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

```
Problem size: 50
Population size: 50
Max generation: 60
GENERATION 0 WORST 17 BEST 34 AVERAGE 25.06
Optimum value: 50
GENERATION 1 WORST 20 BEST 34 AVERAGE 26.36
 IMPROVEMENT 35 BY IloArrayOnePointXover GENERATION 2
GENERATION 2 WORST 21 BEST 35 AVERAGE 28.26
GENERATION 3 WORST 21 BEST 35 AVERAGE 28.98
GENERATION 4 WORST 23 BEST 35 AVERAGE 30.14
GENERATION 5 WORST 24 BEST 35 AVERAGE 30.9
 IMPROVEMENT 38 BY IloArrayOnePointXover GENERATION 6
GENERATION 6 WORST 23 BEST 38 AVERAGE 31.1
GENERATION 7 WORST 27 BEST 38 AVERAGE 32.44
```

```
 IMPROVEMENT 40 BY IloArrayOnePointXover GENERATION 8
GENERATION 8 WORST 31 BEST 40 AVERAGE 33.68
GENERATION 9 WORST 31 BEST 40 AVERAGE 34.46
GENERATION 10 WORST 31 BEST 40 AVERAGE 35.02
 IMPROVEMENT 41 BY IloArrayTwoPointXover GENERATION 11
GENERATION 11 WORST 31 BEST 41 AVERAGE 35.64
 IMPROVEMENT 42 BY IloArrayUniformXover GENERATION 12
GENERATION 12 WORST 30 BEST 42 AVERAGE 36.2
GENERATION 13 WORST 33 BEST 42 AVERAGE 37.4
GENERATION 14 WORST 31 BEST 42 AVERAGE 37.48
GENERATION 15 WORST 32 BEST 42 AVERAGE 37.7
 IMPROVEMENT 44 BY IloArrayUniformXover GENERATION 16
GENERATION 16 WORST 34 BEST 44 AVERAGE 39
GENERATION 17 WORST 36 BEST 44 AVERAGE 39.28
GENERATION 18 WORST 36 BEST 44 AVERAGE 39.88
GENERATION 19 WORST 36 BEST 44 AVERAGE 40.46
GENERATION 20 WORST 36 BEST 44 AVERAGE 41.38
GENERATION 21 WORST 36 BEST 44 AVERAGE 42.06
 IMPROVEMENT 45 BY IloArrayTwoPointXover GENERATION 22
GENERATION 22 WORST 39 BEST 45 AVERAGE 42.4
 IMPROVEMENT 47 BY IloArrayUniformXover GENERATION 23
GENERATION 23 WORST 39 BEST 47 AVERAGE 42.9
GENERATION 24 WORST 39 BEST 47 AVERAGE 43.4
GENERATION 25 WORST 39 BEST 47 AVERAGE 43.62
GENERATION 26 WORST 40 BEST 47 AVERAGE 43.94
GENERATION 27 WORST 40 BEST 47 AVERAGE 43.86
 IMPROVEMENT 48 BY IloArrayTwoPointXover GENERATION 28
GENERATION 28 WORST 39 BEST 48 AVERAGE 44.08
GENERATION 29 WORST 39 BEST 48 AVERAGE 44.66
GENERATION 30 WORST 39 BEST 48 AVERAGE 45.24
GENERATION 31 WORST 38 BEST 48 AVERAGE 45.34
GENERATION 32 WORST 41 BEST 48 AVERAGE 45.82
GENERATION 33 WORST 43 BEST 48 AVERAGE 46.26
 IMPROVEMENT 49 BY IloArrayTwoPointXover GENERATION 34
GENERATION 34 WORST 42 BEST 49 AVERAGE 46.48
GENERATION 35 WORST 44 BEST 49 AVERAGE 46.72
GENERATION 36 WORST 44 BEST 49 AVERAGE 46.86
GENERATION 37 WORST 45 BEST 49 AVERAGE 47.14
GENERATION 38 WORST 45 BEST 49 AVERAGE 47.36
 IMPROVEMENT 50 BY IloArrayOnePointXover GENERATION 39
GENERATION 39 WORST 45 BEST 50 AVERAGE 47.52
OPTIMUM FOUND
 IloArrayOnePointXover INVOKE 150 SUCCESS 150 IMPROVE 4
 IloArrayTwoPointXover INVOKE 181 SUCCESS 181 IMPROVE 4
 IloArrayUniformXover INVOKE 154 SUCCESS 154 IMPROVE 3
 IloArraySwap INVOKE 153 SUCCESS 153 IMPROVE 0
 IloArrayTranslocate INVOKE 156 SUCCESS 156 IMPROVE 0
 IloArrayMutate INVOKE 181 SUCCESS 181 IMPROVE 0
```

The complete program is available in the `YourSolverHome/examples/src/ea1max_values.cpp` file.

Review Exercises

1. Vary the value of the power used from 2.0. For example, try values of 10.0 and 0.1. Increasing the value will tend to place more emphasis on selecting fitter individuals rather than less fit ones; it will increase the selection pressure. Decreasing the value will tend to decrease selection pressure. Observe how quickly the optimal solution is found.

Suggested Answers

Exercise 1

You should observe that increasing the selection pressure causes the optimal solution to be found more quickly. However, this can have a downside when more complex problems are considered. Suboptimal solutions can dominate any potentially promising (but currently inferior) solutions which emerge, resulting in a stagnation of the population. Genetic algorithms try to find a balance between diversification and intensification of the population by their choice of selection pressure.

Also note that even when the selection pressure is very low (use of a low power), solutions are still found quickly as you continue to perform partial replacement. This means that even if all solutions have a roughly equal chance of participating in a genetic operation, poorer solutions are more likely to be replaced by offspring. This effect has a tendency to increase solution quality.

C H A P T E R

31

Bin Packing Using EA

This lesson describes how you can use evolutionary algorithms to solve a classic optimization problem: that of bin packing. The solution method uses a *direct encoding*, meaning that the genetic operators used operate directly on the natural problem representation. (Contrast this with the indirect encoding used in the next lesson, Chapter 32, *Car Sequencing Using EA*.) The direct encoding normally only works well when tailored genetic operators are used and this lesson will show you how you can write such a genetic operator which tries to take the best features from parents, while discarding the worst ones.

Make a copy of the example file `YourSolverHome/examples/src/tutorial/eabinpack_partial.cpp` and open this copy 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 section. At the end of the section, you will have completed the example and you can compile and run the program.

Problem description

The bin packing problem asks the question: given a set of items, each with an associated weight w_i, and a set of bins, each of capacity C, how many bins are needed in order to pack all the items where no bin can exceed its capacity and all items are indivisible (cannot be split and packed into more than one bin)? This problem is at the root of other seemingly different problems, like assigning television advertisements to breaks between shows, or

assigning lots of work to engineers. For a more detailed explanation of this problem, see Chapter 8, *Combining Constraints: Bin Packing*.

Problem representation: Model

The model of the bin packing problem is relatively straightforward to specify and makes use of the `IloPack` constraint. The two pieces of information needed to build the model are first, the bin capacity *C*, and second, the weights of the items to be packed into the bins. As model construction is so easy for this example, the construction of the model is embedded in the genetic algorithm solving function `SolveWithGA`. The code to construct the model is provided for you and is replicated here:

```
// The bin packing model
IloEnv env = weight.getEnv();
IloModel mdl(env);
IloInt n = weight.getSize();
IloInt totalWeight = IloSum(weight);
IloInt lb = (totalWeight + cap - 1) / cap;
IloInt numBins = lb * 11 / 9 + 5;
IloIntVarArray where(env, n, 0, numBins);
IloIntVarArray load(env, numBins, 0, cap);
IloIntVar used(env, lb, numBins);
mdl.add(used == 1 + IloMax(where));
mdl.add(IloPack(env, load, where, weight));
```

The inputs are the bin capacity `cap` and the vector of weights `weight`. The model is based upon a table of integer variables `where` which are the decision variables. The value of `where[i]` contains the bin where item *i* is placed in any solution to the problem. Another array `load` is constructed, one element of this array per available bin; `load[j]` holds the total weight contained in bin *j*. Finally, a variable is created representing the number of bins used; this will later be used as an optimization criterion. The model assumes that all of the used bins will be those with the smallest indices, and so the number of bins used is one greater than the greatest bin index found in the `where` array. The packing constraint `IloPack` takes the loads, item locations, and item weights as parameters and enforces the bin capacity constraints according to the assignments of items to them.

You will notice that a maximum number of bins `numBins` available for use is calculated based on a simple lower bound. The lower bound is simply the total weight divided by the capacity, rounded up. `numBins` is calculated from the lower bound via a formula for the worst case performance of the *decreasing first fit* method, a well-known packing method that is used here. This means that, so long as decreasing first fit is used, the number of resulting bins will never be more than `(lb * 11 / 9 + 5)`. However, normally the solutions found are much closer to the simple lower bound, and typically any reasonable factor above `lb` is suitable as the number of bins available.

Problem solving: Search goal

The previous section mentioned the use of the *decreasing best fit* method of bin packing. In this section, you will implement this method in an Solver goal. First of all, you will use the algorithm below:

1. Assume that the items are sorted by non-increasing weight and that no bins are present.
2. Take the first (*i.e.* heaviest) unpacked item. If all items are already packed then stop.
3. Pack this item into the first bin that will accommodate it without violating its capacity. If no such bin exists, create a new empty bin of capacity *C*, and place the item there.
4. Go to 2.

As you can see, this algorithm is extremely simple. Items are considered, heaviest first, and packed in the first possible bin. New bins are added when they are needed.

To take advantage of constraint programming, the Solver version of this method (a goal) will be slightly more complicated. The difference will be that decisions can be backtracked. So, for example, if item *i* were packed into bin *b*, then on backtracking, the same item is forbidden from being packed in bin *b*. This means that the goal is much more robust as it can recover from errors and has the potential to perform a complete search over the packing problem.

A backtracking goal is required in this example, as here, advantage is taken of constraint programming to constrain the genetic algorithm to produce ‘‘good’’ solutions. This means constraining the number of available bins when generating a new solution. In this case an *error*, as described above, is the production of a packing that uses more bins than desired. In such a case, a backtrack occurs to change a previous decision in the search of a solution respecting the desired number of bins.

Step 1 Write the packing goal

You will now write the packing goal. Add the following code after the comment

```
// A simple goal to pack bins
```

```
ILCGOAL3(Pack, IlcIntVarArray, load,
               IlcIntVarArray, where,
               IlcIntArray, weight) {
  IlcGoal g;
  IlcInt i = IlcChooseFirstUnboundInt(where);
  if (i >= 0) {
    IlcInt b = where[i].getMin();
    IlcConstraint ct = where[i] == b;
    IlcGoal pack = load[b].getMin() == 0 ? ct: IlcOr(ct, !ct);
    g = IlcAnd(pack, this);
  }
  return g;
}
```

It is assumed here that the items are sorted in non-increasing order of weight. This example does this by ordering the weights in this way in the input file.

The goal follows the algorithm straightforwardly, except in the creation of choice points to allow backtracking. The test of the load of the bin against zero is used to ensure that only *one* empty bin—the first one—is considered for packing. That is, the goal does not backtrack on the choice to put an item in an empty bin as all choices of placing the item into a non-empty bin by this point have been exhausted, and all empty bins are equivalent.

Problem solving: Initial population

After creating the model of the problem, the next step in creating an evolutionary or genetic algorithm is normally the building of a *solution prototype* defining the decision variables of the problem and the objective function. This solution prototype will then be used by the genetic algorithm as a template for the creation of new solutions.

Step 2 Create the solution prototype

To create the solution prototype add the following code after the comment

```
// Solution prototype
```

```
IloSolution prototype(env);
prototype.add(where);
prototype.add(load);
prototype.add(used);
prototype.add(IloMinimize(env, used));
```

This code creates the prototype and adds the variables that are used during the execution of the genetic algorithm. The objective (minimize the number of bins used) is also added to allow solutions to be compared for quality.

You can now create the initial population. This population needs to be reasonably diverse and one simple way of doing this is to randomize it. Given the existing decreasing first fit packing goal, the `IloRandomPerturbation` goal modifier creates a randomized version which swaps the left and right branches of choice points with a certain probability.

Step 3 Create the initial population

Add the following code after the comment `// Create initial population`

```
IloSolutionPool population(env);
IloGoal packGoal = Pack(env, load, where, weight);
packGoal = IloRandomPerturbation(env, packGoal, 50.0 / (50 + n));
IloPoolProc src = IloSource(env, packGoal, prototype);
IloPoolProc create = src(popSize) >> population;
IloSolver solver(mdl);
solver.solve(IloExecuteProcessor(env, create));
```

A population is created, together with the decreasing first fit packing goal, which is then perturbed, each decision being inverted with probability $50/(50+n)$ where n is the number of items to pack. A probability based on the reciprocal of the number of items to pack ensures that the number of inversions is bounded. The extra constant on the denominator ensures that the probability is not too high for small n (that is, small problems). There is nothing particularly magic about this formula; many other formulas will work just as well.

The population is built using the `IloSource` operator which takes a goal and a solution prototype, invoking the goal to produce the solution. When cast to a pool processor (implicitly here), the prototype tells the resulting processor what type of solution to produce. Finally, `popSize` solutions are requested of `IloSource` through the parentheses operator, the resultant solutions being placed in `population`. The `create` processor is then cast to a goal via `IloExecuteProcessor` and executed by the solver, which is in turn created using the model.

Problem solving: Genetic operators

The initial population being successfully created, evolution of the solutions in that population can then begin. This evolution is brought about by the application of genetic operators such as *mutation* and *crossover* to members of the population. Additionally, here the power and flexibility of constraint programming will be used to inhibit the creation of poor quality solutions. The latter is done by adding constraints to the cost variable before the application of a genetic operator. The idea here is to say that with probability p, the new solution produced is constrained to be better than the worst of the parents. With probability

1-p, the new solution is no worse than the worst parent. (For a mutation operator where there is only one parent, read "parent" for "worst parent.") This means that children can never be worse than their worst parent in this example. This leads to an intensified search which could in general be damaging as it limits exploration. However, in the case of bin packing, the cost function is not very fine-grained and there are large numbers of different solutions at the same cost. Hence, in this case, such a constraint is not particularly limiting in terms of exploration.

You will write an operator that performs this cost-constraining function.

Step 4 Create the operator that constrains cost

Add the following code after the comment: `// Improve on worst in input pool`

```
ILOIIMOP2(ImproveOn, 1, IloIntVar, used, IlcFloat, p) {
  IloInt limit = (IloInt)getInputPool().getWorstSolution().getValue(used);
  IloSolver solver(getSolver());
  solver.setMax(used, limit - (solver.getRandom().getFloat() < p));
  return 0;
}
```

This operator is defined by the `ILCIIMOP2` macro and takes six parameters. The first parameter is the name of the operator to be created. The second is the minimum number of solutions that the operator requires as input; one solution in this case. Next come the two optional parameters with their types, as for the definition of a goal using the `ILCGOAL` macro. In this case, these are the cost variable (the number of bins used), and a probability of improvement. Contained in the input pool are the parents of the operator. The above code simply sets an upper bound on the number of bins used which is *limit-1* with probability *p*, and *limit* with probability *1-p.*

You will now move on to the creation of the genetic operators themselves. This lesson creates two genetic operators: a mutation operator and a crossover operator, based on a generic combination operator which can work with arbitrarily many parents. You will write this operator later, but first you will begin with the code which uses it.

The genetic operators used will correspond to a particular form. They will first limit the cost variable using `ImproveOn` as described above and then enter the operator proper. As you will see later, the operator may not build a complete solution, and so a third part—a completion operator—is required which extends this partial solution to a complete one. This completion operator is none other than the packing goal `Pack`. The code to build a genetic operator from its three basic components (constrain, mutation or crossover, completion) will be created using the `&&` operator over pool operators. Finally, the complete genetic operators that are built will be limited in their execution to avoid very long times being spent on potentially fruitless search.

Step 5 Create the pool processor

Add the following code after the comment: `// Genetic operators`

```
IloPoolOperator improve = ImproveOn(env, used, 0.75);
IloSearchLimit limit(IloFailLimit(env, 1000));
IloPoolOperator mutop = PackingOperator(env, 1, load, where, weight, used);
IloPoolProc mut = IloLimitOperator(env, improve && mutop && packGoal, limit);
IloPoolProc breed = mut;
if (popSize > 1) {
  IloPoolProcArray ops(env);
  ops.add(mut);
  IloPoolOperator xoop = PackingOperator(env, 2, load, where, weight, used);
  IloPoolProc xo = IloLimitOperator(env, improve && xoop && packGoal, limit);
  ops.add(xo);
  IloExplicitEvaluator<IloPoolProc> operatorProbs(env);
  operatorProbs.setEvaluation(mut, mutationProb);
  operatorProbs.setEvaluation(xo, 1.0 - mutationProb);
  IloRouletteWheelSelector<IloPoolProc, IloPoolProcArray> opSel(
    env, operatorProbs
  );
  breed = IloSelectProcessor(env, ops, opSel);
}
```

There are a few points to note in the above code. The goal of the code is to produce a pool processor `breed` capable of taking an input and producing offspring. How this is done depends upon the size of the population; if it has only one member it is via mutation, otherwise it is via a combination of mutation and crossover (combination). The genetic operator `PackingOperator`, which you will define later, is used to construct both the mutation and crossover operators. The second parameter of this operator is the number of parents to operate over, and it is this parameter that differentiates the mutation and crossover operators. Note that both the mutation and crossover operators are subjected to a modification via the `&&` operator on processors and via `IloLimitOperator`. These modifications result in operators limited in number of fails and which constrain the cost first (with probability 0.75) and complete the solution after the execution of the main operator. Finally, when both mutation and crossover operators are used, a *roulette wheel selector* is used to select between them for each new child generated. A roulette wheel selector uses a table of probabilities (or equally non-negative weights, which are normalized internally to give probabilities) to select one item from the given pool. After selection, `IloApplyProcessor` executes the processor found in the pool of processors given as parameter.

Problem solving: Genetic packing operator

In the previous section, `PackingOperator` was introduced as the genetic operator used for both mutation and crossover. Here, you will see how you can implement such a genetic

operator and also learn how to make such an operator work well by favoring good features in the parents.

The first idea of the genetic operator is to take not single objects, but the contents of *entire bins* from the parents for inclusion in the child. The advantage of this is that groups of objects that go well together can be maintained in the child. The second idea is that only ‘‘good’’ bins are chosen to go into the child; a good bin being one that is filled to the average required load—an idea that will be fully introduced in what follows. The third idea is to pack a randomized number of bins in the operator itself, leaving a randomized number of bins to be packed by the decreasing first fit packing goal; this increases the robustness and exploration capability of the search.

As before, an algorithm will be presented first:

1. Let B be the number of bins available for use to produce the child, W be the total object weight to be packed, and $r=W/B$ be the mean required load of each bin.
2. Choose a number of bins f to fill in the child chosen from 0 to b.
3. Begin with the current bin c equal to 0, and a number of tries t equal to 0.
4. If $c = f$ or $t = limit$, then stop.
5. Choose a random parent p and a random bin b within p.
6. If no items in b have already been placed in the child, and the load of b is at least r, then place all items in b in bin c of the child, increase c, and reset the number of tries t to zero. Otherwise, increase the number of tries t.
7. Go to 4.

The algorithm picks bins from the parents randomly, packing those which meet the criteria, until the desired number of bins have been packed, or a certain number of tries have been exhausted. Two criteria are used. The first has to do with the feasibility of packing: no bin can be included which contains an item already packed in the child, as no item can be packed in more than one bin. The second has to do with the quality of the bins chosen from the parents; these bins must meet the average load requirement to produce a solution in the number of bins required. That is, only bins which are ‘‘good enough’’ are selected.

You are now ready to write the code of the genetic operator. Steps 1-4 of the above algorithm are already coded. You will find the code below:

```
// A genetic operator over bins
ILOIIMOP5(PackingOperator, numParents,
          IloInt, numParents,
          IloIntVarArray, load,
          IloIntVarArray, where,
          IloIntArray, weight,
          IloIntVar, used) {
  IloSolver solver = getSolver();
  IloInt numItems = weight.getSize();
  IloInt numBins = solver.getMax(used);
  IloSolutionPool parents = getInputPool();
  IloInt numParents = parents.getSize();
  IloInt reqdMeanLoad = (IloSum(weight) + numBins - 1) / numBins;
  IlcRandom rnd = solver.getRandom();
  IloInt targetBin = rnd.getInt(numBins);
  IloInt currentBin = 0;
  IloInt tries = 0;

  while (currentBin < targetBin && tries < numParents * numBins) {
```

The above code is relatively simple and closely follows algorithmic steps 1-4. However, it is important to point out the parameters of the `ILCIIMOP5` macro. Recall that the first two parameters are the name of the operator and the minimum number of parents required. For the `ImproveOn` operator, this second parameter is the constant 1. However, this parameter can be any general expression, and here the size is equal to `numParents`, the first optional parameter of the operator. In this way, you can create operators which use a variable number of parents, specified when the operator is constructed.

The above code chooses to set the limit on the number of tries equal to the product of the number of parents and the number of available bins.

You will now add the remaining code which chooses a parent and bin, and inserts its contents into the child.

Step 6 Pack a single bin

Insert the following code after the comment: `// Pack a single bin`

```
IloSolution p = parents.getSolution(rnd.getInt(numParents));
IloInt srcBin = rnd.getInt(p.getValue(used));
if (p.getValue(load[srcBin]) >= reqdMeanLoad) {
  IloBool fit = IloTrue;
  for (IlcInt i = 0; fit && i < numItems; i++) {
    if (p.getValue(where[i]) == srcBin)
      fit = solver.getIntVar(where[i]).isInDomain(currentBin);
  }
  if (fit) {
    for (IlcInt i = 0; i < numItems; i++) {
      if (p.getValue(where[i]) == srcBin)
        solver.setValue(where[i], currentBin);
    }
    tries = 0;
    currentBin++;
  }
}
tries++;
```

Again, this code closely follows the algorithm. First, a random parent and bin from that parent are selected. Then, given that the load of the chosen bin is sufficient, the code looks to see if all items from the selected bin can be placed in the current bin in the child. This is done by examination of the domains of the variables of the child. (Recall that the purpose of a genetic operator is to take solutions—instances of `IloSolution`—as input, and to produce as output an instantiation of Solver variables which is the child solution.) This examination is equivalent to testing if the objects from the chosen bin have already been scheduled in the child because if they have been scheduled, they will have a `where` variable bound to a value strictly less than `currentBin`. Assuming all objects can go in `currentBin`, they are placed there via `solver.setValue()`, the current bin is moved on, and the number of tries is reset.

Problem solving: Putting it together

Nearly everything is in place to create a genetic algorithm; there is an initial population, and a processor to produce offspring based on a combination of improvement, mutation or crossover, and solution completion via decreasing first fit. All that remains is to decide on a parent selection strategy, a replacement strategy whereby new offspring will replace current population members, and to create a goal that will perform one complete generation of the genetic algorithm.

Here, a random parent selection strategy is chosen, together with a *full replacement* strategy. The latter produces as many offspring as there are population members, and completely replaces the population by the offspring.

Step 7 Create the single generation goal

All this is done fairly concisely by the following code which you add after the comment

```
// Single generation goal
```

```
IloSolutionPool offspring(env);
IloPoolProc rndSel = IloSelectSolutions(env,
                       IloRandomSelector<IloSolution, IloSolutionPool>(env)
                     );
IloPoolProc gen = population >> rndSel >> breed(popSize) >> offspring;
IloGoal genGoal = IloExecuteProcessor(env, gen)
               && IloExecuteProcessor(env, population >> IloDestroyAll(env))
               && IloExecuteProcessor(env, offspring >> population);
```

A pool to contain the offspring is created, as is a random selector for selection of parents. Then a processor is constructed which performs selection and breeding and places the result in the offspring pool. The breeding operator is instructed to produce `popSize` solutions via the function call operator. The goal to run a generation of the genetic algorithm is then constructed by chaining `IloExecuteProcessor` functions. First, `gen` is executed which produces the offspring, the current population is destroyed, and finally the offspring are placed in the population pool.

The code of the genetic algorithm loop and display of the resulting solution is provided for you.

Complete program

The complete program follows and can be viewed on-line in `YourSolverHome\examples\src\eabinpack.cpp`.

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


#include <ilsolver/iim.h>

ILOSTLBEGIN

// Read in a bin packing problem
IloBool ReadProblem(const char * fname, IloInt& cap, IloIntArray& weight) {
  ifstream is(fname);
  if (!is.good())
    return IloFalse;
  IloInt n;
```

```
  is >> n >> cap;
  for (IloInt i = 0; i < n; i++) {
    IloInt w;
    is >> w;
    weight.add(w);
  }
  is.close();
  return IloTrue;
}

// A simple goal to pack bins
ILCGOAL3(Pack, IlcIntVarArray, load,
               IlcIntVarArray, where,
               IlcIntArray, weight) {
  IlcGoal g;
  IlcInt i = IlcChooseFirstUnboundInt(where);
  if (i >= 0) {
    IlcInt b = where[i].getMin();
    IlcConstraint ct = where[i] == b;
    IlcGoal pack = load[b].getMin() == 0 ? ct: IlcOr(ct, !ct);
    g = IlcAnd(pack, this);
  }
  return g;
}

// The packing goal wrapper
ILOCPGOALWRAPPER3(Pack, solver,
                  IloIntVarArray, load,
                  IloIntVarArray, where,
                  IloIntArray, weight) {
  return Pack(solver,
              solver.getIntVarArray(load),
              solver.getIntVarArray(where),
              solver.getIntArray(weight));
}

// A genetic operator over bins
ILOIIMOP5(PackingOperator, numParents,
          IloInt, numParents,
          IloIntVarArray, load,
          IloIntVarArray, where,
          IloIntArray, weight,
          IloIntVar, used) {
  IloSolver solver = getSolver();
  IloInt numItems = weight.getSize();
  IloInt numBins = solver.getMax(used);
  IloSolutionPool parents = getInputPool();
  IloInt numParents = parents.getSize();
  IloInt reqdMeanLoad = (IloSum(weight) + numBins - 1) / numBins;
  IlcRandom rnd = solver.getRandom();
  IloInt targetBin = rnd.getInt(numBins);
  IloInt currentBin = 0;
  IloInt tries = 0;

  while (currentBin < targetBin && tries < numParents * numBins) {
    // Pack a single bin
    IloSolution p = parents.getSolution(rnd.getInt(numParents));
    IloInt srcBin = rnd.getInt(p.getValue(used));
```

```
    if (p.getValue(load[srcBin]) >= reqdMeanLoad) {
      IloBool fit = IloTrue;
      for (IlcInt i = 0; fit && i < numItems; i++) {
        if (p.getValue(where[i]) == srcBin)
          fit = solver.getIntVar(where[i]).isInDomain(currentBin);
      }
      if (fit) {
        for (IlcInt i = 0; i < numItems; i++) {
          if (p.getValue(where[i]) == srcBin)
            solver.setValue(where[i], currentBin);
        }
        tries = 0;
        currentBin++;
      }
    }
    tries++;
  }
  return 0;
}

// Improve on worst in input pool
ILOIIMOP2(ImproveOn, 1, IloIntVar, used, IlcFloat, p) {
  IloInt limit = (IloInt)getInputPool().getWorstSolution().getValue(used);
  IloSolver solver(getSolver());
  solver.setMax(used, limit - (solver.getRandom().getFloat() < p));
  return 0;
}

// Main GA Solving body
void SolveWithGA(IloInt cap,
                 IloIntArray weight,
                 IloInt popSize,
                 IloInt maxGen,
                 IloNum mutationProb) {

  // The bin packing model
  IloEnv env = weight.getEnv();
  IloModel mdl(env);
  IloInt n = weight.getSize();
  IloInt totalWeight = IloSum(weight);
  IloInt lb = (totalWeight + cap - 1) / cap;
  IloInt numBins = lb * 11 / 9 + 5;
  IloIntVarArray where(env, n, 0, numBins);
  IloIntVarArray load(env, numBins, 0, cap);
  IloIntVar used(env, lb, numBins);
  mdl.add(used == 1 + IloMax(where));
  mdl.add(IloPack(env, load, where, weight));

  // Solution prototype
  IloSolution prototype(env);
  prototype.add(where);
  prototype.add(load);
  prototype.add(used);
  prototype.add(IloMinimize(env, used));

  // Create initial population
  IloSolutionPool population(env);
  IloGoal packGoal = Pack(env, load, where, weight);
```

```
packGoal = IloRandomPerturbation(env, packGoal, 50.0 / (50 + n));
IloPoolProc src = IloSource(env, packGoal, prototype);
IloPoolProc create = src(popSize) >> population;
IloSolver solver(mdl);
solver.solve(IloExecuteProcessor(env, create));

// Genetic operators
IloPoolOperator improve = ImproveOn(env, used, 0.75);
IloSearchLimit limit(IloFailLimit(env, 1000));
IloPoolOperator mutop = PackingOperator(env, 1, load, where, weight, used);
IloPoolProc mut = IloLimitOperator(env, improve && mutop && packGoal, limit);
IloPoolProc breed = mut;
if (popSize > 1) {
  IloPoolProcArray ops(env);
  ops.add(mut);
  IloPoolOperator xoop = PackingOperator(env, 2, load, where, weight, used);
  IloPoolProc xo = IloLimitOperator(env, improve && xoop && packGoal, limit);
  ops.add(xo);
  IloExplicitEvaluator<IloPoolProc> operatorProbs(env);
  operatorProbs.setEvaluation(mut, mutationProb);
  operatorProbs.setEvaluation(xo, 1.0 - mutationProb);
  IloRouletteWheelSelector<IloPoolProc, IloPoolProcArray> opSel(
    env, operatorProbs
  );
  breed = IloSelectProcessor(env, ops, opSel);
}

// Single generation goal
IloSolutionPool offspring(env);
IloPoolProc rndSel = IloSelectSolutions(env,
                       IloRandomSelector<IloSolution, IloSolutionPool>(env)
                     );
IloPoolProc gen = population >> rndSel >> breed(popSize) >> offspring;
IloGoal genGoal = IloExecuteProcessor(env, gen)
               && IloExecuteProcessor(env, population >> IloDestroyAll(env))
               && IloExecuteProcessor(env, offspring >> population);

// Genetic algorithm loop
IloInt best = population.getBestSolution().getValue(used);
IloInt i;
for (i = 0; i <= maxGen && best > lb; i++) {
  env.out() << "Generation " << i << ": ";
  env.out() << best << " - "
            << population.getWorstSolution().getValue(used) << endl;
  solver.solve(genGoal);
  best = population.getBestSolution().getValue(used);
}

// Display best
solver.solve(IloRestoreSolution(env, population.getBestSolution()));
IloInt binsUsed = (IloInt)solver.getValue(used);
env.out() << "Best solution uses " << binsUsed << " bins" << endl;
if (binsUsed == lb)
  env.out() << "Solution is optimal" << endl;
else
  env.out() << "Solution is within "
            << binsUsed - lb << " of optimal" << endl;
for (i = 0; i < binsUsed; i++) {
```

```
    env.out() << "Bin " << i + 1 << " of weight "
              << solver.getValue(load[i]) << ": ";
    for (IloInt j = 0; j < n; j++) {
      if (solver.getValue(where[j]) == i)
        env.out() << weight[j] << " ";
    }
    env.out() << endl;
  }
}

// Main body
int main(int argc, char * argv[]) {
  IloEnv env;
  try {
    const char * fname = "../../../examples/data/binpack1.dat";
    IloInt maxGen = 100;
    IloInt popSize = 30;
    IloNum mutationProb = 0.75;
    if (argc > 1)
      fname = argv[1];
    if (argc > 2)
      maxGen = atoi(argv[2]);
    if (argc > 3)
      popSize = atoi(argv[3]);
    if (argc > 4)
      mutationProb = atof(argv[4]);
    IloInt cap;
    IloIntArray weight(env);
    if (ReadProblem(fname, cap, weight))
      SolveWithGA(cap, weight, popSize, maxGen, mutationProb);
    else
      env.out() << "Could not open " << fname << endl;
  } catch (IloException & ex) {
    env.out() << "Caught: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Output

Execution of the program results in the following output:

```
Generation 0: 52 - 54
Best solution uses 51 bins
Solution is optimal
Bin 1 of weight 150: 83 67
Bin 2 of weight 150: 96 54
Bin 3 of weight 150: 61 54 35
Bin 4 of weight 150: 41 40 39 30
Bin 5 of weight 150: 77 73
Bin 6 of weight 148: 92 56
Bin 7 of weight 150: 83 67
```

```
Bin 8 of weight 150: 81 69
Bin 9 of weight 150: 82 68
Bin 10 of weight 148: 100 48
Bin 11 of weight 150: 93 57
Bin 12 of weight 149: 82 67
Bin 13 of weight 150: 94 56
Bin 14 of weight 150: 76 74
Bin 15 of weight 150: 82 68
Bin 16 of weight 149: 100 49
Bin 17 of weight 150: 85 65
Bin 18 of weight 150: 78 72
Bin 19 of weight 150: 97 53
Bin 20 of weight 148: 90 58
Bin 21 of weight 150: 83 67
Bin 22 of weight 150: 90 60
Bin 23 of weight 150: 79 71
Bin 24 of weight 150: 88 62
Bin 25 of weight 150: 86 64
Bin 26 of weight 150: 77 73
Bin 27 of weight 150: 81 69
Bin 28 of weight 150: 100 50
Bin 29 of weight 148: 82 66
Bin 30 of weight 150: 77 73
Bin 31 of weight 150: 84 66
Bin 32 of weight 149: 79 70
Bin 33 of weight 150: 53 53 44
Bin 34 of weight 150: 81 69
Bin 35 of weight 150: 76 74
Bin 36 of weight 150: 96 54
Bin 37 of weight 146: 92 54
Bin 38 of weight 150: 98 52
Bin 39 of weight 144: 90 54
Bin 40 of weight 146: 80 66
Bin 41 of weight 142: 78 64
Bin 42 of weight 127: 66 61
Bin 43 of weight 146: 73 73
Bin 44 of weight 150: 63 47 40
Bin 45 of weight 147: 61 48 38
Bin 46 of weight 149: 47 42 30 30
Bin 47 of weight 150: 43 42 34 31
Bin 48 of weight 150: 42 37 37 34
Bin 49 of weight 146: 43 35 34 34
Bin 50 of weight 138: 37 34 34 33
Bin 51 of weight 126: 32 32 31 31
```

C H A P T E R

32

Car Sequencing Using EA

This lesson describes a method for solving the car sequencing problem using an evolutionary algorithm operating on an *indirect* representation. An indirect representation is one which does not directly encode an assignment to decision variables, but which encodes enough information for a *decoder* to produce such an assignment from the given information. The main advantage of using an indirect representation is that problem constraints can largely be ignored by the genetic operators, with the decoding phase (which instantiates the decision variables) taking care of maintaining feasibility through constraint propagation. This means that the genetic operators used can be fairly standard ones, and so custom operators are not needed for this example.

Make a copy of the example file `YourSolverHome/examples/src/tutorial/eacarseq_partial.cpp` and open this copy 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 section. At the end of the section, you will have completed the example and you can compile and run the program.

Problem description: Optimization model

A full description of the car sequencing problem is given in “Writing a goal: Car sequencing” on page 314 and is not repeated here. However, recall that the car sequencing problem is a *decision problem*: that is, a single solution is sought, and all solutions have

equal validity. A solution to the car sequencing problem schedules *n* vehicles in *n* slots while respecting the capacity constraints on each option.

For an evolutionary algorithm to work properly, it needs to have a notion of the quality of a solution. As such, evolutionary algorithms are not suited to pure decision problems unless the ‘‘distance’’ from a proposed solution to a real solution can be measured and used as a cost. That is, the decision problem needs to be transformed into an optimization problem before being solved via evolutionary algorithms. For the car sequencing problem, there are two fundamental ways of doing this: authorize an overuse of resources or an overuse of time. In the former, the capacity constraints on options are softened, allowing them to be violated. Any such violation can be given a cost. The sum of all such violation costs can then be used as a measure of distance from a solution to the original problem. If an assignment with this cost at zero is found, a solution to the car sequencing problem has been found. This example explores the other alternative for transformation into an optimization problem: allow more than *n* slots to build *n* cars. The number of additional slots used is used as a measure of the cost: again, if it is reduced to zero, a solution to the original car sequencing problem has been found.

In order to allow the number of slots to be increased, a certain number of ‘‘empty’’ configurations are added to the problem. These configurations require no options and so may be used in the schedule to buffer, or create free space, between real configurations. As you shall see in the description of the decoder, empty configurations are never inserted by choice—they are only inserted when no real configuration can be placed that would respect the capacity constraints on all options. One might view these configurations as a stall of the production pipeline. The number of empty configurations appearing before the last car is built can then be used as a cost. If this cost is reduced to zero, the schedule cannot be further compressed and a solution to the original problem has been found. As a convention, configurations are numbered from 1, the empty configuration having index 0.

Problem representation: Cost variable

The example contains a lot of code related to the reading of car sequencing problem instances and the construction of the model. As the focus of this lesson is on the use of genetic algorithms, and not on modeling, it is therefore suggested that you take some time to examine the `CarSeqProblem` structure which holds information about the car sequencing problem, and the `BuildModel` function which builds the car sequencing model. When you feel you are familiar with these, you can move on with the rest of the lesson.

The example code includes a function called `SolveWithGA` which will be used as the main entry point for the lesson. Its signature is below:

```
void SolveWithGA(IloSolver solver,
                 IloModel model,
                 CarSeqProblem problem,
                 IloIntVarArray sequence,
                 IloIntVar cost) {
```

The function takes the following parameters:

- `solver`: an instance of `IloSolver`, to use as an engine;
- `model`: the IBM® ILOG® Concert Technology model of the car sequencing problem;
- `problem`: an instance of `CarSeqProblem` containing problem data;
- `sequence`: the direct representation. An array of constrained variables holding the sequence of car configurations to be built;
- `cost`: the cost variable to be minimized. If cost is reduced to 0, a solution to the original car sequencing problem has been found.

The code to create all of the above objects is in the partially completed example with the exception of the cost variable. You will therefore write the code which will create it now, but first let us describe the idea. Assume that the original car sequencing problem required the scheduling of n cars in n slots. As explained, this problem is modified to allow up to $n + s$ slots, where s is the amount of *slack*, that is the number of empty configurations available. The cost is the number of empty configurations that appear before the last car in the schedule. However, such a calculation can be difficult to model; a simpler, equivalent alternative is to look at the last s slots of the sequence. Let k be the number of empty configurations occurring after the last car in this part; the cost is then $s - k$.

The problem is now to model the cost via such a calculation of k.

Step 1 Build the cost variable

Add the following code after the comment

```
// Build the cost variable

IloIntVar BuildCostVar(IloModel model, IloIntVarArray seq, IloInt slack) {
  IloEnv env(model.getEnv());
  IloIntVar cost(env, 0, slack);
  IloConstraintArray carAfter(env, slack);
  for (IloInt i = slack - 1; i >= 0; i--) {
    IlcInt j = seq.getSize() - slack + i;
    carAfter[i] = seq[j] != 0;
    if (i < slack - 1)
      carAfter[i] = carAfter[i] || carAfter[i + 1];
    model.add(carAfter[i] == cost > i);
  }
  return cost;
}
```

This code can seem complex, but can be broken down quite easily. `carAfter[i]` is true if and only if a car appears at or after position `i` in the *slack area* (last *s* elements of the sequence). Using `carAfter`, constraints then state that for each position `i`, the cost is strictly greater than `i` if and only if there is a car at `i` or after. Thus, if all elements of `carAfter` up to and including `i` are false, and are then true afterwards, then `carAfter[i]` being true ensures that $cost > i$, while `carAfter[i+1]` being true ensures that $cost <= i+1$, thus assigning the cost to the value $i+1$.

Problem representation: Car priorities

The first job to perform in the principal function `SolveWithGA` is to create the indirect representation of the car sequencing problem. Here, a system based on priorities has been chosen. A priority is assigned to each car to be built. Later a decoder will build the sequence of cars based on these priorities. The indirect representation thus defines an order in which cars should be built just as the direct representation does by assigning a car configuration to each slot. In this sense, the two representations are similar. It is important to notice the differences though. The indirect representation based on priorities gives a *desired* order, not a required order. The decoder will take these priorities, and try to respect the order defined by them as much as possible, but in reality, constraint propagation will mean that the order cannot be respected completely. Thus the indirect representation can be seen as a guide as to producing a solution.

You now create the indirect representation and add it to the model.

Step 2 Create the indirect representation

Add the following code after the comment

```
// Indirect representation
```

```
IloEnv env(solver.getEnv());
IloIntVarArray prio(problem._env, problem._numberOfCars, 0, 1000);
model.add(prio);
```

Here, priorities go from 0 to 1000, but any upper bound would be reasonable so long as it is not too low; a low maximum priority would result in too many cars with equal priority in any solution, which could lead to arbitrary tie-breaking rules being applied frequently in the decoder. Such reliance on tie-breaking rules is undesirable as it moves the search emphasis from the genetic search to these tie-breaking rules.

Problem solving: Representations and decoding

Next, the *solution prototype* is created, which will be used as a basis for all solutions created for use in the genetic algorithm. This solution contains the variables of both the indirect and direct representations.

Step 3 Create the solution prototype

Add the following code after the comment `// Create the solution prototype`

```
IloSolution prototype(env);
prototype.add(prio);
prototype.add(sequence);
prototype.add(IloMinimize(env, cost));
```

Here, both the indirect representation (`prio`) and the direct representation (`sequence`) are present in the prototype. The objective is also added to the solution which allows different solutions to be compared for quality.

Now, you will create the decoding goal which transforms an assignment in the indirect representation to one in direct representation.

Step 4 Create the decoding goal

Add the following code after the comment `// Create the decoding goal`

```
IloGoal decode = Decode(env, problem, prio, sequence);
```

Aside from the environment, the decoding goal takes three parameters: a description of the problem data (`problem`), the variables of the indirect representation (`prio`), and those of the direct representation (`sequence`). You will now move on to implementing this decoding goal which is at the heart of the solving method.

The decoding algorithm must instantiate all variables in `sequence` by using the values of the variables of `prio`. Each variable of `sequence` corresponds not to a car, but to a configuration, whereas each *car* has a priority in the indirect representation. The decoding algorithm is described below.

1. Start the current slot at the leftmost position 0.
2. If the current slot is beyond the end of the sequence then stop; a solution has been found.
3. If the current slot is assigned then:
   - let c equal the configuration assigned to the slot
   - if c is not the empty configuration then let k be the highest priority unscheduled car with configuration c; mark k as scheduled
   - move the current slot to the right and go to step 2.
4. Let C be the set of possible configurations for the current slot. Choose the highest priority unscheduled car k with a configuration $c \in C$
5. Create a choice point:
   - set the current slot to configuration c
   - remove configuration c from the current slot
6. Go to step 2.

Roughly speaking, at each step in the decoding process, the above algorithm places the highest priority unscheduled car compatible with the problem constraints (option capacities and number of each configuration to be built). Note that empty configurations are never considered as candidate vehicles by the above algorithm; they will be inserted automatically when all real configurations are impossible for a particular slot.

The priority theme is quite a general form of indirect representation which can be adapted for numerous other problems.

The decoding algorithm is written as an `IlcGoal` with a helper class which manages the priorities. This helper class stores a list of ordered priorities for each configuration as opposed to a priority for each car, which makes finding the best configuration faster, without changing the operation of the algorithm.

Step 5 Create the helper class

Add the following code of the helper class after the comment
`// Class to produce configs in priority order`

```
class ConfigurationPriorities {
private:
  IlcIntArray2   _priorities;
  IlcRevIntArray _index;

  static void Sort(IlcIntArray a);
public:
  ConfigurationPriorities(CarSeqProblem problem, IlcIntVarArray prio);
  IlcInt getHighestPriority(IlcInt conf) const {
    return _priorities[conf][_index[conf]->getValue()];
  }
  void markScheduled(IlcInt conf) {
    IloSolver solver(_priorities.getSolver());
    _index[conf]->setValue(solver, _index[conf]->getValue() + 1);
  }
};
```

This class has two fields: `_priorities` is a two-dimensional array of integers (`IlcIntArray2`) specifying the priorities. The two indices which access the priorities are the configuration and the index of the car with that configuration. Each column of this priority table is sorted in decreasing order. To give an example, imagine that you have 7 cars to build and two configurations. Assume the first four cars are of configuration one and the remainder have configuration two. Further assume that in a particular solution, the priorities of these cars (as instantiated in `prio`, passed to the constructor) are {342, 14, 875, 442, 129, 509, 812}. The constructor of `ConfigurationPriorities` will construct `_priorities` as follows:

`_priorities[1] =` {875, 442, 342, 14}

`_priorities[2] =` {812, 509, 129}

Note that index 0 of `_priorities` is not used as configuration 0 is the empty configuration; cars of this configuration have no priorities assigned, but are inserted only where necessary. The member function `getHighestPriority` delivers the priority of the highest priority unscheduled car of the given configuration.

The second field, `_index`, holds an array of indices in the `_priorities` array. There is one index for each configuration, and all indices are initialized to zero. For any configuration, all priorities below the index correspond to cars already scheduled, while the remainder correspond to cars waiting to be scheduled. As cars will be scheduled in a backtracking search, the indices are reversible.

The two inline member functions of the `ConfigurationPriorities` class are now easily explained. `getHighestPriority(conf)` delivers the priority of the highest priority unscheduled car of configuration `conf`. This will be used later in deciding between

configurations to place in a slot. The member function `markScheduled(conf)` is called to say that the highest priority unscheduled car of configuration `conf` has been scheduled, and so the index marker for that configuration should be moved on.

The codes of the constructor and sorting member function are already provided for you.

Now that the helper class has been defined, you will move on to writing the decoding method. This method is implemented as a goal and closely follows the algorithm already given.

Step 6 Add the decoder goal

Add the following code after the comment `// Decode priorities to a sequence`

```
ILCGOAL3(DecodeSlave, ConfigurationPriorities *, priorities,
                      IlcIntVarArray, seq,
                      IlcInt, idx) {
  IlcInt i;
  for (i = idx; i < seq.getSize() && seq[i].isBound(); i++)
    if (seq[i].getValue() != 0)
      priorities->markScheduled(seq[i].getValue());

  // No unbound slots, found a solution
  if (i == seq.getSize())
    return 0;
```

Here, you have implemented steps 2 and 3 of the decoding algorithm previously given. The decoding goal is passed the helper class, the sequence (direct representation) to decode to, and the current index in the sequence. Any scheduled slots are marked off by the helper class so long as they do not contain the empty configuration. At the end of the `for` loop, the index `i` points either to an unbound slot or past the end of the sequence. In the latter case, all slots have been decided, and a solution has been found.

In the next stage of decoding, given a slot to be decided, you will choose for it the highest priority available configuration.

Step 7 Decide on a configuration for the current slot

Add the following code after the comment `// Decide on a configuration`

```
  IloInt bestPrio = -1;
  IloInt bestConf = 0;
  for (IlcIntExpIterator it(seq[i]); it.ok(); ++it) {
    IlcInt conf = *it;
    if (conf != 0) {
      IlcInt p = priorities->getHighestPriority(conf);
      if (p > bestPrio) {
        bestPrio = p;
        bestConf = conf;
      }
    }
  }
```

This code is relatively simple, and its function is to exit with `bestConf` holding the configuration of the highest priority car which can be scheduled in the current slot. Empty configurations are not considered as candidates.

The chosen configuration is placed (with the configuration being removed on backtracking), and the goal recalled.

Step 8 Recurse to instantiate the remainder of the sequence

Add the following code after the comment `// Instantiate remainder of sequence`

```
  IlcGoal loop = DecodeSlave(getSolver(), priorities, seq, i);
  return IlcAnd(IlcOr(seq[i] == bestConf, seq[i] != bestConf), loop);
}
```

The decoding goal needs only a small driving goal (which calls `DecodeSlave` with an initial zero index) and a wrapper to create an `IloGoal` from an `IlcGoal`. These codes are provided for you and are both named `Decode`.

Problem solving: Creating the initial population

After creating the decoding goal, you will move on to create the initial population of solutions to start the genetic algorithm. Before doing this, you will first put in place an *operator factory* which will be used to generate and parameterize genetic operators, including an operator which creates a random member to be used in population construction.

Step 9 Set up an operator factory

Add the following code after the comment `// Set up an operator factory`

```
IloEAOperatorFactory factory(env, prio);
factory.setAfterOperate(decode);
factory.setSearchLimit(IloFailLimit(env, 100));
factory.setPrototype(prototype);
```

The operator factory is specified as operating over the `prio` array (that is, the indirect representation). After any genetic operation on the priorities, it is stated that the decoding goal must be invoked. In addition, this entire process (genetic operator + decoding) is limited to 100 fails in order to avoid excessive exploration times when a solution is difficult to produce. Finally, it is indicated to the factory that for production of new solutions by genetic operators, `prototype` should be used as a template. This prototype will be cloned internally and the clone used to contain the new solution.

You are now in a position to create the initial population.

Step 10 Make the initial population

Add the following code after the comment `// Create initial population`

```
const IloInt popSize = 30;
IloSolutionPool pop(env);
IloPoolProc create = factory.randomize();
solver.solve(IloExecuteProcessor(env, create(popSize) >> pop));

// Sort the population wrt. quality and display.
pop.sort();
env.out() << "Initial population: ";
DisplayCosts(pop);
```

Here, a pool processor is retrieved from the factory which is used to create a population member. This processor randomizes the values of the variables specified to the factory at construction time (in this case, the `prio` array). As you specified that the decoding goal should be invoked after each genetic operator produced by `factory`, then the `create` processor will also decode these randomized priorities to a car sequence. The desired population size is specified to the population creation operator via the `()` operator, which will ensure that at least `popSize` solutions are created. As these solutions are created randomly each time, then `popSize` *different* solutions should result, which are placed into the population solution pool `pop` using the `>>` operator. The operation of pool processors is such that if a solution failed to be produced at any invocation of the processor (due to the exhaustion of a search limit, for example), the processor is called again and again until the desired number of solutions is produced.

The resulting pool processor is invoked by the `IloExecuteProcessor` function which converts the processor to an instance of `IloGoal`.

The population is sorted in decreasing order of quality, which is possible as the objective was added to the solution prototype earlier; thus each solution knows how it should be compared with another.

Problem solving: A genetic algorithm cycle

Once the initial population has been generated you can now move on to the construction of the genetic algorithm main loop; each cycle around the loop being termed a generation. Here, in each generation, only one new individual will be generated, and hence in this genetic algorithm, a system of *replacement* will be used; this will be described later.

First, an intensification method will be introduced. Here, constraint programming is used to improve the quality of the new solutions produced by the genetic algorithm. This is done by adding constraints on the cost which state that any produced solution must be better than the parent or parents that produced it. To avoid stagnation and stalling where it is very difficult to improve on a parent, the improvement rule is only applied probabilistically. Imagine two probabilities p and q. You will stipulate that the newly generated solution must be (strictly) better than the best parent with probability p, otherwise it must be strictly better than the worst parent with probability q. This enforcement is carried out via an operator.

Step 11 Add the improvement goal

Add the following code after the comment `// Improve on worst in input pool`

```
ILOIIMOP2(ImproveOn, AnySize, IlcFloat, p, IlcFloat, q) {
  IloSolutionPool pool = getInputPool();
  IloIntVar var = pool.getSolution(0).getObjectiveVar();
  IlcInt limit = IlcIntMax;
  if (getSolver().getRandom().getFloat() < p)
    limit = (IloInt)pool.getBestSolution().getObjectiveValue() - 1;
  else if (getSolver().getRandom().getFloat() < q)
    limit = (IloInt)pool.getWorstSolution().getObjectiveValue() - 1;
  return getSolver().getIntVar(var) <= limit;
}
```

The operator is relatively straightforward, deciding on `limit` based on the best and worst solutions in the input pool (which will be the pool of parents when later used). A wrapper to produce an `IloPoolProcessor` from this operator is provided for you.

The operator factory is now primed such that `ImproveOn` is called before the invocation of any genetic operator generated by the factory.

Step 12 Create the parents solution pool

Add the following code after the comment

```
// Enforce probabilistic improvement
```

```
factory.setBeforeOperate(ImproveOn(env, 0.5, 0.9));
```

The operator dictates that with 50% chance a constraint is added which will insist that the offspring is better than the best parent, and with 45% chance that it is better than the worst parent. A 5% chance remains that the offspring can be equal in quality or worse than the worst parent.

You will now create the genetic operators to be used over the indirect representation.

Step 13 Create the selection of operators that will be used

Add the following code after the comment `// Create genetic operators`

```
IloPoolProcArray ops(env);
ops.add(factory.mutate(2.0 / problem._numberOfCars, "mutation"));
ops.add(factory.uniformXover("uxover"));
ops.add(factory.onePointXover("1ptxo"));

// Make a processor that will perform a breeding cycle
IloRandomSelector<IloSolution, IloSolutionPool> randomSelector(env);
IloPoolProc rndSel = IloSelectSolutions(env, randomSelector, IloTrue);
IloPoolProc breed = IloSelectProcessor(
  env,
  ops,
  IloRandomSelector<IloPoolProc,IloPoolProcArray>(env)
);
```

Here, a pool of processors is created. To this pool is added a mutation operator, which will mutate around two priorities each time it is applied, a uniform crossover operator, and a one-point crossover operator. A breeding operator is then created which applies a random selection of one of the above operators.

The breeding operator will produce a single offspring. After its application, a decision needs to be made on how this new solution is combined into the population: a *replacement* mechanism is needed. First, you will write the code for the use of the replacement, and then the details of its implementation will be examined.

Step 14 Replace the worst parent with the child and destroy this parent

Add the following code after the comment `// Child replaces worst parent`

```
IloSolutionPool parents(env);
IloPoolProc replace =
  IloSelectSolutions(env, ReplaceWorstParent(env, pop, parents));
IloPoolProc destroy = IloDestroyAll(env);
IloPoolProc replaceAndDestroy = replace >> destroy;
```

First, a pool to represent the parents selected for a genetic operator is created. Then, a selector is created which will both select a solution for destruction (producing it as output), and produce a side-effect on the population, removing the solution to be replaced and adding the solution that replaces it. The selector is passed the pool of parents because (as you will see) it operates by replacing one of the parent solutions in the population. The solution produced as output by `replace` will then be destroyed by processor `destroy`.

Before describing the replacement selector, you will see how all of the processors interact to produce one generation of the genetic algorithm.

Step 15 Create a pool processor

Add the following code after the comment
`// Create single-generation processor`

```
  IloPoolProc cycle = pop >> rndSel >> parents >> breed(1) >>
replaceAndDestroy;
```

The flow is relatively self-explanatory: solutions are selected randomly from the population, and placed in the parents pool; these parents are then bred using an operator chosen at random from the three available; the new solution then replaces a member of the population and the replaced solution is destroyed.

A more subtle point is revealed if we ask ourselves: how do we know how many parents to select from the population? The mutation operator requires only one parent, but the crossover operators need two. This problem is resolved by the manner in which pool processor connection (via `>>`) operates. Roughly speaking, control flow is right to left while data flow is left to right. This means that `replaceAndDestroy` is invoked first, which then asks `breed` for solutions. `breed` then chooses one of the three available operators, from which it can work out the number of parents needed. It then asks `rndSel` (indirectly, by passing via the `parents` pool) for the correct number of parents, which are in turn fetched from `pop`. At this point the selected solutions can flow back down the chain to the `breed` pool which creates the offspring which will then be passed to `replaceAndDestroy` to be merged with the population.

Problem solving: Solution replacement

The final remaining point to address is how the replacement of solutions operates in the genetic algorithm. Each generation produces a single offspring, and so a single member of the population will be replaced by this offspring. Here, a particular technique is used to maintain diversity of the population. This technique replaces one of the offspring's parents by the offspring. As children are often similar to parents (in computer genetics, as well as is in real genetics), this tends to keep a good diversity of solutions in the population. In order to favor better solutions, where there is more than one parent, the *worst* parent is chosen for selection.

There is one exception to the above rule. A rule often used in genetic algorithms is the so-called *elitist* rule where the best member of the population is never replaced if there is no equally good solution in the population. This rule is implemented here, and comes into effect when using the mutation operator (only one parent), and the child is of poorer quality than the parent. If, in this case the parent was the best of the population, it is not replaced, but the child is discarded instead.

Step 16 Add the replacement selector

Insert the following code after the comment `// The replacement selector`

```
ILOSELECTOR2(ReplaceWorstParent, IloSolution,
             IloSolutionPool, in,
             IloSolutionPool, pop,
             IloSolutionPool, parents) {
  IloSolution candidate = parents.getWorstSolution();
  IloSolution child = in.getSolution(0);
  pop.sort();
  if (pop.getSize() <= 1 || !candidate.isBetterThan(pop.getSolution(1))) {
    pop.remove(candidate);
    pop.add(child);
  }
  else
    candidate = child;
  select(candidate);
}
```

The parameters to the `ILOSELECTOR2` macro are the name of the resulting selector function, the *type* of object to select, the input pool to the selector from the pool processor to the left, and then the two parameters passed at construction time (population and parent pools).

First, the potential solution to replace, which is the worst of the parents, is taken from the parent pool; the child is also retrieved from the input pool. After sorting the population, the elitist rule is applied. The worst parent is replaced if the population only has one element, or if the second best member of the population is at least as good as the worst parent. This last test avoids replacing the parent if the parent is the best of the population, and better in

quality than the second best of the population. If the worst parent is to be replaced, it is removed from the population and the child added in its place. Otherwise, the candidate for destruction is the child. Finally, the `consider` method of the selector selects the candidate for destruction, which will be passed to the `destroy` processor in the code body.

Complete Program

The complete program follows and can be viewed on-line in `YourSolverHome\examples\src\eacarseq.cpp`.

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


#include <ilsolver/iim.h>

ILOSTLBEGIN

// A structure to hold instance data
struct CarSeqProblem {
  IloEnv _env;
  IloInt _numberOfCars;
  IloInt _numberOfOptions;
  IloInt _numberOfConfigurations;
  IloInt _slack;

  IloIntArray _optionCapacity;
  IloIntArray _optionSequenceLength;

  IloIntArray            _numberOfCarsPerConfiguration;
  IloArray<IloBoolArray> _configurationNeedsOption;

  void read(istream& stream, IloInt slack);
  CarSeqProblem(IloEnv env) : _env(env) { }
};

// Read in a problem from a file
void CarSeqProblem::read(istream & stream, IloInt slack) {
  _slack = slack;

  // Dimension problem
  stream >> _numberOfCars >> _numberOfOptions >> _numberOfConfigurations;

  // setup data structures
  _optionCapacity = IloIntArray(_env, _numberOfOptions);
  _optionSequenceLength = IloIntArray(_env, _numberOfOptions);
  _numberOfCarsPerConfiguration =
    IloIntArray(_env, _numberOfConfigurations + 1);
  _configurationNeedsOption =
    IloArray<IloBoolArray>(_env, _numberOfConfigurations + 1);
  IloInt i;
  for (i = 0; i < _numberOfConfigurations + 1; i++)
    _configurationNeedsOption[i] = IloBoolArray(_env, _numberOfOptions);
```

```
  // read problem body
  for (i = 0; i < _numberOfOptions; i++)
    stream >> _optionCapacity[i];
  for (i = 0; i < _numberOfOptions; i++)
    stream >> _optionSequenceLength[i];
  for (i = 0; i < _numberOfConfigurations; i++) {
    IloInt id, ncars;
    stream >> id >> ncars;
    id++; // ids to begin at 1 (0 is empty configuration)
    _numberOfCarsPerConfiguration[id] = ncars;
    for (IloInt j = 0; j < _numberOfOptions; j++)
      stream >> _configurationNeedsOption[id][j];
  }

  // Setup empty configuration.
  _numberOfCarsPerConfiguration[0] = _slack;
  assert(IloSum(_numberOfCarsPerConfiguration) == _numberOfCars + _slack);
  for (i = 0; i < _numberOfOptions; i++)
    _configurationNeedsOption[0][i] = IloFalse;
}

// Build the Concert model
IloModel BuildModel(CarSeqProblem problem, IloIntVarArray seq) {
  IloEnv env = problem._env;
  IloModel model(env);

  // Global distribute.  Ensures that the correct number of cars
  // with each configration is built.
  IloIntVarArray cards(env, 1 + problem._numberOfConfigurations);
  IloInt i;
  for (i = 0; i < problem._numberOfConfigurations + 1; i++) {
    IloInt nb = problem._numberOfCarsPerConfiguration[i];
    cards[i] = IloIntVar(env, nb, nb);
  }
  model.add(IloDistribute(env, cards, seq));

  // Sequence constraints.  Ensure that no sub-sequence contains
  // more of a given options that allowed
  for (IloInt opt = 0; opt < problem._numberOfOptions; opt++) {
    // Build an array of integers which represent configurations
    // which require option opt.
    IloIntArray pertinentConfigs(env);
    for (i = 0; i <= problem._numberOfConfigurations; i++) {
      if (problem._configurationNeedsOption[i][opt])
        pertinentConfigs.add(i);
    }

    // use us a boolean array for which use[i] = true iff
    // seq[i] has a configuration which requires opt
    IloBoolVarArray use(env, seq.getSize());
    model.add(IloBoolAbstraction(env, use, seq, pertinentConfigs));

    // Limit each subsequence to the correct number of occurrences
    // Loop over all subsequences of the correct sequence length
    // for option opt.  Add a sum to limit the maximum number of
    // occurrences in each sub-sequence.
    IloInt sl = problem._optionSequenceLength[opt];
```

```
    IloInt max = seq.getSize() - sl;
    IloInt x;
    for (x = 0; x <= max; x++) {
      IloBoolVarArray subUse(env, sl);
      for (i = 0; i < sl; i++)
        subUse[i] = use[x + i];
      model.add(IloSum(subUse) <= problem._optionCapacity[opt]);
    }
  }
  return model;
}

// Build the cost variable
IloIntVar BuildCostVar(IloModel model, IloIntVarArray seq, IloInt slack) {
  IloEnv env(model.getEnv());
  IloIntVar cost(env, 0, slack);
  IloConstraintArray carAfter(env, slack);
  for (IloInt i = slack - 1; i >= 0; i--) {
    IlcInt j = seq.getSize() - slack + i;
    carAfter[i] = seq[j] != 0;
    if (i < slack - 1)
      carAfter[i] = carAfter[i] || carAfter[i + 1];
    model.add(carAfter[i] == cost > i);
  }
  return cost;
}

// Useful type definitions
typedef IlcRevInt * IlcRevIntPtr;
ILCARRAY(IlcRevIntPtr)
typedef IlcRevIntPtrArray IlcRevIntArray;

ILCARRAY(IlcIntArray)
typedef IlcIntArrayArray IlcIntArray2;

// Class to produce configs in priority order
class ConfigurationPriorities {
private:
  IlcIntArray2   _priorities;
  IlcRevIntArray _index;

  static void Sort(IlcIntArray a);
public:
  ConfigurationPriorities(CarSeqProblem problem, IlcIntVarArray prio);
  IlcInt getHighestPriority(IlcInt conf) const {
    return _priorities[conf][_index[conf]->getValue()];
  }
  void markScheduled(IlcInt conf) {
    IloSolver solver(_priorities.getSolver());
    _index[conf]->setValue(solver, _index[conf]->getValue() + 1);
  }
};

// Build configuration priorities
ConfigurationPriorities
::ConfigurationPriorities(CarSeqProblem problem, IlcIntVarArray prio) {
  IloSolver solver(prio.getSolver());
  IlcInt numConf = problem._numberOfConfigurations;
```

```
  // Create array of sets
  _priorities = IlcIntArray2(solver, 1 + numConf);
  _index = IlcRevIntArray(solver, 1 + numConf);

  IlcInt k = 0;
  for (IlcInt conf = 1; conf <= numConf; conf++) {
    IlcInt numCars = problem._numberOfCarsPerConfiguration[conf];
    _priorities[conf] = IlcIntArray(solver, numCars);
    for (IlcInt j = 0; j < numCars; j++)
      _priorities[conf][j] = prio[k++].getValue();
    Sort(_priorities[conf]);
    _index[conf] = new (solver.getHeap()) IlcRevInt(solver, 0);
  }
}

// Sort integers in decreasing order
void ConfigurationPriorities::Sort(IlcIntArray a) {
  for (IlcInt i = 0; i < a.getSize() - 1; i++) {
    IlcInt big = i;
    for (IlcInt j = i + 1; j < a.getSize(); j++)
      if (a[j] > a[big]) big = j;
    IlcInt tmp = a[i]; a[i] = a[big]; a[big] = tmp;
  }
}

// Decode priorities to a sequence
ILCGOAL3(DecodeSlave, ConfigurationPriorities *, priorities,
                      IlcIntVarArray, seq,
                      IlcInt, idx) {
  IlcInt i;
  for (i = idx; i < seq.getSize() && seq[i].isBound(); i++)
    if (seq[i].getValue() != 0)
      priorities->markScheduled(seq[i].getValue());

  // No unbound slots, found a solution
  if (i == seq.getSize())
    return 0;

  // Decide on a configuration
  IloInt bestPrio = -1;
  IloInt bestConf = 0;
  for (IlcIntExpIterator it(seq[i]); it.ok(); ++it) {
    IlcInt conf = *it;
    if (conf != 0) {
      IlcInt p = priorities->getHighestPriority(conf);
      if (p > bestPrio) {
        bestPrio = p;
        bestConf = conf;
      }
    }
  }

  // Instantiate remainder of sequence
  IlcGoal loop = DecodeSlave(getSolver(), priorities, seq, i);
  return IlcAnd(IlcOr(seq[i] == bestConf, seq[i] != bestConf), loop);
}
```

```
// Build the priority structure
ILCGOAL3(Decode,
         CarSeqProblem, problem,
         IlcIntVarArray, prio,
         IlcIntVarArray, seq) {
  IloSolver solver(getSolver());
  ConfigurationPriorities * cp = new (solver.getHeap())
    ConfigurationPriorities(problem, prio);
  return DecodeSlave(solver, cp, seq, 0);
}

// An ILO wrapper for the decoding goal
ILOCPGOALWRAPPER3(Decode, solver,
                  CarSeqProblem, problem,
                  IloIntVarArray, prio,
                  IloIntVarArray, seq) {
  IlcIntVarArray solverPrio = solver.getIntVarArray(prio);
  IlcIntVarArray solverSeq = solver.getIntVarArray(seq);
  return Decode(solver, problem, solverPrio, solverSeq);
}

// Force improvement of offspring
ILOIIMOP2(ImproveOn, AnySize, IlcFloat, p, IlcFloat, q) {
  IloSolutionPool pool = getInputPool();
  IloIntVar var = pool.getSolution(0).getObjectiveVar();
  IlcInt limit = IlcIntMax;
  if (getSolver().getRandom().getFloat() < p)
    limit = (IloInt)pool.getBestSolution().getObjectiveValue() - 1;
  else if (getSolver().getRandom().getFloat() < q)
    limit = (IloInt)pool.getWorstSolution().getObjectiveValue() - 1;
  return getSolver().getIntVar(var) <= limit;
}

// The replacement selector
ILOSELECTOR2(ReplaceWorstParent, IloSolution,
             IloSolutionPool, in,
             IloSolutionPool, pop,
             IloSolutionPool, parents) {
  IloSolution candidate = parents.getWorstSolution();
  IloSolution child = in.getSolution(0);
  pop.sort();
  if (pop.getSize() <= 1 || !candidate.isBetterThan(pop.getSolution(1))) {
    pop.remove(candidate);
    pop.add(child);
  }
  else
    candidate = child;
  select(candidate);
}

// Display the costs of solutions
void DisplayCosts(IloSolutionPool pool) {
  IloEnv env(pool.getEnv());
  IloNum sum = 0;
  for (IloSolutionPool::Iterator it(pool); it.ok(); ++it) {
    IloNum cost = (*it).getObjectiveValue();
    sum += cost;
    env.out() << cost << " ";
```

```
  }
  env.out() << "(Mean = " << sum / pool.getSize() << ")" << endl;
}

// Main GA solving function
void SolveWithGA(IloSolver solver,
                 IloModel model,
                 CarSeqProblem problem,
                 IloIntVarArray sequence,
                 IloIntVar cost) {

  // Indirect representation
  IloEnv env(solver.getEnv());
  IloIntVarArray prio(problem._env, problem._numberOfCars, 0, 1000);
  model.add(prio);

  // Create the solution prototype
  IloSolution prototype(env);
  prototype.add(prio);
  prototype.add(sequence);
  prototype.add(IloMinimize(env, cost));

  // Create the decoding goal
  IloGoal decode = Decode(env, problem, prio, sequence);

  // Set up an operator factory
  IloEAOperatorFactory factory(env, prio);
  factory.setAfterOperate(decode);
  factory.setSearchLimit(IloFailLimit(env, 100));
  factory.setPrototype(prototype);

  // Start timing
  IloTimer timer(env);
  timer.start();

  // Create initial population
  const IloInt popSize = 30;
  IloSolutionPool pop(env);
  IloPoolProc create = factory.randomize();
  solver.solve(IloExecuteProcessor(env, create(popSize) >> pop));

  // Sort the population wrt. quality and display.
  pop.sort();
  env.out() << "Initial population: ";
  DisplayCosts(pop);

  // Enforce probabilistic improvement
  factory.setBeforeOperate(ImproveOn(env, 0.5, 0.9));

  // Create genetic operators
  IloPoolProcArray ops(env);
  ops.add(factory.mutate(2.0 / problem._numberOfCars, "mutation"));
  ops.add(factory.uniformXover("uxover"));
  ops.add(factory.onePointXover("1ptxo"));

  // Make a processor that will perform a breeding cycle
  IloRandomSelector<IloSolution, IloSolutionPool> randomSelector(env);
  IloPoolProc rndSel = IloSelectSolutions(env, randomSelector, IloTrue);
```

```
  IloPoolProc breed = IloSelectProcessor(
    env,
    ops,
    IloRandomSelector<IloPoolProc,IloPoolProcArray>(env)
  );


  // Child replaces worst parent
  IloSolutionPool parents(env);
  IloPoolProc replace =
    IloSelectSolutions(env, ReplaceWorstParent(env, pop, parents));
  IloPoolProc destroy = IloDestroyAll(env);
  IloPoolProc replaceAndDestroy = replace >> destroy;

  // Create single-generation processor
  IloPoolProc cycle = pop >> rndSel >> parents >> breed(1) >>
replaceAndDestroy;

  // The main genetic algorithm loop
  const IloInt maxGen = 10000;
  IloInt gen;
  IloSolution best = pop.getBestSolution();
  for (gen = 1; best.getObjectiveValue() != 0 && gen <= maxGen; gen++) {
    solver.solve(IloExecuteProcessor(env, cycle));
    best = pop.getBestSolution();
    pop.sort();
    if ((gen % 10) == 0) {
      env.out() << "Generation " << gen << ": ";
      DisplayCosts(pop);
    }
  }

  // Display best solution and statistics
  solver.solve(IloRestoreSolution(env, best));
  env.out() << "Time taken = " << timer.getTime() << endl;
  env.out() << "Terminated at generation " << gen << endl;
  env.out() << "Solution of cost " << solver.getValue(cost) << " is: " << endl;
  for (IloInt i = 0; i < sequence.getSize(); i++)
    env.out() << solver.getValue(sequence[i]) << " ";
  env.out() << endl;
}

// Solve car sequencing problems using a GA
int main(int argc, const char * argv[]) {
  IloEnv env;
  try {
    // Default input file
    const char * fname = "../../../examples/data/carseq1.dat";

    // Number of empty configurations to add by default
    IloInt slack = 20;

    // Read command line arguments
    if (argc > 1)
      fname = argv[1];
    if (argc > 2)
      slack = atoi(argv[2]);
```

```
    // Open input file and read problem data
    ifstream stream(fname);
    if (!stream.good()) {
      env.out() << "Cannot open " << fname << endl;
      return 0;
    }
    CarSeqProblem problem(env);
    problem.read(stream, slack);

    // Create the decision variables and the model.
    IloIntVarArray seq(env, problem._numberOfCars + problem._slack,
                       0, problem._numberOfConfigurations);
    IloModel model = BuildModel(problem, seq);
    IloIntVar costVar = BuildCostVar(model, seq, problem._slack);

    // Create the solver and solve.
    IloSolver solver(model);
    SolveWithGA(solver, model, problem, seq, costVar);
  } catch (IloException & ex) {
    env.out() << "Caught: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Output

Execution of the program results in the following output:

```
Initial population: 3 4 4 5 5 6 6 6 6 7 7 7 7 7 8 8 8 8 8 8 8 8 9 10 10 11 11
13 13 14 (Mean = 7.83333)
Generation 10: 3 3 4 4 4 5 5 5 6 6 6 6 6 6 7 7 7 7 8 8 8 8 8 9 10 10 10 10 13
14 (Mean = 7.1)
Generation 20: 3 3 3 4 4 4 4 4 4 4 5 5 5 5 5 5 6 6 6 6 7 7 7 8 8 8 9 10 10 14
(Mean = 5.96667)
Generation 30: 2 3 3 3 3 4 4 4 4 4 4 4 5 5 5 5 5 5 6 6 6 6 6 6 8 8 8 10 10 12
(Mean = 5.46667)
Generation 40: 2 2 3 3 3 3 3 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 6 6 6 6 6 6 6 7
(Mean = 4.43333)
Generation 50: 2 2 2 2 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 6 6 6 7
(Mean = 4.03333)
Generation 60: 1 1 2 2 2 2 3 3 3 3 3 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 6 6 7
(Mean = 3.7)
Time taken = 3.76
Terminated at generation 63
Solution of cost 0 is:
14 16 11 5 17 3 17 1 11 7 15 13 6 15 5 17 3 9 7 11 7 15 6 4 5 17 5 12 5 10 14
17 8 16 7 11 5 17 3 9 14 11 2 17 3 15 16 7 11 7 15 8 10 1 17 8 15 5 17 3 9 7 11
5 17 8 10 5 18 2 16 3 10 14 11 1 17 3 10 14 11 7 9 8 10 5 17 1 11 7 15 7 15 11
16 2 15 7 11 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Part VII

Developing Solver Applications

This part consists of the following lessons:

- Chapter 33, *Designing Models*
- Chapter 34, *Debugging and Tracing*
- Chapter 35, *Developing Applications*

CHAPTER

33

Designing Models

This chapter offers a few design principles, gleaned from experienced users of Solver. These principles are meant to help you avoid the errors often made by new users of Solver when they design a model for a problem. Of course, not every problem benefits from a mechanical application of every principle we mention here, but in general these principles should help you develop robust and efficient Solver programs.

Decrease the number of variables

The unknowns of a given problem are typically represented in its model by constrained variables. There are practical ways of cutting down on the number of variables and thus reducing the size of the model and its search space.

Problems best solved with constraint-based programming are generally subject to intrinsic combinatorial growth. Even if reducing the domains of variables by propagating constraints makes it possible to reduce the search space, the initial size of this search space still remains a weighty factor in the execution time.

Principle

Consequently, good practice in designing a model should attempt to minimize the size of the search space in the first place. This size increases exponentially with the number of

variables. Thus, limiting the number of such variables (even at the expense of enlarging their domains) can reduce the combinatorial complexity.

Use dual representation

Object programming and straightforward object manipulation in Solver often make it possible to design a direct and very intuitive model of a problem. Nevertheless, we must not get stuck in this apparent simplicity if that model gives rise to a disproportionate amount of propagation or outright inefficiency.

Principle

Dual representation consists of looking at a problem “from the other side,” so to speak. This technique assumes that the designer knows enough about the problem to step back and consider its essence and thus extract conceptual *symmetries* inherent in it.

Example

Allocating resources offers a good example of this phenomenon. Let's assume that C consumers must choose among R resources; to make it simple, let's limit ourselves to the case where:

- all the resources are available to every consumer;
- each consumer uses at most one resource;
- each resource may be used by at most one consumer.

The apparent complexity of the problem depends on the kind of model. In practice, there are two possible models:

- The consumers choose resources.
- The resources choose consumers.

In the first case, if we associate a constrained variable representing the chosen resource with each consumer, we get C variables with a domain of size $R+1$, where R is the number of possible resources. The apparent complexity of the problem in this model is thus $(R+1)^{C}$.

The second case is, of course, analogous to the first: we associate a constrained variable with each resource such that the variable represents the chosen consumer, so we get R variables with a domain of size $C+1$, where C is the number of possible consumers. The apparent complexity of the problem in this model is thus $(C+1)^{R}$.

When the difference in apparent complexity is great, we must consider the magnitude of C and R very carefully.

Remove symmetries

The apparent complexity of a problem can often be reduced to a much smaller *practical complexity* by detecting intrinsic symmetries. Parts of the search tree can then be safely ignored.

Group by type

When two or more variables have identical characteristics, it is pointless to differentiate them artificially.

Principle

Rather, it is preferable to design a model that takes into account not simply the elementary entities but instead the *types* into which the elementary entities can be meaningfully grouped. Each such type, of course, *quantitatively* handles the elementary entities that belong to it.

Example

For example, let's say that we want to plug electronic cards into racks. We'll assume that there are five types of cards. Each rack may contain only a finite number of cards according to some constraints. We want to use Solver to minimize the number of racks used. This problem may be formulated in two different ways:

- first model—for each card, the problem is to find the rack where it is to be plugged;
- second model—for each rack, the problem is to find the number of cards of a given type plugged into it.

In the first model, we define one variable per card. Once we have a solution, any permutation of two cards of the same type defines a “new” solution. The search space is consequently large and contains many such symmetrical solutions that don't really interest us.

In the second model, we manipulate the number of cards of a given *type*, rather than the cards themselves. This model suppresses the symmetries introduced by the previous one and thus reduces the search space.

Introduce order among variables

There is really no point in examining all the possible solutions for variables and their values when two or more constrained variables satisfy the following conditions:

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

In fact, the permutations give rise to *sets* of solutions that are identical as far as the physical reality of the problem is concerned. We can exploit this idea to minimize the size of the search space.

Principle

If we reduce these domains by introducing a supplementary constraint, such as order, or by imposing a special feature on each of these variables, we can markedly reduce the size of the search space.

Example

Let's assume, for example, that we have the following system of equations:

$x_1 + x_2 + x_3 = 9$

$x_1 \times x_2 \times x_3 = 12$

For the ordered triple *(x1, x2, x3)*, there are six solutions:

(1, 2, 6) (1, 6, 2) (2, 1, 6) (2, 6, 1) (6, 1, 2) (6, 2, 1)

If we permute the variables, the problem is not changed. For instance, if we swap *x1* and *x2*, the problem becomes:

$x_2 + x_1 + x_3 = 9$

$x_2 \times x_1 \times x_3 = 12$

That problem is obviously the same as the first one. In this case, it is a good idea to introduce a supplementary constraint to enforce order among the variables. We can do that in this way:

$x_1 \leq x_2 \leq x_3$

Our additional constraint on the order among the variables greatly reduces the combinatorial possibilities *without removing any real solutions*.

In fact, only one solution can be returned under these conditions:

$(x_1, x_2, x_3) = (1, 2, 6)$

We also note that the presence of large inequalities in the constraint takes care of the possibility of getting the same value for two or more variables. In our example, that is the effect that we want, but more generally, we have to guard against adding a supplementary constraint that inadvertently suppresses solutions that we would like to see.

Use constrained set variables

Among the elements of a set, there is no necessary order, and a set differs from a list in this respect. Solver offers constrained set variables and predefined constraints such as cardinality to represent certain problems as *sets*.

Principle

Representing a problem as a set is useful for avoiding symmetries. Moreover, constrained set variables may help us reduce the apparent complexity of a model.

Example

Let's consider, for example, a crew-allocation problem in an airline company. Let's assume that a given flight requires six crew members.

We can represent this assumption by associating six constrained variables with the flight. Each of these variables represents a member of the crew. However, this representation is symmetrical: since there is no particular difference represented among the six crew members in this model, any permutation of the values of the six variables does not change the crew, but such a permutation would give rise to more (apparently different) solutions that really do not interest us.

A better model—one that suppresses those phony "different" solutions—uses a single constrained set variable, with a cardinality equal to 6. This model avoids the symmetries of the previous one since it accurately represents a crew as a *set* of members.

Back propagate costs

The cost variable must be related to the other constrained variables of the problem through constraints.

Principle

In Solver, the minimization algorithm introduces a supplementary constraint on the upper bound of the cost function at every iteration.

For this reason, the constraints between the cost variable and other variables have to carry propagation as far as possible in *both directions*. By propagating in both directions, we mean that changes in the cost variable must be propagated to the other variables, and any changes in the other variables must be propagated to the cost variable as well.

Example

Let's assume that we want to maximize a sum of five variables that are integers between 0 and 3, inclusive, and let's count the number of tries necessary to do so.

```
ILCGOAL2(computeSum, IlcIntVar, var, IlcIntVarVarArray, vars){
  IloSolver solver = getSolver();
  IlcInt sum = 0;
  for(IlcInt i = 0; i < vars.getSize(); i++)
    sum += vars[i].getValue();
  var.setValue(sum);
  return 0;
}

ILOCPGOALWRAPPER2(computeSum, solver, IloNumVar, var, IloNumVarArray, vars){
  IlcIntVarArray svars = solver.getIntVarArray(vars);
  IlcIntVar svar = solver.getIntVar(var);
  return IlcComputeSum(solver, var, vars);
}


int main(){
  IloEnv env;
  IloModel m(env);
  IloIntVarArray vars(env, 5, 0, 3);
  IloIntVar sum(env, 0, 100);
  m.add(vars);
  m.add(sum);
  m.add(IloMaximize(env, sum));
  IloSolver s(m);
  s.solve(IloGenerate(env, vars) && computeSum(env, sum, vars)); // not good
  cout << "Sum = " << s.getIntVar(sum) << endl;
  cout << "#tries = " << s.getNumberOfChoicePoints() << endl;
  env.end();
  return 0;
}
```

Here's the output of this code:

```
Sum = 15
#tries = 5145
```

In the code that we just discussed, the program evaluates the cost function only after choosing values. In the next version, a constraint is set to reduce the domains *before* the search for a solution.

```
int main(){
  IloEnv env;
  IloModel m(env);
  IloIntVarArray vars(env, 5, 0, 3);
  IloIntVar sum(env, 0, 100);
  m.add(sum == IloSum(vars));  // good
  m.add(IloMaximize(env, sum));
  IloSolver s(m);
  s.solve(IloGenerate(env, vars));
  cout << "Sum = " << s.getIntVar(sum) << endl;
  cout << "#tries = " << s.getNumberOfChoicePoints() << endl;
  env.end();
  return 0;
}
```

This version finds the maximal solution with far fewer tries than the previous one, and thus, of course, represents considerable improvement in performance. Here's its output:

```
Sum = 15
#tries = 45
```

Use appropriate search strategies

Back propagating costs (as explained in the previous section) is efficient to the degree that the first solutions found have a low cost. Good search strategies (that is, good strategies for selecting variables and values to try) promote this goal.

Principle

Solver makes it possible to try many different search strategies fairly easily and without having to change the representation of domain constraints. We urge you to take advantage of this facility to try out different ways of solving a given problem to find the one most advantageous to your situation.

There is no single rule of thumb for discovering what is a good strategy. You should try strategies that:

- are deduced from common sense;
- use know-how from the problem domain;
- borrow methods from operations research.

However, there are two main kinds of strategies: the one used for constraint satisfaction problems, and the one used for optimization problems.

It is our experience that when we consider satisfaction problems, a strategy that is often successful for selecting variables is to choose the most constrained one. In effect, those variables are where propagation is the most significant, and as a consequence, where entire portions of the search tree can be pruned as quickly as possible. A predefined Solver strategy—`IlcChooseMinSizeInt`—can be most profitably applied in this respect.

When we consider optimization problems, strategies based on generation, such as `IloGenerate`, often first generate those variables and their values that are *statistically* the most likely to lead to a low-cost solution.

Example

Let's continue the preceding example. The code presented in the preceding section would search in these ways:

- ◆ It would select variables in the order defined by their index in the array `vars`.
- ◆ It would select values by starting with the lowest ones, since this selection strategy is the default.

In contrast, we could first try the highest values, since we are interested in the highest possible sum. For that purpose, we define our own selector to choose the variables with the largest maximum values in their domains. In our definition, we use the implementation class

IlcIntSelectI. (The *IBM ILOG Solver Reference Manual* documents IlcIntSelectI.) Here's the code to do this:

```
class selectMaxI : public IlcIntSelectI {
public:
  selectMaxI():IlcIntSelectI(){}
  virtual IlcInt select(IlcIntVar var){
    return var.getMax();
  }
};

IlcIntSelect selectMax(IloSolver s){
   return new (s.getHeap()) selectMaxI();
}

class IloSelectMaxI : public IloIntValueSelectorI {
public:
  IloSelectMaxI(IloEnvI* env) : IloIntValueSelectorI(env) {}
  IlcIntSelect extract(const IloSolver solver) {
    return selectMax(solver);
  }
  IloIntValueSelectorI* makeClone(IloEnvI* env) {
    return new (env) IloSelectMaxI(env);
  }
};

int main(){
  IloEnv env;
  IloModel m(env);
  IloIntVarArray vars(env, 5, 0, 3);
  IloIntVar sum(env, 0, 100);
  m.add(sum == IloSum(vars));
  m.add(IloMaximize(env, sum));
  IloSolver s(m);
  s.solve(IloGenerate(env, vars, IloChooseMaxMaxInt, IloSelectMax(env)));
  cout << "Sum = " << s.getIntVar(sum) << endl;
  cout << "#tries = " << s.getNumberOfChoicePoints() << endl;
  env.end();
  return 0;
}
```

This improved version searches in this way:

- It first selects the variables with the largest maximum.
- It selects values by starting with the highest ones.

Here's the output, demonstrating the improvement in the number of tries:

```
Sum = 15
#tries = 5
```

Introduce redundant constraints

Since constraint propagation decreases the search space by reducing the domains of variables, it is obviously important to express all necessary constraints. In some cases, it is even a good idea to introduce *redundant* constraints to reduce the size of the search space by supplementary propagation. Processing supplementary constraints inevitably slows down execution. However, this slowing down may be negligible in certain problems when it is compared with the efficiency gained from reducing the size of the search space.

Principle

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

Of course, there is no need to express grossly obvious redundancy since the highly optimized algorithms that Solver uses to insure *arc consistency* already work well enough. For example, if we are given this system of equations:

$x = y + z$

$z = a + b$

we gain no efficiency whatsoever by adding this constraint:

$x = y + a + b$

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

Example

The problem of the magic sequence is described Chapter 4, *Searching with Predefined Goals: Magic Square*. Let's describe it again briefly. Let's assume that we have $n+1$ unknowns, namely, $x0, x1, \ldots, xn$. These xi must respect the following constraints:

- 0 appears $x0$ times in the solution.
- 1 appears $x1$ times.
- In general, i appears xi times.
- n appears xn times.

The constraint of this problem can easily be written, using the symbolic counting constraint `IloDistribute`. However, the search for a solution can be greatly accelerated by

introducing the following redundant constraint that expresses the fact that $n+1$ numbers are counted.

$1.x1 + 2.x2 + \ldots + n.xn = n+1.$

Use objects

Since Solver supports object-oriented programming, it provides inheritance mechanisms that produce clear, re-usable, and extendable code. You can exploit object orientation in your Solver applications to organize the data in your problem into a coherent model and thus improve the solution.

Principle

If you exploit objects and classes, they can greatly assist you in organizing the information needed to solve your problem. They can even be a great help in the implementation itself. In small scale examples (like the ones in this manual), object orientation may seem like overkill, but its generality rapidly becomes advantageous in real-world problems.

Besides the conventional benefits of inheritance, using objects and enumerated variables in Solver offers at least the following additional advantages:

- You can group all data about the same entity in the same object. This grouping means that you avoid splitting entity data over vectors and then having to index those vectors to retrieve those data.
- You can define and post *class constraints* on any class of objects.

Use reversible and constrained data members

Let's assume that an object has a data member with a value that is modified while the search for a solution is going on. The value assigned to this data member depends on the choices and the hypotheses made before this assignment.

If those choices fail to find a solution further on, Solver backtracks, and the computed value for this data member is no longer valid. For that reason, this data member must be a *reversible data member*. That is, the modifier of this data member must save the initial value of the data member before any modification, so Solver is able to restore the value of the data member to its value before modification if any failure occurs later. For example, Solver provides you with the predefined class `IlcRevInt`, the class of reversible integers, for just such a purpose.

Let's also assume that an object has a data member whose value must be found by Solver. The right way to implement this situation is to associate a constrained variable with this data

member. The value of the data member is defined to be the value of the associated constrained variable. This value is undefined as long as the variable is not bound.

In short, Solver allows you to take full advantage of its constraint-programming facilities at the same time that it fully supports objects, classes, and inheritance of object-oriented programming. Constraints can figure as data members; data members can be fully reversible.

Exploit generic and class constraints

Often, in a problem, a set of objects must satisfy similar constraints. In these cases, it is worth trying to use *generic* constraints, that is, constraints that are shared by several constrained variables. Since generic constraints are shared, they may save a lot of memory. For instance, the constraint `IloAllDiff` is a generic constraint: only one constraint is allocated, whatever the number of variables in the array it constrains.

In fact, there are two ways in Solver to implement such a constraint: as a generic constraint or as a class constraint.

- *generic constraint*: the constraint has a data member containing the array of constrained variables sharing the constraint;
- *class constraint*: the constructor for the class posts the constraint on each instance of the given class.

Use a known solution for testing

The previous sections have explained how to design good models for various problems. However, if errors slip in when you implement the constraints of the problem, it can be very difficult to understand why the application finds wrong “solutions,” even though the model is very good.

There is a simple way to fix such cases: use an instance of the problem with a known solution to test against the constraints. For testing a solution, you just have to assign the values corresponding to a solution *before* posting the constraints. Then no failure should happen within this function. If a failure occurs with the known solution, you know there is a problem among the constraints.

To find the source of the problem, use a trace event to locate the faulty constraint. See Chapter 34, *Debugging and Tracing* for more information.

For example, let's test a known solution of the map-coloring problem.

```
#include <ilsolver/ilosolver.h>

ILOSTLBEGIN

const char* Names[] = {"blue", "white", "red", "green"};
```

```
int main(){
  IloEnv env;
  try {
    IloModel model(env);
    IloIntVar Belgium(env, 0, 3), Denmark(env, 0, 3),
      France(env, 0, 3), Germany(env, 0, 3),
      Netherlands(env, 0, 3), Luxembourg(env, 0, 3);

    // Test a solution
    model.add(Belgium == 0);
    model.add(Denmark == 1);
    model.add(France == 1);
    model.add(Germany ==  0);
    model.add(Netherlands == 1);
    model.add(Luxembourg == 2);

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

    IloSolver solver(model);
    // Search for a solution
    if (solver.solve()) {
      solver.out() << solver.getStatus() << " Solution" << endl;
      solver.out() << "Belgium:     "
                   << Names[(IloInt)solver.getValue(Belgium)]     << endl;
      solver.out() << "Denmark:     "
                   << Names[(IloInt)solver.getValue(Denmark)]     << endl;
      solver.out() << "France:      "
                   << Names[(IloInt)solver.getValue(France)]      << endl;
      solver.out() << "Germany:     "
                   << Names[(IloInt)solver.getValue(Germany)]     << endl;
      solver.out() << "Netherlands: "
                   << Names[(IloInt)solver.getValue(Netherlands)] << endl;
      solver.out() << "Luxembourg:  "
                   << Names[(IloInt)solver.getValue(Luxembourg)]  << endl;
    }
  }
  catch (IloException& ex) {
    cerr << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

C H A P T E R

34

Debugging and Tracing

Since Solver is a C++ library, you can use any C++ development environment to develop and debug applications that exploit Solver. In addition, there are certain supplementary features offered by Solver for tracing constraint propagation, and this chapter shows you how to use those facilities. Among other topics, this chapter describes how to:

- trace the modifications of some or all constrained variables;
- distinguish between *direct* modifications made by a program and *derived* modifications made by propagation of constraints;
- report the cause of each constraint propagation;
- generate trace messages in the standard output or in a file.

Using a C++ development environment

You can use your favorite C++ development environment to trace and to set breakpoints for any function of Solver.

Tracing demons

Whatever C++ debugger you are using, you are able to trace the execution of a Solver program, since Solver is a standard C++ library. If you specifically want to trace the

propagation of constraints, you have several choices. The simplest consists of tracing the constraint *demons* used for propagation.

Let's consider the following constraint:

```
class IlcLeConstraint: public IlcConstraintI {
  // x <= y + c
  IlcIntExp _x;
  IlcIntExp _y;
  IlcInt _c;
public:
  IlcLeConstraint(IloSolver solver, IlcIntExp x, IlcIntExp y, IlcInt c=0)
    : IlcConstraintI(solver), _x(x), _y(y), _c(c) {}
    void post();
    void propagate();
    IlcBool isViolated() const;
    void metaPostDemon(IlcDemonI*);
    IlcConstraintI* makeOpposite() const;
    void xDemon(){
      _y.setMin(_x.getMin() - _c);
    }
    void yDemon(){
      _x.setMax(_y.getMax() + _c);
    }
};

ILCCTDEMON0(IlcLeConstraintXDemon,IlcLeConstraint,xDemon);

ILCCTDEMON0(IlcLeConstraintYDemon,IlcLeConstraint,yDemon);

void IlcLeConstraint::post(){
  _x.whenRange(IlcLeConstraintXDemon(getSolver(),this));
  _y.whenRange(IlcLeConstraintYDemon(getSolver(),this));
}

void IlcLeConstraint::metaPostDemon(IlcDemonI* demon){
  _x.whenRange(demon);
  _y.whenRange(demon);
}

IlcBool IlcLeConstraint::isViolated() const {
  return _x.getMin() > _y.getMax() + _c;
}

void IlcLeConstraint::propagate(){
  xDemon();
  yDemon();
}

IlcConstraintI* IlcLeConstraint::makeOpposite() const {
  IloSolver solver=getSolver();
  return new (solver.getHeap()) IlcLeConstraint(solver,_y,_x,-_c -1);
}

IlcConstraint IlcLe(IlcIntExp x, IlcIntExp y, IlcInt c){
  IloSolver solver=x.getSolver();
  return IlcConstraint(new (solver.getHeap()) IlcLeConstraint(solver,x,y,c));
}
```

The idea is to set watch points on the member functions you want to trace. Let's assume that you are interested in the demon triggered by modifications of the variable `_x` in that constraint. It is sufficient to set a watch point on the member function `IlcLeConstraint::xDemon`.

If you are using `dbx` on a UNIX work station, for example, you would trace those demons in this way:

```
(dbx) trace IlcLeConstraint::xDemon
```

Consult the documentation of your favorite development environment for details about setting watch points.

This approach is possible only if you have access to the source code. That is, it can be used only on the constraints you define.

The rest of this chapter explains how to trace the internals of Solver (for which you don't have the source code).

Stopping on constraint propagation

The *IBM ILOG Solver Reference Manual* documents the classes `IlcTrace`. Also in “Trace events” on page 598, we go into greater detail about trace events. Right now, in the context of tracing from a debugger, we want to indicate that instances of both those classes can be useful.

When you are running a Solver program from a debugger, and using instances of the class `IlcTrace` to trace the modifications of constrained variables, you can execute the constraint propagation step by step. All trace member functions of `IlcTrace` call the function `ilc_trace_stop_here`. A breakpoint can be set for the function `ilc_trace_stop_here` to stop the program after each call to a trace member function of the class `IlcTrace`. You do that in this way if you are using `dbx` or `gdb`:

```
(dbx) stop in ilc_trace_stop_here
(gdb) break ilc_trace_stop_here
```

On some platforms, such as Windows NT® , for example, you must prefix an underscore to the name of the function, like this: `_ilc_trace_stop_here`.

Stopping on failures

When you are running a Solver program from a debugger, such as `dbx` or `gdb`, you can stop the execution on failures. A breakpoint can be set for the function `ilc_fail_stop_here`

to stop the program after each call to `IlcConstraintI::fail` or `IlcGoalI::fail`. You do that like this:

```
(dbx) stop in ilc_fail_stop_here
(gdb) break ilc_fail_stop_here
```

On some platforms, such as Windows NT® , for example, you must prefix an underscore to the name of the function, like this: `_ilc_fail_stop_here`.

Trace

Solver provides a trace facility, which is based on *trace events*. You can limit your trace to only those modifications that affect a constrained variable. You can also access a constrained variable before it is modified and after it is modified. You can also access at any time the demon or constraint that precipitated the modification of a constrained variable.

The trace facility is provided by the class `IlcTrace`, and its implementation class `IlcTraceI`. An instance of `IlcTraceI` enables you to trace such events as failure of a variable, a demon, or a constraint in a solution search. With an instance of this class, you can access the modifications of an active demon or an active constraint at any time by calling the member functions: `IlcTraceI::getActiveDemon`, `IloSolver::getActiveDemon`, and `IloSolver::getActiveGoal`.

The handle class `IlcPrintTrace` and its implementation class `IlcPrintTraceI` derive from the trace classes. They facilitate printing and displaying trace information.

All printing functions of Solver use an output stream, `IloSolver::out`. By default, this output stream is equal to `cout` (except on one platform where it is redirected to a log file).

Trace events

Solver uses `#define` to define the events that you can trace with an instance of `IlcTraceI`:

```
#define IlcTraceDemons ((IlcUInt) 1)
#define IlcTraceConstraint ((IlcUInt) 2)
#define IlcTraceProcess ((IlcUInt) 4)
#define IlcTraceVars ((IlcUInt) 8)
#define IlcTraceFail ((IlcUInt) 16)
#define IlcTraceNoEvent ((IlcUInt) 32)
#define IlcTraceAllEvent ((IlcUInt) 31)
```

When you create an instance of `IlcTraceI`, use one of those values to the constructor to indicate which event or events to trace. After you have created a trace (an instance of `IlcTraceI`) if you want to alter the events that it traces, use the member functions `setTraceDemon`, `setTraceConstraint`, and so forth, to alter the trace.

Use the member function `IlcPrintTrace::trace` to hook a variable; that is, create a link between a print trace and a variable. If you *hook* a constrained variable, you can trace each modification of the domain of that variable.

Tracing choice points and failures

Trace functions can be activated or inhibited with respect to a solver. The member function `IloSolver::setTraceMode` takes a Boolean value as its argument. If this argument is `IlcTrue`, it activates the trace functions of Solver with respect to the invoking solver. If the argument is `IlcFalse`, it inhibits the trace functions.

We'll use the `sendmory` example here to illustrate how tracing works. If we modify the `sendmory` example by setting the trace mode to `IlcTrue`, the program becomes:

```
ILOCPTRACEWRAPPER0(PrintConstraintTrace, solver) {
  solver.setTraceMode(IlcTrue);
  IlcPrintTrace trace(solver, IlcTraceConstraint);
  solver.setTrace(trace);
}

int main(){
  IloEnv env;
  try {
    IloModel mdl(env);
    IloIntVar S(env, 0, 9, "S"),
              E(env, 0, 9, "E"),
              N(env, 0, 9, "N"),
              D(env, 0, 9, "D"),
              M(env, 0, 9, "M"),
              O(env, 0, 9, "O"),
              R(env, 0, 9, "R"),
              Y(env, 0, 9, "Y");
    IloIntVarArray vars (env, 8, S, E, N, D, M, O, R, Y);
    mdl.add(S != 0);
    mdl.add(M != 0);
    IloConstraint alldiff=IloAllDiff(env,vars);
    mdl.add(alldiff);
    mdl.add(  1000*S + 100*E + 10*N + D
     + 1000*M + 100*O + 10*R + E
     == 10000*M + 1000*O + 100*N + 10*E + Y);
   IloSolver solver(mdl);
   IlcIntVarArray svars=solver.getIntVarArray(vars);
   for (IlcInt i=0;i<vars.getSize();i++){
     svars[i].setName(vars[i].getName());
   }
   solver.addTrace(PrintConstraintTrace(env));
   solver.solve(IloGenerate(env, vars));
   }
  catch (IloException ex) {
    cerr << "Error: " << ex << endl;
  }
  env.end();
  return 0;
```

```
}
```

Its output becomes:

```
>> begin-constraint-propagate ((S[0..9] != 0))
>> end-constraint-propagate ((S[1..9] != 0))
>> begin-constraint-propagate ((M[0..9] != 0))
>> end-constraint-propagate ((M[1..9] != 0))
>> begin-constraint-post (IlcAllDiff(009DE938) {S[1..9], E[0..9], N[0..9],
D[0..
9], M[1..9], O[0..9], R[0..9], Y[0..9], })
>> end-constraint-post (IlcAllDiff(009DE938) {S[1..9], E[0..9], N[0..9],
D[0..9]
, M[1..9], O[0..9], R[0..9], Y[0..9], })
>> begin-constraint-propagate (IlcAllDiff(009DE938) {S[1..9], E[0..9], N[0..9],
D[0..9], M[1..9], O[0..9], R[0..9], Y[0..9], })
>> end-constraint-propagate (IlcAllDiff(009DE938) {S[1..9], E[0..9], N[0..9],
D[
0..9], M[1..9], O[0..9], R[0..9], Y[0..9], })
>> begin-constraint-post (((IlcArraySumI(009DEC30)[1000..9918] -
IlcArraySumI(00
9DEBD8)[9000..89910]) == Y[0..9]))
>> end-constraint-post (((IlcArraySumI(009DEC30)[1000..9918] -
IlcArraySumI(009D
EBD8)[9000..89910]) == Y[0..9]))
>> begin-constraint-propagate (((IlcArraySumI(009DEC30)[1000..9918] -
IlcArraySu
mI(009DEBD8)[9000..89910]) == Y[0..9]))
>> end-constraint-propagate (((IlcArraySumI(009DEC30)[9000..9918] -
IlcArraySumI
(009DEBD8)[9000..10710]) == Y[0..9]))
*/
```

There you see the constraint posting and the steps of the constraint propagation algorithm.

Tracing variable processing

As Solver propagates constraints, it maintains them in a queue. Tracing the propagation queue shows the order in which variables are processed by the Solver propagation algorithm. In the following example, we create a trace to trace the constraint propagation queue using the ILOCPTRACEWRAPPER0 macro, like this:

```
ILOCPTRACEWRAPPER0(PrintAllVars, solver) {
```

Then we turn on the trace mode, like this:

```
  solver.setTraceMode(IlcTrue);
```

Next, use `IlcTraceVars` as a flag to the constructor `IlcPrintTrace`. It causes a message to be printed each time a demon or constraint is executed by the propagation algorithm:

```
IlcPrintTrace trace(solver, IlcTraceVars);
```

Then we set that hook as the global trace hook by using the member function `IloSolver::setTrace`, like this:

```
solver.setTrace(trace);
```

Here are those steps in the context of a complete program.

```
ILOCPTRACEWRAPPER0(PrintAllVars, solver) {
  solver.setTraceMode(IlcTrue);
  IlcPrintTrace trace(solver, IlcTraceVars);
  solver.setTrace(trace);
}

int TraceAll2(){
  IloEnv env;
  try {
    IloModel mdl(env);
    IloNumVar S(env, 0, 9, ILOINT, "S"),
      E(env, 0, 9, ILOINT, "E"),
      N(env, 0, 9, ILOINT, "N"),
      D(env, 0, 9, ILOINT, "D"),
      M(env, 0, 9, ILOINT, "M"),
      O(env, 0, 9, ILOINT, "O"),
      R(env, 0, 9, ILOINT, "R"),
      Y(env, 0, 9, ILOINT, "Y");
    IloNumVarArray vars (env, 8, S, E, N, D, M, O, R, Y);

    mdl.add(S != 0);
    mdl.add(M != 0);

    IloConstraint alldiff=IloAllDiff(env,vars);
    mdl.add(alldiff);
    mdl.add(  1000*S + 100*E + 10*N + D
      + 1000*M + 100*O + 10*R + E
      == 10000*M + 1000*O + 100*N + 10*E + Y);

    IloSolver solver(mdl);
    solver.addTrace(PrintAllVars(env));
    solver.solve(IloGenerate(env, vars));
  }
  catch (IloException ex) {
    cerr << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

That program output is:

```
>> begin-set-min: S[0..9] 1
>> end-set-min: S[1..9] 1
>> begin-set-min: M[0..9] 1
>> end-set-min: M[1..9] 1
>> begin-set-min: S[1..9] 9
>> end-set-min: S[9] 9
>> begin-set-max: M[1..9] 1
>> end-set-max: M[1] 1
>> begin-set-max: O[0..9] 1
>> end-set-max: O[0..1] 1
>> begin-set-max: E[0..9] 8
>> end-set-max: E[0..8] 8
>> begin-set-max: N[0..9] 8
>> end-set-max: N[0..8] 8
>> begin-set-max: D[0..9] 8
>> end-set-max: D[0..8] 8
>> begin-set-max: R[0..9] 8
>> end-set-max: R[0..8] 8
>> begin-set-max: Y[0..9] 8
>> end-set-max: Y[0..8] 8
>> begin-set-max: O[0..1] 0
>> end-set-max: O[0] 0
>> begin-remove-value: E[0..8] 1
>> end-remove-value: E[0 2..8] 1
>> begin-remove-value: N[0..8] 1
>> end-remove-value: N[0 2..8] 1
>> begin-remove-value: D[0..8] 1
>> end-remove-value: D[0 2..8] 1
>> begin-remove-value: R[0..8] 1
>> end-remove-value: R[0 2..8] 1
>> begin-remove-value: Y[0..8] 1
>> end-remove-value: Y[0 2..8] 1
>> begin-set-min: N[0 2..8] 2
>> end-set-min: N[2..8] 2
>> begin-set-min: D[0 2..8] 2
>> end-set-min: D[2..8] 2
>> begin-set-min: E[0 2..8] 2
>> end-set-min: E[2..8] 2
>> begin-set-min: R[0 2..8] 2
>> end-set-min: R[2..8] 2
>> begin-set-min: Y[0 2..8] 2
>> end-set-min: Y[2..8] 2
>> begin-set-min: N[2..8] 3
>> end-set-min: N[3..8] 3
>> begin-set-max: E[2..8] 7
>> end-set-max: E[2..7] 7
>> begin-set-min: E[2..7] 3
>> end-set-min: E[3..7] 3
>> begin-set-min: N[3..8] 4
>> end-set-min: N[4..8] 4
>> begin-set-min: E[3..7] 4
>> end-set-min: E[4..7] 4
>> begin-set-min: N[4..8] 5
>> end-set-min: N[5..8] 5
IlcOr   : 1
>> begin-set-value: E[4..7] 4
>> end-set-value: E[4] 4
```

```
>> begin-remove-value: D[2..8] 4
>> end-remove-value: D[2..3 5..8] 4
>> begin-remove-value: R[2..8] 4
>> end-remove-value: R[2..3 5..8] 4
>> begin-remove-value: Y[2..8] 4
>> end-remove-value: Y[2..3 5..8] 4
>> begin-set-min: D[2..3 5..8] 8
>> end-set-min: D[8] 8
>> begin-set-min: R[2..3 5..8] 8
>> end-set-min: R[8] 8
>> begin-set-max: N[5..8] 5
>> end-set-max: N[5] 5
>> begin-set-max: Y[2..3 5..8] 2
>> end-set-max: Y[2] 2
>> begin-set-min: E[4..7] 5
>> end-set-min: E[5..7] 5
>> begin-set-min: N[5..8] 6
>> end-set-min: N[6..8] 6
IlcOr   : 2
>> begin-set-value: E[5..7] 5
>> end-set-value: E[5] 5
>> begin-remove-value: D[2..8] 5
>> end-remove-value: D[2..4 6..8] 5
>> begin-remove-value: R[2..8] 5
>> end-remove-value: R[2..4 6..8] 5
>> begin-remove-value: Y[2..8] 5
>> end-remove-value: Y[2..4 6..8] 5
>> begin-set-min: D[2..4 6..8] 7
>> end-set-min: D[7..8] 7
>> begin-set-min: R[2..4 6..8] 8
>> end-set-min: R[8] 8
>> begin-set-max: N[6..8] 6
>> end-set-max: N[6] 6
>> begin-set-max: Y[2..4 6..8] 3
>> end-set-max: Y[2..3] 3
>> begin-set-max: D[7..8] 7
>> end-set-max: D[7] 7
>> begin-set-max: Y[2..3] 2
>> end-set-max: Y[2] 2
```

The choice points and the failures are now visible.

In the following example, we associate the trace only with a single variable, `E`.

```
ILOCPTRACEWRAPPER1(TraceVarArray, solver, IloIntVarArray, vars) {
  IlcIntVarArray svars=solver.getIntVarArray(vars);
  solver.setTraceMode(IlcTrue);
  IlcPrintTrace trace(solver, IlcTraceVars);
  trace.trace(svars[1]);
}


int TraceAll3(){
  IloEnv env;
  try {
    IloModel mdl(env);
    IloIntVar S(env, 0, 9, "S"),
      E(env, 0, 9, "E"),
      N(env, 0, 9, "N"),
      D(env, 0, 9, "D"),
      M(env, 0, 9, "M"),
      O(env, 0, 9, "O"),
      R(env, 0, 9, "R"),
      Y(env, 0, 9, "Y");
    IloIntVarArray vars (env, 8, S, E, N, D, M, O, R, Y);

    mdl.add(S != 0);
    mdl.add(M != 0);

    IloConstraint alldiff=IloAllDiff(env,vars);
    mdl.add(alldiff);
    mdl.add(  1000*S + 100*E + 10*N + D
      + 1000*M + 100*O + 10*R + E
      == 10000*M + 1000*O + 100*N + 10*E + Y);

    IloSolver solver(mdl);
    solver.addTrace(TraceVarArray(env, vars));
    solver.solve(IloGenerate(env, vars));
  }
  catch (IloException ex) {
    cerr << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

The output becomes:

```
>> begin-set-max: E[0..9] 8
>> end-set-max: E[0..8] 8
>> begin-remove-value: E[0..8] 1
>> end-remove-value: E[0 2..8] 1
>> begin-set-min: E[0 2..8] 2
>> end-set-min: E[2..8] 2
>> begin-set-max: E[2..8] 7
>> end-set-max: E[2..7] 7
```

```
>> begin-set-min: E[2..7] 3
>> end-set-min: E[3..7] 3
>> begin-set-min: E[3..7] 4
>> end-set-min: E[4..7] 4
IlcOr   : 1
>> begin-set-value: E[4..7] 4
>> end-set-value: E[4] 4
IlcFail : 1
>> begin-set-min: E[4..7] 5
>> end-set-min: E[5..7] 5
IlcOr   : 2
>> begin-set-value: E[5..7] 5
>> end-set-value: E[5] 5
```

The following program traces all modifications of all variables. To get that effect, we turn on the trace mode and use the flag `IlcTraceAll` when we construct the trace.

```
ILOCPTRACEWRAPPER0(PrintAllTrace, solver) {
  solver.setTraceMode(IlcTrue);
  IlcPrintTrace trace(solver, IlcTraceAll);
  solver.setTrace(trace);
}

int TraceAll4(){
  IloEnv env;
  try {
    IloModel mdl(env);
    IloIntVar S(env, 0, 9, "S"),
      E(env, 0, 9, "E"),
      N(env, 0, 9, "N"),
      D(env, 0, 9, "D"),
      M(env, 0, 9, "M"),
      O(env, 0, 9, "O"),
      R(env, 0, 9, "R"),
      Y(env, 0, 9, "Y");
    IloIntVarArray vars (env, 8, S, E, N, D, M, O, R, Y);

    mdl.add(S != 0);
    mdl.add(M != 0);

    IloConstraint alldiff=IloAllDiff(env,vars);
    mdl.add(alldiff);
    mdl.add(  1000*S + 100*E + 10*N + D
      + 1000*M + 100*O + 10*R + E
      == 10000*M + 1000*O + 100*N + 10*E + Y);

    IloSolver solver(mdl);
    solver.addTrace(PrintAllTrace(env));
    solver.solve(IloGenerate(env, vars));
  }
  catch (IloException ex) {
    cerr << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

The output is too lengthy to show entirely, but it starts like this:

```
>> begin-constraint-propagate ((S[0..9] != 0))
>> begin-set-min: S[0..9] 1
>> end-set-min: S[1..9] 1
>> end-constraint-propagate ((S[1..9] != 0))
>> begin-constraint-propagate ((M[0..9] != 0))
>> begin-set-min: M[0..9] 1
>> end-set-min: M[1..9] 1
>> end-constraint-propagate ((M[1..9] != 0))
>> begin-constraint-post (IlcAllDiff(00ACED48) {S[1..9], E[0..9], N[0..9],
D[0..
9], M[1..9], O[0..9], R[0..9], Y[0..9], })
>> end-constraint-post (IlcAllDiff(00ACED48) {S[1..9], E[0..9], N[0..9],
D[0..9]
, M[1..9], O[0..9], R[0..9], Y[0..9], })
>> begin-constraint-propagate (IlcAllDiff(00ACED48) {S[1..9], E[0..9],
N[0..9],
D[0..9], M[1..9], O[0..9], R[0..9], Y[0..9], })
>> end-constraint-propagate (IlcAllDiff(00ACED48) {S[1..9], E[0..9],
N[0..9], D[
0..9], M[1..9], O[0..9], R[0..9], Y[0..9], })
>> begin-constraint-post ((IlcArraySumI(00ACF070)[1000..9099] ==
(IlcArraySumI(0
0ACEF38)[9000..89919] - 91 * E[0..9])))
>> end-constraint-post ((IlcArraySumI(00ACF070)[1000..9099] ==
(IlcArraySumI(00A
CEF38)[9000..89919] - 91 * E[0..9])))
>> begin-constraint-propagate ((IlcArraySumI(00ACF070)[1000..9099] ==
(IlcArrayS
umI(00ACEF38)[9000..89919] - 91 * E[0..9])))
>> begin-set-min: S[1..9] 9
>> end-set-min: S[9] 9
>> begin-set-max: M[1..9] 1
>> end-set-max: M[1] 1
>> begin-set-max: O[0..9] 1
>> end-set-max: O[0..1] 1
>> end-constraint-propagate ((IlcArraySumI(00ACF070)[9000..9099] ==
(IlcArraySum
I(00ACEF38)[9000..10719] - 91 * E[0..9])))
>> begin-var-process: S[9]
>> begin-demon-process (IlcAllDiffDemon)
>> begin-set-max: E[0..9] 8
>> end-set-max: E[0..8] 8
>> begin-set-max: N[0..9] 8
>> end-set-max: N[0..8] 8
>> begin-set-max: D[0..9] 8
>> end-set-max: D[0..8] 8
>> begin-set-max: R[0..9] 8
>> end-set-max: R[0..8] 8
>> begin-set-max: Y[0..9] 8
>> end-set-max: Y[0..8] 8
>> end-demon-process (IlcAllDiffDemon)
>> begin-demon-process ((IlcArraySumI(00ACF070)[9000..9088] ==
(IlcArraySumI(00
ACEF38)[9000..10628] - 91 * E[0..8])))
>> begin-set-max: O[0..1] 0
>> end-set-max: O[0] 0
```

```
>> end-demon-process ((IlcArraySumI(00ACF070)[9000..9088] ==
...
```

The examples in this chapter show you some possibilities of the tracing mode in Solver. You should choose the level of detail most useful for your application: the more detail, the longer the output, and the longer its analysis, of course.

Creating your own trace

You can also create your own trace by subclassing the virtual functions of `IlcTraceI`. Here is an example:

```
class MyTraceI : public IlcTraceI {
public:
  MyTraceI(IloSolver solver, IlcUInt flags, const char* name=0) :
      IlcTraceI(solver.getManager().getImpl(),flags,name){}
      ~MyTraceI(){}

      // virtual functions
      void beginSetMinIntVar(const IlcIntExp var, IlcInt min){
        var.getSolver().out() << "mySetMinIntVar";
        print(var,min);
      }
      void beginSetValueIntVar(const IlcIntExp var, IlcInt val){
        var.getSolver().out() << "mySetValue";
        print(var,val);
      }

      void print(const IlcIntExp var, IlcInt val){
        var.getSolver().out()<< var << " val:" << val << endl;
        IlcGoalI* goal=var.getSolver().getActiveGoal().getImpl();
        IlcDemonI* demon=getActiveDemon().getImpl();
        if (goal == 0) var.getSolver().out() << "no active goal" << endl;
        else var.getSolver().out() << "goal:" << *goal << endl;
        if (demon == 0) var.getSolver().out() << "no active demon" << endl;
        else {
          if (demon->isAConstraint()){
            var.getSolver().out() << "the active demon is the constraint:" <<
*demon << endl;
          } else {
            var.getSolver().out() << "propagation of demon" << endl;
            if (demon->getConstraintI() == 0){
              var.getSolver().out() << "the demon has no constraint associated
with" << endl;
            } else {
              var.getSolver().out() << "the associated constraint with the
demon is:";
              var.getSolver().out() << *(demon->getConstraintI()) << endl;
            }
          }
        }
      }
};
```

```
class MyTrace : public IlcTrace {
public:
  MyTrace(IloSolver solver, IlcUInt flags=4, const char* name=0)
    : IlcTrace() {
      _impl = new (solver.getHeap()) MyTraceI(solver, flags, name);
    }

    MyTrace(MyTraceI* impl) {
      _impl = impl;
    }
    ~MyTrace(){}
    MyTraceI* getImpl() const {
      return (MyTraceI*)_impl;
    }
    const MyTrace& trace(IlcIntVar var) const{
      getImpl()->trace(var);
      return *this;
    }
    void operator=(const MyTrace& tr){
      _impl = tr._impl;
    }
};
```

This trace must be wrapped using the macro ILOCPTRACEWRAPPER. (See "Tracing variable processing" on page 600 for an example of how to do this.) It can then be called in an application, as follows:

```
  MyTrace trace(solver,IlcTraceAllEvent);
  trace.trace(svars[1]);
```

This produces the following output:

```
mySetMinIntVarE[0 2..8] val:2
no active goal
propagation of demon
the associated constraint with the demon is:IlcAllDiff(00ACED48) {S[9], M[1],
N[2..8], D[2..8], E[0 2..8], O[0], R[0 2..8], Y[0 2..8], }
mySetMinIntVarE[2..7] val:3
no active goal
the active demon is the constraint:(IlcArraySumI(00ACF070)[9022..9088] ==
(IlcArraySumI(00ACEF38)[9272..9728] - 91 * E[2..7]))
mySetMinIntVarE[3..7] val:4
no active goal
the active demon is the constraint:(IlcArraySumI(00ACF070)[9022..9088] ==
(IlcArraySumI(00ACEF38)[9362..9728] - 91 * E[3..7]))
IlcOr   : 1
mySetValueE[4..7] val:4
goal:IlcIntVarSetValue
no active demon
IlcFail : 1
mySetMinIntVarE[4..7] val:5
goal:IlcIntVarRemoveValue
no active demon
IlcOr   : 2
mySetValueE[5..7] val:5
```

```
goal:IlcIntVarSetValue
no active demon
```

C H A P T E R

35

Developing Applications

When you are developing an application, of course, you are concerned with a great many more parts than simply the solution to the problem. No doubt you also have to handle such aspects as the user interface, connections to databases, networking with other systems, and so forth. In short, developing an application with Solver is just one activity among many others relevant to a highly technical project.

Perhaps your first difficulty is to define the objectives of a given application. In as much as these objectives may evolve during development, they may prove harder to achieve than predicted.

For those reasons, we recommend an *incremental* method of development. With respect to Solver an incremental development cycle entails these phases:

1. Describe the problem.
2. Specify the objectives of the application.
3. Design a model and prototype the application.
4. Implement and optimize the application.

Obviously, these phases are not entirely independent of one another. Developers find themselves returning from one phase to another as they work. In fact, the prototyping phase frequently calls the entire cycle into question as the end-user reacts to the initial model, discovers that some constraint has not yet been formulated, and brings the cycle back to phase one; often the end-user is wishing at the same time to be able to modify a solution

interactively and thus hoping to return to the specification phase as well. Even phase 4, when the application is exercised with real data, can call the entire design into question if performance proves poor. In short, a developer can expect several loops among these phases throughout a typical development cycle.

This chapter acknowledges the kinds of difficulties a developer encounters while creating an application that exploits Solver, and it offers a few hints for avoiding or overcoming these difficulties.

Describing the problem

When we consider the aspects of an application that involve constraints, there are a number of important points to keep in mind.

- We need a complete and exhaustive statement of all the *constraints*, as a minimal part of the problem description.
- We need a way to identify the “soft” constraints (that is, those constraints that we would like to satisfy but that we may relax or drop altogether if the problem proves intractable).
- We must clearly separate the constraints themselves from any strategies we might adopt to solve the problem. This principle of separating the constraints from the strategies is part of the general rule of constraint programming that emphasizes making the problem representation independent of the solution search.

Define all constraints

For example, in the design of a telecommunication network, the problem description might demand that every pair of nodes in the network must have at least two paths between those nodes for robustness and reliability.

To meet that demand, we’ll design a model, taking into account that constraint on every pair of nodes. Let’s say we implement a first-cut of our model, and Solver finds a solution for us—a solution that is quite logical: it consists of a huge loop, passing through every node. When we show this solution to our clients, we learn, alas, that solution is not acceptable in practice, so we go back to the phase of describing our problem again, and we add constraints on the length of paths between pairs of nodes. In consequence of this change in the *problem description*, we need to change the *model* and the *implementation*.

A better problem description in the first place—one that included all the constraints, especially those on the lengths of paths between pairs of nodes—might have spared us this repetition. Certainly a sound initial problem description is the surest safeguard against the infamous encroachments of an incremental specification.

Identify soft constraints

As we indicated in our list, we must identify the "soft" constraints in the problem. The soft constraints are those that we would like to satisfy, but that we might relax or drop if the problem proves intractable. These are the constraints that might lead to the elimination of potential solutions if we insist on imposing them. At the same time, these soft constraints are typically the constraints that, if we fail to take them into account, we'll come up with a "solution" that we cannot exploit in reality because the soft constraints embody some fundamental and practical aspect of the problem.

When we separate the soft from the hard constraints, we are creating the means to accomplish two tasks:

- to qualify acceptable solutions;
- to measure the quality of proposed solutions.

This work—separating hard and soft constraints so we can compare the adequacy of various solutions—is often very difficult, especially when the constraints represent social rules, accumulated and elaborated over time, such social rules as customary working hours, conventional combinations of tasks, habitual job assignments, and so forth. If we ignore such constraints, we are likely to find unacceptable solutions; yet if we enforce such constraints, we may fail to find a solution at all. In any case, the nature of such constraints have an obvious impact on the model and implementation since they influence the cost function.

Separate constraints from strategies

At times, the expertise needed to resolve a problem may be mixed up in the problem specification. As we describe the problem, design a model, formulate the problem representation, and implement it, we need to recognize the kind of problem-expertise that we could exploit to reach a good solution, but we also need to distinguish that expertise from the real constraints of the problem. Basically, we do not want to complicate a problem unnecessarily with superfluous constraints; rather, we want to make the most of any expertise that could be useful in reaching better solutions.

Specifying objectives

Applications built with Solver span a wide spectrum of tools that aid decision-making. One way of characterizing them is that they range between two extremes:

- "black box" applications that simply go out and automatically find solutions;
- interactive applications that simulate a situation so that user can easily ask "what if?" and see the consequences.

Black-box and user acceptance

How does the choice of a black-box style influence the model and implementation of a program using Solver? It's true that simply solving the problem, like a black box, is often the expressed goal of a new project. However, experience has repeatedly shown that user-acceptance is low for black-box applications, in large part because such an opaque model does not inspire user-confidence. Users cannot see what is going on and consequently doubt the solution from a black-box application.

Response time

Response time is also a sensitive issue with respect to user-acceptance of an application, and certainly response time is an issue that greatly complicates implementation. In a black-box application, response time is a conspicuous issue because the end-user simply waits for the solution (or non-solution) of the problem.

One way of addressing this difficulty is to limit the amount of time spent searching for a solution, and in the case of failure, to provide an explanation for the failure. While it may be more or less easy to implement an application that satisfies limits on the amount of time needed to find a solution, it is seldom easy to implement a module that explains why the application failed to find a solution.

Interactive applications and user-modifiable constraints

At the opposite extreme from a black box are those completely interactive applications that let the end-user play the role of an intelligent genie who guides the solution search and who intervenes to modify the solution, or even the constraints to get a better solution. How does this kind of application influence the model and implementation for a program using Solver?

In designing the model for such an application, you, the developer, definitely have to distinguish those constraints that are susceptible to user-modification. And certainly the way that the application manages this user-interaction has a direct impact on the implementation and solution procedures.

Designing the model and prototyping

While the phase of designing a model is usually presented as quite distinct from the prototyping phase, the two are so integrally related that we really should discuss them together.

Like any other phase, these two have the dual purpose of validating the preceding steps and serving as the take-off point for the next phase. In fact, they represent the moment at which the specifications of the application really become visible. For that reason, the end-user is all

the more crucial at this juncture in the development, especially if this application is new or one that is insufficiently specified.

Extract the miniproblem

Before you actually embark on the total model, it's a good idea to experiment with small-scale models. In that way, you'll acquire useful knowledge about the problem, and you'll get this information early enough in the development cycle to be useful later. Here are a few general points (though by no means all) about this experimental period.

In the model-prototype, the first decision to make is how to extract a miniproblem from the overall problem. The miniproblem provides the basis for a first-cut model of the whole problem. In practice, it's important to get a first-cut model as early as possible. That model serves two purposes: as a validation field for the specifications and as an experimental domain for the feasibility of the project. In this context, it's thus critical to choose a representative miniproblem. If the miniproblem that you extract is too simple, it won't show you how the application will perform. If it's too complicated, it will demand too much work—very possibly wasted effort if the model reveals a flaw in the preceding phases. In other words, the choice of a miniproblem has serious consequences. Your general experience and your knowledge of similar applications will greatly facilitate your choice. One fallback choice—sometimes the essential choice—is to use a small-scale instance of the entire problem.

Once the problem is well defined, it's up to you to design a sound model for it. Fortunately, with Solver, you can use the same language for the entire set of tasks: designing the model, prototyping the application, and implementing the project. In this way, the model itself will immediately yield the first prototype.

Decompose the model

First of all, if the model can be logically decomposed, you should do so. You can decompose the model in the same way that you partition the variables into subsets with few constraints linking the variables of different subsets. It is really important to find such a decomposition and to carry it out because the decomposition will save an exponential factor in the solution time.

Determine precision

After decomposition, the next issue to address is the degree of precision. Clearly, the degree of precision (that is, the granularity of the definition and solution) needs to be adapted to the objectives of the application. To put the issue crudely, there is no point in counting seconds if your problem works in terms of years. In short, choose an appropriate unit, both for the problem representation and for the solution.

Respect semantics

Also adapt the language of your application to the language of the problem itself. Doing so will make communication with your end-user much easier. It will also insure that your constraints are relevant to the problem, and in the long run, it will make maintenance easier as well. Even more important, the natural structure of the problem domain can lead directly to good models for the application. By good models, in this context we mean compact, flexible, and understandable. Always stay close to the semantics of the problem. Yes, always.

An object-oriented approach distinguishes *structural* constraints (inherent in a class) from *specific* constraints (intrinsic to an instance). This distinction offers a powerful tool for organizing the constraints expressed by an end-user.

We recommend that you create the first model as closely as possible to the way the end-user initially formulated the problem.

Avoid symmetry

We strongly recommend that your model should not introduce symmetries that do not already exist in reality. Moreover, one of the purposes of successive prototypes is to eliminate symmetries in order to reduce the search space and to represent equivalent solutions uniquely. In constraint programming generally, symmetries in the model introduce greater computation time in the search for solutions and consequently seriously hamper performance.

This step—eliminating these kinds of symmetries—is obligatory before you can estimate execution time. If you know already on the basis of the model that the execution time is exponential with respect to the number of data (usually the number of principal variables), various trials with data sets of different sizes will give you an estimate of the constant factor in the exponential complexity.

Unfortunately, the case sometimes occurs where you cannot eliminate symmetries in the model itself; you have to eliminate them during the search for solutions. In other words, those symmetries cannot be taken into account beforehand but only afterwards.

Prototype rapidly and cleanly

In order to take full advantage of rapid prototyping during development, you need, of course, to master performance and master the specifications.

There's an easy way to arrive at the stage of rapid prototyping when you are using Solver: simply add a process to your model to generate solutions. You'll then have a working prototype that you can use in various practical ways:

- to have your end-user validate the constraints;

◆ to validate data with constraints.

Validate the model

This kind of work—validation—in the development cycle is greatly simplified if a solution is already available to you. With a known solution in hand, you can check the way that your constraints have been written so that you will be alerted to any errors in interpretation or any bug in translation between the specifications and the implementation. You can also check the global coherence of your initial constraints, their coherence as a set. That is, when you have a known solution already, you can use it as a reference point in testing and verification.

Once you have validated it, your first model itself will play the role of a reference. It will enable you to test new solutions that you get from the implementations you develop.

Since your first model will play this role in verification, you really should not use any idiosyncratic tricks or “dirty” coding practices at this point. Such practices will prove troublesome later when you try to modify this first prototype. Moreover, such tricks diminish confidence in the robustness of any solutions that you get. How do you avoid the sort of programming tricks that we counsel against here? One way is to work in a plain, declarative style. Don’t create new, unconventional constraints; instead, use the predefined constraints you’ll find ready in Solver.

Choose types of variables

Your choice of *types* for variables will also be important in your model. The right kind of variables can be coded directly from properties of the model. Think, for example, in terms of constrained set variables. According to the types of variables that you choose, you’ll be able to express constraints more or less easily.

Use symbolic constraints as much as possible; by symbolic constraints, we mean such constraints as whether a variable is an element of a set, or what the cardinality of a set may be.

Symbolic constraints often enable you to reduce the number of variables in the program. Such a reduction in variables will be useful since the apparent search space is exponential in terms of the number of variables, but it is only polynomial with respect to the size of their domains.

Experiment with models

In spite of these common sense rules, there are many ways to design the model and create the prototype for any given problem. You’ll need to try more than one model to find the best fit for your problem, but don’t think of these trials as wasted effort. The time that you spend in designing the model, even time spent “playing around” with it, is time well spent since that time will be saved in later phases of implementation.

Among the fruitful experiments that you may want to conduct in this context is to reverse the roles of variables and values.

Implementing and optimizing

Since Solver encourages you to use the same language for the model, the prototype, and the implementation, there is not a clear line of demarcation between designing the model and implementing the program. The most that we can say about this subject is that prototyping is inherently a repetitive activity. Implementation and optimization of the implementation begin as soon as you start to improve the model that you've created.

While there may be many aspects of implementation and optimization to carry out for your overall application (for example, improving the user-interface, database connections, network transactions, etc.) we'll focus here only on implementation and optimization strictly related to Solver. We'll start by reviewing the obvious: there is no point in optimizing too soon in the development cycle. The gymnastics that you may go through to improve performance marginally in the early phases of development may very well have little or no effect on performance in the finished product while they may very well cause you real difficulty in later phases of development as you try to make necessary modifications.

Furthermore, (and this point should be equally obvious) there is no point in optimizing just any aspect of your application in wild hope of changing characteristics specific to Solver.

There are, however, practical ways to optimize with Solver.

Use multiple data sets

First, we remind you that the solution of a combinatoric problem is quite sensitive to variations in data, so you need to optimize a Solver program with respect to multiple sets of data to have any reliable effect at all. In fact, the robustness of your program will depend heavily on tests run over several sets of data. Such tests should induce redundant constraints that will vary greatly according to the particular instance of the problem under consideration.

This point about using multiple sets of data to test your program is so important that we'll insist that if you don't have access to real data, you should consider generating realistic data through random variation.

At this point in the development cycle, you should also settle once and for all the format of data. If, for example, it is straightforward and quick to sort an array by posing a few constraints, obviously it will be even quicker to use a conventional sorting technique. This guideline can be generalized: most ordinary preprocessing (unrelated to constraint programming) can be handled more efficiently in C++ than in Solver.

Optimize propagation and search

The areas to optimize in your program are the *constraint propagation* and the *solution search*. Improvements in those two areas are, of course, interdependent, and the greatest gains in efficiency come about when you work simultaneously on both aspects.

Look at changes

To help you see what you're doing, it's a good idea to write an interface that shows the modifications of variables and the points where backtracking occurs. Such an interface lets you rapidly distinguish which variables to take into account early in the solution search. It also shows you the areas where propagation is weakest, areas that you should re-enforce with redundant constraints.

Use active constraints

Possibly you will have to write new constraints to express conditions specific to your application, or at least to express them more efficiently, more directly. In such a case, you should use an *active* constraint, that is, one that propagates everything to all the variables that it contains.

Set granularity

If you write your own constraint (rather than exploiting predefined constraints from Solver), you also need to think about the granularity of propagation in your application. In other words, you must define the hooks between constraints and events that modify variables. If you hook onto only the value event, you may miss out on certain propagations; in contrast, if you hook onto every modification with the domain event, you'll invoke the constraint so frequently that you'll benefit little if at all from domain reductions, thus penalizing the global resolution time.

Whether you are using predefined Solver constraints or writing your own, you can also choose the granularity of the solution search. In the algorithms that Solver uses, you can decide how to split up the search space at the level of the chosen variable. The standard choice `IloInstantiate` works on the minimal bound of the chosen variable; `IloDichotomize` works on the median (average) value of the chosen variable.

Your choice about this issue should be guided by experimentation; in other words, you should try more than one strategy and measure the results. The reason that experimentation is so important for this issue is that the best choice depends on information propagated as a function of events.

For example, when most of the constraints are arithmetic and are consequently propagated on the event range, binary generation using `IloDichotomize` is usually the most efficient.

In contrast, if most of the constraints propagate only on value events, it's a better idea to *generate* the values of the variables.

Use expertise

Professional knowledge about the problem itself proves most useful in writing a search algorithm when the standard procedures prove insufficient. In real-world problems (unlike purely theoretical problems) the hardest parts are actually often confined to one sole part of the model. In consequence, you have to start searching for a partial solution in that region in order to minimize the exponential factor.

For example, when you're building a schedule, you have to assign the most difficult or the most critical items first. The easier parts fall into place around them.

This kind of solution clearly shows that the procedure for solving a problem depends very much on the instance of the problem under consideration, not merely on its formal statement.

Preprocess variables

The primitive algorithm `IloGenerate` chooses variables separately in the order that they are given. This convention can be extremely useful when you've already preprocessed the variables in some appropriate way to put them into the "right" order.

Of course, this tactic is inadequate if the order itself depends on the states of other variables. Such a dynamic order (one depending on other variable states) can be expressed straightforwardly in Solver by means of the evaluation functions, `IlcChooseXXX`.

Obviously, the criteria for evaluation must be pertinent; for example, the domain *size* of binary variables (whose values are either 0 or 1, and thus the size of whose domain is at most two) would not work well here.

Exploit different strategies

At some point in the development cycle, you'll need to think about the criteria for choosing variables and values. These choices are so specific to the problem under consideration, sometimes even specific to the end-user, that there are no "canned" rules ready to use "off-the-shelf." You really must exploit professional expertise from the problem domain.

Solver treats optimization problems by enforcing the idea that the current solution must cost less than the preceding solution at each iteration.

The essential factor is in the propagation of the cost variable; at every moment, its boundaries must be the best evaluation possible of the cost of the solution partially examined.

We cannot say often enough that “optimality at all costs” is frequently unrealistic or even impossible. To get a good solution in a reasonable amount of time depends above all on the generation strategy. The behavior that you want, of course, is to get significantly better solutions very fast. With such an aim, it may be quite relevant to impose a rule such as, “the current solution must be 10% better than the next.” Or, similarly, for example, it may be a good idea to eliminate once and for all the solutions close to the one already obtained by removing the values near the value belonging to the most recent solution.

Optimize locally

To exploit the possibilities of *local* optimization, you can take advantage of the Solver library.

In fact, you can, at some point, leave aside the backtracking mechanism and concentrate on local optimizations. Starting from a solution with a certain cost, Solver will improve that solution up to the point of getting a satisfactory solution.

This technique has the additional advantage of working remarkably well on very large instances of problems.

As a technique, it is extremely simple in principle. Solver uses generation and propagation to get an initial solution; this solution is then improved by local modifications. The problem constraints, of course, are used to guarantee that the iterations for improvements stay within the authentic solution space.

A few last words

In this chapter, we hoped to bring together the main principles actually put into practice on a daily basis by real developers using Solver.

This chapter is necessarily incomplete. We haven’t even touched on problems of over-constraint, nor replanning. A good example of replanning, where a given plan is changed to take account of modifications in the available resources, appears online in the standard distribution in the subdirectory of Solver examples in the file `replan.cpp`.

We hope that the overall impression you’ve gotten from this chapter of “recipes” is that developing applications with Solver is intimately linked to the activity of prototyping.

Index

A

accessing
  attributes of set elements **385** to **386**
  information about constrained variables **335**
activating
  constraints **279**, **619**
  demon when user-defined variables are bound **356**
  trace functions **599**
active path **239**
`add` member function
  `IloModel` class **48**, **94**
adding
  objects to model **46**
algorithm
  greedy improvement **411**
  minimization **279**
  optimal solution **278**
  propagation **337**
algorithms
  evolutionary **491**
  filtering **215**
  genetic **491**
  search **29**
apparent complexity
  dual representation and **582**
  example **582**
  fewer variables, larger domains **582**
arc consistency **590**
arithmetic expression
  conforming to IEEE 754 **203**
array **57**
  constrained integer variables **57**
  extensible **57**
assigning frequencies **209**

B

backtrack **36**
bin packing problem **139**, **541**
binarization of search tree **238**
binding
  counters **183**
  propagation events and **337**
bound communication **288**
boundary
  modifiers and **333**
  propagation events and **337**, **339**
branch **238**
  search tree **34**
branch & bound **278**
Building Warehouses **466**
  complete program **460**
  output **465**

C

canonical expressions **197**
car sequencing **557**
car sequencing problem **103**

cardinality
example **387**, **388**, **585**
sets and **584**
changing money problem **55**
child node **239**
choice criteria **358**, **620**
example: Assigning Frequencies **358**
in development cycle **620**
predefined **278**
choice point **76**
labels and **307**
setting **307**
tracing **599**
class constraint **144**
definition **592**
example: Plugging Cards **182**
clique **207**
largest **208**
maximal **208**
closed node **238**
comparator **498**, **525**
complete program
Building Warehouses **460**
complexity
apparent **582**
dual representation and **582**
fewer variables, larger domains **582**
symbolic constraint and **355**
configuration
example: Configuring Tankers **381**
set variables and **381**
constrained data member **591**
constrained floating-point variable
declaring **190**
definition **189**
precision **189**
constrained integer variables **46**
array **57**
constrained set variable
example **381**
constrained set variables **122**
constrained variable
trace hooks for **599**
constrained variables **31**
constraint **30**, **31**
activating **279**, **619**
add to model **48**
class **144**, **182**, **592**
counting **590**
creating new **340**
defining new class of **338**
defining the invariant **336**
delaying propagation **352**
differs from metaconstraint **345**
element **182**
filter levels **215**
generic **181**, **182**, **592**
global **215**
inherent **616**
not a test **279**
operators for combining **190**
order among **346**
predefined **74**, **106**, **123**, **144**
propagation algorithm **337**
range **147**
redundant **183**, **590**
relaxing **88**
stepping through propagation of **597**
structural **616**
writing new class of **332**
constraint programming **29**
constraint propagation **30**, **215**
during search **32**, **35**
initial **32**, **33**
controlling choice in solution search **74**
cost **227**, **232**
cost function
back propagating **585**
minimizing **178**
counters
binding **183**
counting constraint **590**
creating
new constraints **340**
crew scheduling problem **117**
crossover **545**
current domain **35**
current solution **410**
custom selector
selector

custom **521**
customer support **24**
customizable goal **375**

D

data
  input **230**
data from file **314**
data member
  constrained **591**
  reversible **591**
decision variables **30**, **31**, **47**
declaring
  constrained floating-point variables **190**
decoder **493**, **557**
decoding goal **561**
decompose problem **256**
decreasing best fit **543**
default goal **48**
defining
  invariant of constraints **336**
  new class of constraints **338**
  new class of metaconstraints **343**
  selectors **588**
demon
  definition **346**
  propagating selectively **346**, **352**
Depth-Bounded Discrepancy Search (DDS) **256**, **259**, **266**
Depth-First Search (DFS) **239**
development cycle **611**
direct encoding **541**
direct representation **493**
discrepancy **241**, **260**
domain **30**
  computed **334**
  current **35**
  event **337**
  interval for constrained floating-point variables **189**
  notation **31**
  reducing and floating-point expressions **201**
  reducing and `IloGenerateBounds` **201**
  reducing by elementary modifiers **332**
  stored for variables **334**
dual representation
  definition **582**
  symmetry and **582**
dynamic order **620**

E

edges **207**
element constraint **182**
`end` member function
  `IloEnv` class **46**, **304**
  `IloExpr` class **73**
  `IloSolver` class **304**
`endSearch` member function
  `IloSolver` class **61**
environment **46**
escaping local minima **426**
evaluator **498**
  node **234**, **238**
event
  domain **337**, **619**
  propagation **337**
  range **337**, **619**
  value **337**, **619**, **620**
Evolutionary Algorithms (EA) **491**, **541**, **557**
example
  cardinality **387**
  combining constrained floating-point with integer variables **196**
  configuration **381**
  Folium of Descartes **192**
  fulfilling orders **381**
  partition **388**
  Plugging Cards **177**
  Rack Configuration **177**
  set variables **381**
  tanks on trucks **381**
  union **387**
exhaustive propagation **346**
expressions **72**
  factored and canonical **197**
  operators **73**
extensible array **57**
`extract` member function
  `IloSolver` class **366**, **369**
extractable objects **297**

extraction
exceptions **299**
goals **301**
table of correspondences **302**

F

factored expressions **197**
fail **76**
`fail` member function
`IlcConstraintI` class **598**
`IlcGoalI` class **598**
`IloSolver` class **308**
failure
stepping through **597**
filter
value **377**
filter levels **215**
filtering algorithms **215**
first-fail principle **75**
fragment **474**
frequencies
assigning **209**

G

generation loop **514**
generic constraint **181**
definition **592**
example: Plugging Cards **182**
genetic algorithm cycle **567**
genetic algorithms **491**
genetic operations **504**
genetic operator **514**
operator
genetic **545**
genetic packing operator **547**
`getStatus` member function
`IloAlgorithm` class **49**
`getValue` member function
`IloSolver` class **49**
global constraint **215**
constraint
global **106**
global minimum **421**
goal **234**
calling **306**
choice points and **307**
controlling execution **306**
customizable **375**
decoding **561**
default **48**
executing **306**
labels and **307**
predefined **49**
goal stack **61**
graph coloring problem **207**
greedy improvement algorithm **411**
grouping by type
reducing symmetry **56**

H

handle
definition **340**
implementation class and **340**
heap **340**

I

IEEE 754 **203**
`ilc_fail_stop_here` function **597**
`ilc_trace_stop_here` function **597**
`IlcAnd` function **308**
`IlcChooseMinSizeInt` function
pruning search tree **588**
`IlcConstraintAggregator` **374**
`IlcConstraintI` class
`fail` member function **598**
`isViolated` virtual member function **343**, **344**
`makeOpposite` virtual member function **343**
`metaPostDemon` virtual member function **343**
`post` pure virtual member function **342**
`propagate` pure virtual member function **342**
`IlcGenerate` function
optimizing low-cost solutions **588**
`ILCGOAL` macro **311**
`IlcGoalI` class **314**
`fail` member function **598**
`IlcInstantiate` function **311**, **312**

IlcOr function **307**, **308**
IlcRevInt class **356**, **591**
IloAbs function **355**
IloAlgorithm class
getStatus member function **49**
out member function **49**
IloAllDiff class **74**, **213**, **215**, **592**
IloAndGoal function **309**
IloAny type definition **24**, **91**
IloAnyArray class **91**
IloAnyVar class **91**
IloApply function **242**, **260**
IloApplyMetaHeuristic function **424**
IloBasicLevel filter level **215**
IloBool type definition **24**, **49**
IloCard class **126**
IloCard function **387**, **388**
IloChooseFirstUnboundInt function **74**
IloChooseIntIndex type definition **74**
IloChooseMaxMaxInt function **75**
IloChooseMaxMinInt function **75**
IloChooseMaxRegretMin function **235**, **236**
IloChooseMaxSizeInt function **75**
IloChooseMinMaxInt function **75**
IloChooseMinMinInt function **75**
IloChooseMinSizeInt function **75**
IloConcatenate **440**
IloConcatenate function **434**
IloContinue **440**
IloCustomizableGoal **375**
IloDDSEvaluator function **259**
ILODEFINELNSFRAGMENT0 **474**
IloDichotomize function **619**
IloDistribute class **106**, **215**, **218**
IloDistribute function **319**, **590**
example **318**
IloEnv class **46**
end member function **46**, **304**
IloEqIntersection class **127**
IloEqMax function **381**, **386**
IloEqMin function **381**, **386**
IloEqPartition function **381**, **388**
IloEqSum function **386**, **388**
IloEqUnion function **381**, **387**
IloExpr class **72**
end member function **73**
IloExprBase class **58**
IloExtendedLevel filter level **215**
IloFailLimit function **261**
IloFlip function **419**, **455**
IloFunction function **385**
IloGenerate algorithm **74**
IloGenerate function **74**, **410**, **620**
example: Plugging Cards **183**
in solution search **74**, **108**
IloGenerate parameter choice **75**
IloGenerateBounds function **201**
IloIfThen class **144**
IloImprove function **426**, **427**
IloInitializeImpactGoal **375**
IloInstantiate function **235**, **619**
example: Plugging Cards **183**
IloInt type definition **24**
IloIntArray class **59**, **230**
IloIntArray2 class **230**
IloIntMax constant **46**
IloIntVar class **46**
IloIntVarArray class **57**, **72**
IloLargeNHoodI **474**
IloLimitSearch function **261**
IloMaximize function **93**
IloMediumLevel filter level **215**
IloMetaHeuristic class **421**
IloMetaHeuristicI class
subclassing to create new meta-heuristic **443**
IloMinimize function **279**
IloMinimizeVar function **243**
IloModel class **46**
add member function **48**, **94**
IloNHoodI class
subclassing to create new neighborhood **431**
IloNHoodModifierI class
subclassing to create new neighborhood modifier **440**
IloNodeEvaluator class **241**, **259**
IloNullIntersect function **123**
IloNum type definition **24**
IloNumSetVar class **122**
IloNumSetVarArray class **122**
IloObjective class **93**
IloOrGoal function **309**

IloRandom class **423**
IloRandomize **440**
IloRandomize function **423**, **455**
IloRandomPerturbation **545**
IloRange class **147**
IloRangeArray class **147**
IloRestoreSolution function **280**, **281**, **418**
IloSBSEvaluator function **240**, **261**
IloScalProd function **58**, **181**
IloScanDeltas function **416**, **417**
IloScanNHood function **418**, **426**
IloSearchLimitI class **365**
IloSearchSelector class **243**
IloSelectSearch function **262**
IloSingleMove function **426**, **455**
IloSolution class **280**, **281**, **410**, **412**
IloSolver class **48**, **412**
  end member function **304**
  endSearch member function **61**
  extract member function **366**, **369**
  fail member function **308**
  getValue member function **49**
  next member function **61**, **279**, **281**, **306**, **308**
  printInformation member function **76**, **465**
  setDefaultFilterLevel member function **217**
  solve member function **48**, **306**, **412**
  startNewSearch member function **60**, **281**
IloStoreSolution function **282**
IloSum function **181**, **234**
IloTabuSeach class **444**
IloTabuSearch function **426**
IloTimer class **215**
  start member function **217**
impacts **373**
  initialization **375**
implementation class
  definition **340**
  handles and **340**
  role in defining a new constraint class **340**
incremental solve **148**
indirect representation **493**, **557**
inherent constraint
  in model **616**
initial constraint propagation **32**, **33**
initial population **544**
initial population generation **497**
initial solution pool **510**
input data **230**
input from file **314**
installing Solver **23**
interval
  domain of constrained floating-point variables **189**, **196**
  notation **31**
invariant
  defining for constraint **336**
  definition **336**
  example **336**
  implementing by elementary modifiers **336**
isViolated virtual member function
  IlcConstraintI class **343**, **344**

L

Large Neighborhood Search **474**
Large Neighborhood Search (LNS) **467**
largest clique **208**
leaf **238**
least regret **236**
limit search **256**
linking library to application **23**
listener **507**, **525**
load balancing **288**
local minimum **413**
  and greedy search **420**
local move **414**
local optimization **621**
local search **409**
locating warehouses problem **227**
logical operators **145**

M

magic square problem **69**
makeOpposite virtual member function **344**
  IlcConstraintI class **343**
map coloring problem **43**
  minimizing colors **87**
mathematical programming **29**
maximal clique **208**
measuring quality of solutions **613**

memory
reclaiming **46**
memory management **46**
metaconstraint **92**, **145**
defining new class of **343**
differs from constraint **345**
metaPostDemon virtual member function
IlcConstraintI class **343**
method
incremental development **611**
searching for optimal solution **278**
three-stage **30**
minimizing
algorithm for **279**
example **280**
objectives **279**
model
adding constraints **48**
adding objects **46**
improving **584**
representation of problem **29**
synchronization **303**
modeling
as an equation **56**, **70**
modifier
domain reduction **332**
elementary **332**
for the class IlcIntExp **333**
implementing invariants **336**
volatility of **333**
monitoring **507**
move
search **35**
multiphase search **291**
multiple operator **521**
mutation **545**

N

naming
accessors **24**
arguments **23**
Boolean accessors **24**
classes **23**
conventions **23**
data members **24**
instances **23**
member functions **23**
modifiers **24**
neighborhood **416**
creating new by subclassing IloNHoodI **431**
defining **418**
large **467**, **474**
random **423**
neighborhood modifier
creating new by subclassing IloNHoodModifierI **440**
next member function
IloSolver class **61**, **279**, **281**, **306**, **308**
node **207**, **238**
child **239**
closed **238**
open **238**
parent **239**
unexplored **238**
node evaluator **234**, **238**
notation
domain **31**
interval **31**
range **31**
set **31**
square brackets **31**

O

objective **30**, **88**
maximize **92**
minimize **92**
objects **90**
extractable **297**
offspring **514**
one max problem **509**
open node **238**
operator
combine constraints **190**
genetic packing **547**
multiple **521**
operator factory **565**
operators **47**
combine constraints **47**

expressions **73**
genetic **504**
logical **145**
optimizing
guidelines for **618**, **621**
locally **621**
propagation **619**
solution search **278**, **619**
order
constraints and **346**
dynamic **620**
example **183**
imposing on variables **584**
imposing to improve model **584**
permuting variables and **584**
sets and **584**
variables and choice function **358**
variables and `IloGenerate` **620**
`out` member function
`IloAlgorithm` class **49**
output
Building Warehouses **465**
example: Configuring Tankers **403**
example: Frequency Allocation **364**
example: Plugging Cards **187**
example: Set Variables **403**
example:Polynomial Roots **195**

P

parent **503**
parent node **239**
partial replacement **533**
partitioning
example **388**
personal computer (pc)
linking Solver library on **23**
pool
solution **492**, **496**, **510**
pool evaluator
evaluator
pool **533**
pool processor **492**, **501**, **547**
population **492**, **497**
`post` pure virtual member function **337**
`IlcConstraintI` class **342**
predefined constraint **74**, **106**, **123**, **144**
predefined goal **49**
print solution **49**
`printInformation` member function
`IloSolver` class **76**
problem decomposition **256**
problem description
example: Configuring Tankers **381**
example: Set Variables **381**
guidelines **612**
problem representation
example: Allocating Frequencies **355**
example: Configuring Tankers **382**
example: Set Variables **382**
guidelines **614**
problem solution
example: Allocating Frequencies **356**
example: Car Sequencing **322**
example: Rack Configuration **183**
processor
pool **492**, **501**, **547**
programming
constraint **29**
mathematical **29**
`propagate` pure virtual member function **337**, **339**
`IlcConstraintI` class **342**
propagation
algorithm **337**
back **585**
by means of demons **346**
constraint **30**, **215**
during search **32**, **35**
initial **32**, **33**
constraints, stepping **597**
events **337**
exhaustive **346**
see bound communication
tracing **596**
pruning **75**, **239**

Q

qualifying acceptable solutions **613**

R

range
  event **337**
  notation **31**
range constraints **147**
reduce symmetry
  introduce order among variables **151**
reducing
  domains and `IloGenerateBounds` **201**
  domains by elementary modifiers **332**
  domains of floating-point expressions **201**
redundant constraint **183**, **590**
  definition **590**
  eliminating symmetry **183**
  example: Plugging Cards **183**
  improving performance **591**
regret
  least **236**
relaxing constraints **88**
restart **378**
restarting solution search **279**, **280**
reversible data member **591**
right move **241**
root **238**
  search tree **34**
roulette wheel selector **547**

S

search
  algorithms **29**
  limit **256**
  local **409**
  multiphase **291**
  using impacts **373**
search criterion
  parameters: `IloChooseMaxMaxInt` **75**
  parameters: `IloChooseMaxMinInt` **75**
  parameters: `IloChooseMaxSizeInt` **75**
  parameters: `IloChooseMinMaxInt` **75**
  parameters: `IloChooseMinMinInt` **75**
  parameters: `IloChooseMinSizeInt` **75**
search limits **365**
search move **35**
search procedures **266**
search selector **234**, **243**
search space **33**
  reducing size of **581**
search strategies **30**, **35**, **234**, **240**
search strategy **374**
  example **356**
search tree **34**, **238**
  binarization **238**
  branch **34**
  root **34**
selector **378**, **498**, **514**
  defining **588**
  roulette wheel **547**
  search **234**, **243**
  tournament **529**
set
  cardinality **584**
  constrained variables **122**
  notation **31**
  order in **584**
set variable **584**
  example **381**
`setDefaultFilterLevel` member function
  `IloSolver` class **217**
setting
  choice points **307**
slack **320**
Slice-Based Search (SBS) **240**
solution **496**
  current **410**
  improving with local search **410**
  print **49**
  storing intermediate **280**
  testing **592**
solution delta **415**
  testing **417**
solution pool **492**, **496**, **510**
solution prototype **510**, **544**, **561**
solution replacement **506**, **570**
solution search
  controlling a choice **74**
  optimal solutions **278**
  restarting **279**, **280**
solve

incremental **148**
solve member function
IloSolver class **48**, **306**, **412**
solve process **32**
summary **38**
solver heap
allocating memory on **304**
stack
goal **61**
start member function
IloTimer class **217**
startNewSearch member function
IloSolver class **60**, **281**
strategies
search **30**, **35**, **240**
strongly coupled problem **495**
structural constraint
in model **616**
support
customer **24**
symbolic constraint
guidelines **617**
symmetry
dual representation and **582**
guidelines **616**
reducing **56**, **151**, **180**
removing **183**, **583**
synchronization **291**
synchronization of the model **303**

T

talent scheduling problem **468**
termination detection **288**
test not a constraint **279**
three-stage method **30**
describe **30**
model **31**
solve **32**
tournament selector **529**
trace hook
definition **599**
using **599**
tracing
choice points **599**
propagation **596**
tree traversal **239**
type
grouping by **56**
types in Solver **24**

U

union of sets **387**
UNIX
linking Solver library on **23**

V

value event **337**
value filter **377**
values **30**
variable
permuting order of **584**
set **381**, **584**
variable filter
filter
variable **375**
variables
constrained **31**
constrained integer **46**
array **57**
constrained set **122**
decision **30**, **31**, **47**

W

worker **287**
workers **288**