

Finite Domain Constraint Programming in Oz. A Tutorial.

**Christian Schulte
Gert Smolka**

**Version 1.2.3
December 1, 2001**



Abstract

This document is an introduction to constraint programming in Oz. We restrict our attention to combinatorial problems that can be stated with variables ranging over finite sets of nonnegative integers. Problems in this class range from puzzles to real world applications as diverse as scheduling, ware house allocation, configuration and placement.

Credits

Mozart logo by Christian Lindig

License Agreement

This software and its documentation are copyrighted by the German Research Center for Artificial Intelligence (DFKI), the Swedish Institute of Computer Science (SICS), and other parties. The following terms apply to all files associated with the software unless explicitly disclaimed in individual files.

The authors hereby grant permission to use, copy, modify, distribute, and license this software and its documentation for any purpose, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions. No written agreement, license, or royalty fee is required for any of the authorized uses. Modifications to this software may be copyrighted by their authors and need not follow the licensing terms described here, provided that the new terms are clearly indicated on the first page of each file where they apply.

IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS SOFTWARE AND ITS DOCUMENTATION ARE PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 1 |
| 2 | Propagate and Distribute | 5 |
| 2.1 | Finite Domains and Constraints | 5 |
| 2.2 | Constraint Propagation | 6 |
| 2.3 | Spaces, Propagators, and Constraint Stores | 6 |
| 2.4 | Interval and Domain Propagation | 8 |
| 2.5 | Incompleteness of Propagation | 9 |
| 2.6 | Distribution and Search Trees | 9 |
| 2.7 | An Example | 10 |
| 2.8 | Distribution Strategies | 12 |
| 2.9 | Search Order | 13 |
| 2.10 | Models | 13 |
| 3 | Writing Problem Solvers in Oz | 15 |
| 3.1 | Format of Scripts | 15 |
| 3.2 | Example: Send More Money | 16 |
| 3.3 | The Explorer | 18 |
| 3.4 | New Primitives | 21 |
| 3.5 | Watching Propagators | 22 |
| 3.6 | Example: Safe | 23 |
| 4 | Elimination of Symmetries and Defined Constraints | 25 |
| 4.1 | Example: Grocery | 25 |
| 4.2 | Example: Family | 27 |
| 4.3 | Example: Zebra Puzzle | 29 |
| 5 | Parameterized Scripts | 33 |
| 5.1 | Example: Queens | 33 |
| 5.2 | Example: Changing Money | 35 |

| | | |
|-----------|---|-----------|
| 6 | Minimizing a Cost Function | 37 |
| 6.1 | Example: Coloring a Map | 37 |
| 6.2 | Example: Conference | 39 |
| 7 | Propagators for Redundant Constraints | 43 |
| 7.1 | Example: Fractions | 43 |
| 7.2 | Example: Pythagoras | 44 |
| 7.3 | Example: Magic Squares | 47 |
| 7.4 | Exercises | 49 |
| 8 | Reified Constraints | 51 |
| 8.1 | Getting Started | 51 |
| 8.2 | Example: Aligning for a Photo | 53 |
| 8.3 | Example: Self-referential Aptitude Test | 55 |
| 8.4 | Example: Bin Packing | 59 |
| 9 | User-Defined Distributors | 65 |
| 9.1 | A Naive Distribution Strategy | 65 |
| 9.2 | A Domain-Splitting Distributor | 66 |
| 10 | Branch and Bound | 69 |
| 10.1 | Example: Aligning for a Photo, Revisited | 69 |
| 10.2 | Example: Locating Warehouses | 70 |
| 11 | Scheduling | 75 |
| 11.1 | Building a House | 75 |
| 11.1.1 | Building a House: Precedence Constraints | 76 |
| 11.1.2 | Building a House: Capacity Constraints | 78 |
| 11.1.3 | Building a House: Serializers | 80 |
| 11.2 | Constructing a Bridge | 82 |
| 11.3 | Strong Propagators for Capacity Constraints | 88 |
| 11.4 | Strong Serializers | 91 |
| 11.5 | Solving Hard Scheduling Problems | 93 |
| 11.5.1 | The Problem ABZ6 | 94 |
| 11.5.2 | The MT10 Problem | 96 |
| A | Traps and Pitfalls | 97 |

| | | |
|----------|-----------------------------|------------|
| B | Golden Rules | 101 |
| C | Example Data | 103 |
| D | Answers To Exercises | 107 |

Introduction

This document is a first introduction to constraint-based problem solving and its implementation in Oz. We restrict our attention to combinatorial problems that can be stated with variables ranging over finite sets of nonnegative integers. Problems in this class range from puzzles to real world applications as diverse as scheduling, warehouse allocation, configuration and placement.

The two basic techniques of constraint programming are constraint propagation and constraint distribution. Constraint propagation is an efficient inference mechanism obtained with concurrent propagators accumulating information in a constraint store. Constraint distribution splits a problem into complementary cases once constraint propagation cannot advance further. By iterating propagation and distribution, propagation will eventually determine the solutions of a problem.

Constraint distribution can easily lead to an exponential growth of the number of subproblems to be considered. Fortunately, this potential combinatorial explosion can often be contained by combining strong propagation mechanisms with problem specific heuristics for selecting distribution steps.

The following puzzles give a first idea of the combinatorial problems we will solve in this document:

Money

The Send More Money Problem consists in finding distinct digits for the letters D, E, M, N, O, R, S, Y such that S and M are different from zero (no leading zeros) and the equation

$$SEND + MORE = MONEY$$

is satisfied. The unique solution of the problem is $9567 + 1085 = 10652$.

Safe

The code of Professor Smart’s safe is a sequence of 9 distinct nonzero digits C_1, \dots, C_9 such that the following equations and inequations are satisfied:

$$\begin{aligned} C_4 - C_6 &= C_7 \\ C_1 * C_2 * C_3 &= C_8 + C_9 \\ C_2 + C_3 + C_6 &< C_8 \\ C_9 &< C_8 \\ C_1 \neq 1, \dots, C_9 &\neq 9 \end{aligned}$$

Can you determine the code?

Coloring

Given a map showing the West European countries Netherlands, Belgium, France, Spain, Portugal, Germany, Luxemburg, Switzerland, Austria, and Italy, find a coloring such that neighboring countries have different color and a minimal number of colors is used.

Grocery

A kid goes into a grocery store and buys four items. The cashier charges \$7.11, the kid pays and is about to leave when the cashier calls the kid back, and says ‘Hold on, I multiplied the four items instead of adding them; I’ll try again; Hah, with adding them the price still comes to \$7.11’. What were the prices of the four items?

Queens

Place 8 queens on a chess board such that no two queens attack each other.

The problems have in common that they can be stated with variables that are a priori constrained to finite sets of nonnegative integers. Consequently, the problems could be solved by simply checking all possible combinations of the values of the variables. This naive generate and test method is however infeasible for most realistic problems since there are just too many possible combinations.

More Information

While this tutorial tries to be as self-contained as possible for constraint programming in Oz, it is expected that the reader has already a working knowledge of functional Oz programming. As an introduction for functional and object-oriented programming in Oz we suggest “*Tutorial of Oz*”. The full functionality of Oz provided for constraint programming is included in the document *Part Constraint Programming, (System Modules)*.

The Examples

The tutorial features a large number of examples. To ease the understanding the examples should be tried in the Oz Programming Environment (OPI)¹. The Oz programs are contained in the following file². Oz programs for some solutions to the exercises are contained in the following file³.

Acknowledgements

The tutorial is based on the document “Finite Domain Constraint Programming in Oz. A Tutorial” by Gert Smolka, Christian Schulte, and Jörg Würtz for a previous version of Oz.

¹*“The Oz Programming Interface”*

²`FiniteDomainTutorial.oz`

³`FiniteDomainTutorialSolutions.oz`

Propagate and Distribute

This section presents the architecture of constraint-based problem solving at the concrete instance of finite domain problems. We will often refer to the underlying solution method with the slogan ‘propagate and distribute’. The slogan recalls the two inference rules of the method, constraint propagation and constraint distribution.

2.1 Finite Domains and Constraints

A finite domain is a finite set of nonnegative integers. The notation $m\#n$ stands for the finite domain m, \dots, n .

A constraint is a formula of predicate logic. Here are typical examples of constraints occurring with finite domain problems:

$$\begin{array}{l} X = 67 \quad X \in 0\#9 \quad X = Y \\ X^2 - Y^2 = Z^2 \quad X + Y + Z < U \quad X + Y \neq 5 \cdot Z \\ X_1, \dots, X_9 \text{ are pairwise distinct} \end{array}$$

domain constraints A domain constraint takes the form $x \in D$, where D is a finite domain. Domain constraints can express constraints of the form $x = n$ since $x = n$ is equivalent to $x \in n\#n$.

basic constraints A basic constraint takes one of the following forms: $x = n$, $x = y$, or $x \in D$, where D is a finite domain.

finite domain problems A finite domain problem is a finite set P of quantifier-free constraints such that P contains a domain constraint for every variable occurring in a constraint of P . A variable assignment is a function mapping variables to integers.

solutions A solution of a finite domain problem P is a variable assignment that satisfies every constraint in P .

Notice that a finite domain problem has at most finitely many solutions, provided we consider only variables that occur in the problem (since the problem contains a finite domain constraint for every variable occurring in it).

2.2 Constraint Propagation

Constraint propagation is an inference rule for finite domain problems that narrows the domains of variables. For instance, given the inequation

$$X < Y$$

and the domain constraints

$$X \in 23\#100$$

and

$$Y \in 1\#33$$

constraint propagation can narrow the domains of X and Y to

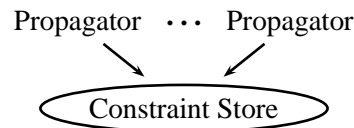
$$X \in 23\#32$$

and

$$Y \in 24\#33$$

2.3 Spaces, Propagators, and Constraint Stores

The computational architecture for constraint propagation is called a space and consists of a number of propagators connected to a constraint store:



The constraint store stores a conjunction of basic constraints up to logical equivalence. An example for such a conjunction is

$$X \in 0\#5 \wedge Y = 8 \wedge Z \in 13\#23.$$

The propagators impose nonbasic constraints, for instance, $X < Y$ or $X^2 + Y^2 = Z^2$. A propagator for a constraint C is a concurrent computational agent that tries to narrow the domains of the variables occurring in C .

example of communicating propagators Two propagators that share a variable X can communicate with each other through the constraint store. To see an example for communicating propagators, suppose we have two propagators imposing the constraints

$$X + Y = 9 \quad 2 \cdot X + 4 \cdot Y = 24$$

over a constraint store containing the following information about X and Y :

$$X \in 0\#9 \quad Y \in 0\#9$$

As is, the first propagator cannot do anything. The second propagator, however, can narrow the domains of both X and Y :

$$X \in 0\#8 \quad Y \in 2\#6$$

Now the first propagator can narrow the domain of X :

$$X \in 3\#7 \quad Y \in 2\#6$$

Now the second propagator can narrow the domains of both X and Y :

$$X \in 4\#6 \quad Y \in 3\#4$$

This once more activates the first propagator, which narrows the domain of X :

$$X \in 5\#6 \quad Y \in 3\#4$$

Now the second propagator gets active once more and determines the values of X and Y :

$$X = 6 \quad Y = 3$$

telling a constraint Given a constraint store storing a constraint S and a propagator imposing a constraint P , the propagator can tell to the constraint store a basic constraint B if the conjunction $S \wedge P$ entails B and B adds new and consistent information to the store. To tell a constraint B to a constraint store storing the constraint S means to update the store so that it holds the conjunction $S \wedge B$.

operational and declarative semantics of propagators The operational semantics of a propagator determines whether the propagator can tell the store a basic constraint or not. The operational semantics of a propagator must of course respect the declarative semantics of the propagator, which is given by the constraint the propagator imposes.

We require that the constraint store be always consistent; that is, there must always be at least one variable assignment that satisfies all constraints in the constraint store.

determined variables We say that the constraint store determines a variable x if the constraint store knows the value of x , that is, if there exists an integer n such that the constraint store entails $x = n$.

failed propagators We say that a propagator is inconsistent if there is no variable assignment that satisfies both the constraint store and the constraint imposed by the propagator (e.g., $X + Y = 6$ over $X \in 3\#9$ and $Y \in 4\#9$). We say that a propagator is failed if its operational semantics realizes that it is inconsistent.

entailed propagators We say that a propagator is entailed if every variable assignment that satisfies the constraint store also satisfies the constraint imposed by the propagator (e.g., $X < Y$ over $X \in 3\#5$ and $Y \in 6\#9$). As soon as the operational semantics of a propagator realizes that the propagator is entailed, the propagator ceases to exist.

We require that the operational semantics of a propagator detects inconsistency and entailment at the latest when the store determines all variables of the propagator.

stable propagators We say that a propagator is stable if it is either failed or its operational semantics cannot tell new information to the constraint store.

failed, stable, and solved spaces We say that a space is failed if one of its propagators is failed. We say that a space is stable if all of its propagators are stable. We say that a space is solved if it is not failed and there are no propagators left.

propagation order does not matter When a space is created, its propagators start to tell basic constraints to the constraint store. This propagation activity continues until the space becomes stable. An important property of constraint propagation as we consider it here is the fact that the order in which the propagators tell information to the store does not matter. When we start a space repeatedly from the same state, it will either always fail or always arrive at the same constraint store (up to logical equivalence).

solutions of a space A variable assignment is called a solution of a space if it satisfies the constraints in the constraint store and all constraints imposed by the propagators. The solutions of a space stay invariant under constraint propagation and deletion of entailed propagators.

2.4 Interval and Domain Propagation

There are two established schemes for the operational semantics of a propagator. Domain propagation narrows the domains of the variables as much as possible; interval propagation only narrows the bounds of a domain.

Consider a propagator for the constraint

$$2 \cdot X = Y$$

over a constraint store

$$X \in 1\#10 \quad Y \in 1\#7$$

Under domain propagation, the propagator can narrow the domains to

$$X \in 1\#3 \quad Y \in \{2, 4, 6\}$$

Under interval propagation, the propagator can narrow only the domain bounds, which yields

$$X \in 1\#3 \quad Y \in 2\#6$$

In practice, interval propagation is usually preferable over domain propagation because of its lower computational costs. We will see later that Oz offers for some constraints two propagators, one for interval and one for domain propagation.

2.5 Incompleteness of Propagation

Constraint propagation is not a complete solution method. It may happen that a space has a unique solution and that constraint propagation does not find it. It may also happen that a space has no solution and that constraint propagation does not lead to a failed propagator.

A straightforward example for the second case consists of three propagators for

$$X \neq Y \quad X \neq Z \quad Y \neq Z$$

and a constraint store

$$X \in 0\#1 \quad Y \in 0\#1 \quad Z \in 0\#1.$$

This space has no solution. Nevertheless, none of the propagators is inconsistent or can tell something to the constraint store.

To see an example for the case where a unique solution is not found by constraint propagation, suppose we have interval propagators for the constraints

$$3 \cdot X + 3 \cdot Y = 5 \cdot Z \quad X - Y = Z \quad X + Y = Z + 2$$

and a constraint store

$$X \in 4\#10 \quad Y \in 1\#7 \quad Z \in 3\#9$$

This space has the unique solution $X = 4$, $Y = 1$, $Z = 3$. Nevertheless, none of the propagators can narrow a variable domain.

If we narrow the domains to

$$X \in 5\#10 \quad Y \in 1\#6 \quad Z \in 4\#9$$

the space becomes unsatisfiable. Still, none of the above propagators is inconsistent or can narrow a variable domain.

2.6 Distribution and Search Trees

To solve a finite domain problem P , we can always choose a constraint C and solve both $P \cup \{C\}$ and $P \cup \{\neg C\}$. We say that we have distributed P with C .

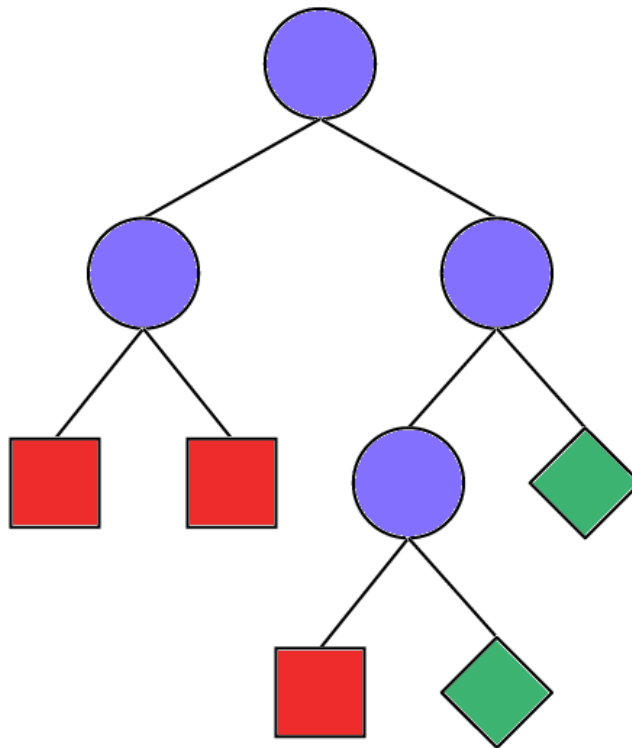
We can apply the idea to spaces. Suppose S is a stable space that is neither failed nor solved. Then we can choose a constraint C and distribute S with C . Distribution yields two spaces, one obtained by adding a propagator for C , and one obtained by adding a propagator for $\neg C$.

The combination of constraint propagation and distribution yields a complete solution method for finite domain problems. Given a problem, we set up a space whose store

contains the basic constraints and whose propagators impose the nonbasic constraints of the problem. Then we run the propagators until the space becomes stable. If the space is failed or solved, we are done. Otherwise, we choose a not yet determined variable x and an integer n such that $x = n$ is consistent with the constraint store and distribute the space with the constraint $x = n$. Since we can tell both $x = n$ and $x \neq n$ to the constraint store (the store already knows a domain for x), chances are that constraint propagation can restart in both spaces.

By proceeding this way we obtain a search tree as shown in Figure 2.1. Each node of the tree corresponds to a space. Each leaf of the tree corresponds to a space that is either solved or failed. The search tree is always finite since there are only finitely many variables all a priori constrained to finite domains.

Figure 2.1: A search tree. Diamonds represent solved spaces and boxes represent failed spaces.



2.7 An Example

To see the outlined propagate and distribute method at a concrete example, consider the problem specified by the following constraints:

$$\begin{array}{lll} X \neq 7 & Z \neq 2 & X - Z = 3 \cdot Y \\ X \in 1\#8 & Y \in 1\#10 & Z \in 1\#10 \end{array}$$

To solve the problem, we start with a space whose store constrains the variables x , Y , and Z to the given domains. We also create three propagators imposing the constraints

$X \neq 7$, $Z \neq 2$, and $X - Z = 3 \cdot Y$. We assume that the propagator for $X - Z = 3 \cdot Y$ realizes interval propagation, and that the propagators for the disequations $X \neq 7$ and $Z \neq 2$ realize domain propagation.

The propagators for the disequations immediately write all their information into the store and disappear. The store then knows the domains

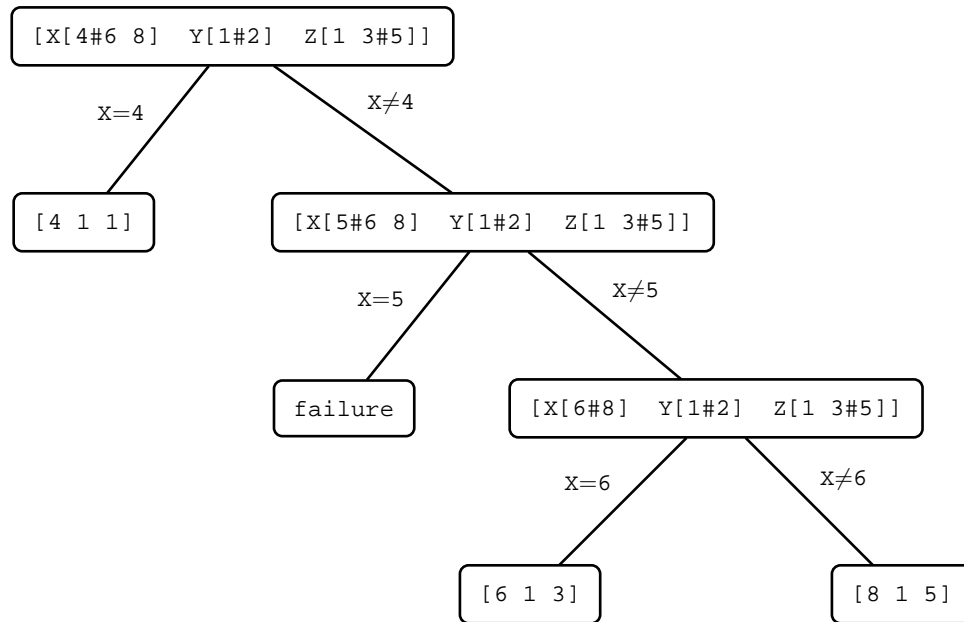
$$X \in [1\#6\ 8] \quad Y \in 1\#10 \quad Z \in [1\ 3\#10]$$

where $[1\ 3\#10]$ denotes the finite domain $\{1\} \cup \{3, \dots, 10\}$. The interval propagator for $X - Z = 3 \cdot Y$ can now further narrow the domains to

$$X \in [4\#6\ 8] \quad Y \in 1\#2 \quad Z \in [1\ 3\#5].$$

Now the space is stable but neither failed nor solved. Thus, we continue with a first distribution step. We choose to distribute with the constraint $X = 4$. Figure 2.2 shows the resulting search tree.

Figure 2.2: A search tree containing 3 choice nodes, 1 failure node, and 3 solution nodes.



The space obtained by adding a propagator for $X = 4$ can be solved by propagation and yields the solution

$$X = 4 \quad Y = 1 \quad Z = 1$$

The space obtained by adding a propagator for $X \neq 4$ becomes stable immediately after this propagator has written its information into the constraint store, which then looks as follows:

$$X \in [5\#6\ 8] \quad Y \in 1\#2 \quad Z \in [1\ 3\#5]$$

This time we distribute with respect to the constraint $X = 5$.

The space obtained by adding a propagator for $X = 5$ fails since $X - Z = 3 \cdot Y$ is inconsistent with the store obtained by adding $X = 5$.

The space obtained by adding a propagator for $X \neq 5$ becomes stable immediately after this propagator has written its information into the constraint store, which then looks as follows:

$$X \in [6 \ 8] \quad Y \in 1\#2 \quad Z \in [1 \ 3\#5]$$

Now we distribute with respect to the constraint $X = 6$.

The space obtained by adding a propagator for $X = 6$ can be solved by propagation and yields the solution

$$X = 6 \quad Y = 1 \quad Z = 3$$

Finally, the space obtained by adding a propagator for $X \neq 6$ can also be solved by propagation, yielding the third and final solution

$$X = 8 \quad Y = 1 \quad Z = 5$$

An alternative to the propagate and distribute method is a naive enumerate and test method, which would enumerate all triples (X, Y, Z) admitted by the initial domain constraints and test the constraints $X \neq 7$, $Z \neq 2$, and $X - Z = 3 \cdot Y$ for each triple. There are $8 * 10 * 10 = 800$ candidates. This shows that constraint propagation can reduce the size of the search tree considerably.

2.8 Distribution Strategies

A distributor is a computational agent implementing a distribution strategy. If a thread creates a distributor, the thread is blocked until the distributor has done its job. If a distribution step is needed, the distributor becomes active and generates the constraint with which the space will be distributed. If there is more than one distributor in existence, one of them is chosen indeterministically whenever a distribution step is needed.

Usually, a distribution strategy is defined on a sequence x_1, \dots, x_n of variables. When a distribution step is necessary, the strategy selects a not yet determined variable in the sequence and distributes on this variable.

standard possibilities to distribute on a variable There are a few standard possibilities to distribute on a variable x :

- distribute with $x = l$, where l is the least possible value for x .
- distribute with $x = u$, where u is the largest possible value for x .
- distribute with $x = m$, where m is a possible value for x that is in the middle of the least and largest possible value for x .
- distribute with $x \leq m$, where m is a possible value for x that is in the middle of the least and largest possible value for x (so called domain splitting).

naive distribution A naive distribution strategy will select the leftmost undetermined variable in the sequence.

first-fail distribution A first-fail distribution strategy will select the leftmost undetermined variable in the sequence whose domain in the constraint store has minimal size. In other words, it will select the leftmost undetermined variable for which the number of different possible values is minimal.

For most problems, first-fail strategies yield much smaller search trees than naive strategies.

2.9 Search Order

So far we have not specified in which order search trees are explored. Although this order has no impact on the shape and size of a search tree, it does have an impact on the time and memory resources needed to find one or all solutions:

- If we are only interested in one solution, there is no need to explore the entire search tree. Ideally, we would just explore the path leading from the root to the solution.
- If we are interested in all solutions, we need to explore the entire search tree. However, whether we explore the tree in depth-first or breadth-first manner will make a big difference in the memory needed. The memory requirements of breadth-first exploration are typically exponentially larger than those of depth-first exploration.

We will assume that the search engine explores the search trees always in a depth-first fashion. Moreover, when the engine distributes with a constraint C , it explores the space obtained with C first and the space obtained with $\neg C$ second.

The above assumptions ensure that the exploration of a search tree is a deterministic process, provided the distribution strategy generating the constraints to distribute with is deterministic.

2.10 Models

A model of a problem is a representation of the problem as a finite domain problem (as defined in Section 2.1). A model specifies the variables and the constraints representing the problem.

Nontrivial problems will admit different models and different distribution strategies, coming with different computational properties and search trees of different size. The art of constraint programming consists in finding for a problem a model and a distribution strategy that yield a computationally feasible search tree.

Writing Problem Solvers in Oz

We are now well-prepared to write and run our first finite domain problem solvers in Oz. For running, analyzing, and debugging problem solvers we will use the Explorer¹, a graphical tool of the Mozart programming environment.

A script for a finite domain problem is a program that can compute one or all solutions of the problem. In Oz, scripts will be run on predefined search engines implementing the propagate and distribute method just described. Separating scripts from the search engines running them is an important abstraction making it possible to design scripts at a very high level. To develop a script for a given problem, we start by designing a model and a distribution strategy. We then obtain an executable script by implementing the model and distribution strategy with the predefined abstractions available in Oz.

3.1 Format of Scripts

In Oz, a script takes the form of a procedure

```
proc {Script Root}
  %% declare variables
in
  %% post constraints
  %% specify distribution strategy
end
```

The procedure declares the variables needed, posts the constraints modeling the problem, and specifies the distribution strategy.

The argument `Root` stands for the solutions of the problem to be solved. If the solutions of a problem are given by more than one variable, say `x`, `y`, and `z`, we may simply combine these variables into one record by posting a constraint like

```
Root = solution(x:X y:Y z:Z)
```

The procedure

```
{SearchAll Script ?Solutions}
```

¹“Oz Explorer – Visual Constraint Programming Support”

will run the script `Script` until the entire search tree is explored and return the list of the solutions found.

The procedure

```
{SearchOne Script ?Solutions}
```

will run the script `Script` until the first solution is found. If a solution is found, it is returned as the single element of a list; otherwise, the empty list is returned.

3.2 Example: Send More Money

We will now write a script for the Send More Money Puzzle.

Problem Specification

The Send More Money Problem consists in finding distinct digits for the letters *D*, *E*, *M*, *N*, *O*, *R*, *S*, *Y* such that *S* and *M* are different from zero (no leading zeros) and the equation

$$SEND + MORE = MONEY$$

is satisfied. The unique solution of the problem is $9567 + 1085 = 10652$.

Model

We model the problem by having a variable for every letter, where the variable stands for the digit associated with the letter. The constraints are obvious from the problem specification.

Distribution Strategy

We distribute on the variables for the letters with a first-fail strategy. The variables are ordered according to the alphabetic order of the letters. The strategy tries the least possible value of the selected variable.

Script

Figure 3.1 shows a script realizing the model and distribution strategy just discussed. The script first declares a local variable for every letter. Then it posts the following constraints:

1. `Root` is a record that has a field for every letter. The fields of `Root` are the digits for the corresponding letters. This constraint is basic.
2. The fields of `Root` are integers in the domain `0#9`. This constraint is basic.
3. The fields of `Root` are pairwise distinct. This constraint is nonbasic.
4. The values of the variables `S` and `M` are different from zero (no leading zeros). These constraints are nonbasic.
5. The digits for the letters satisfy the equation `SEND+MORE=MONEY`. This constraint is nonbasic.

Figure 3.1: A script for the Send More Money Puzzle.

```

proc {Money Root}
  S E N D M O R Y
in
  Root = sol(s:S e:E n:N d:D m:M o:O r:R y:Y)           % 1
  Root ::: 0#9                                           % 2
  {FD.distinct Root}                                     % 3
  S \=: 0                                                % 4
  M \=: 0
  1000*S + 100*E + 10*N + D                             % 5
+ 1000*M + 100*O + 10*R + E
=: 10000*M + 1000*O + 100*N + 10*E + Y
  {FD.distribute ff Root}
end

```

Posting of constraints

posting of constraints is defined differently for basic and nonbasic constraints. Basic constraints are posted by telling them to the constraint store. Nonbasic constraints are posted by creating propagators implementing them.

Note that the propagators for `S\=:0` and `M\=:0` can immediately write their complete information into the constraint store since the store already knows domains for `S` and `M`.

The last line `{FD.distribute ff Root}` posts a distributor that will distribute on the field of `Root` with the first-fail strategy (specified by the atom `ff`). Equivalently, we could write

```
{FD.distribute ff [D E M N O R S Y]}
```

and thus specify the variables and their order explicitly. The order of the fields of `Root` is given by the canonical ordering of the respective features `d`, `e`, `m`, `n`, `o`, `r`, `s`, and `y`.

The statement

```
{Browse {SearchAll Money}}
```

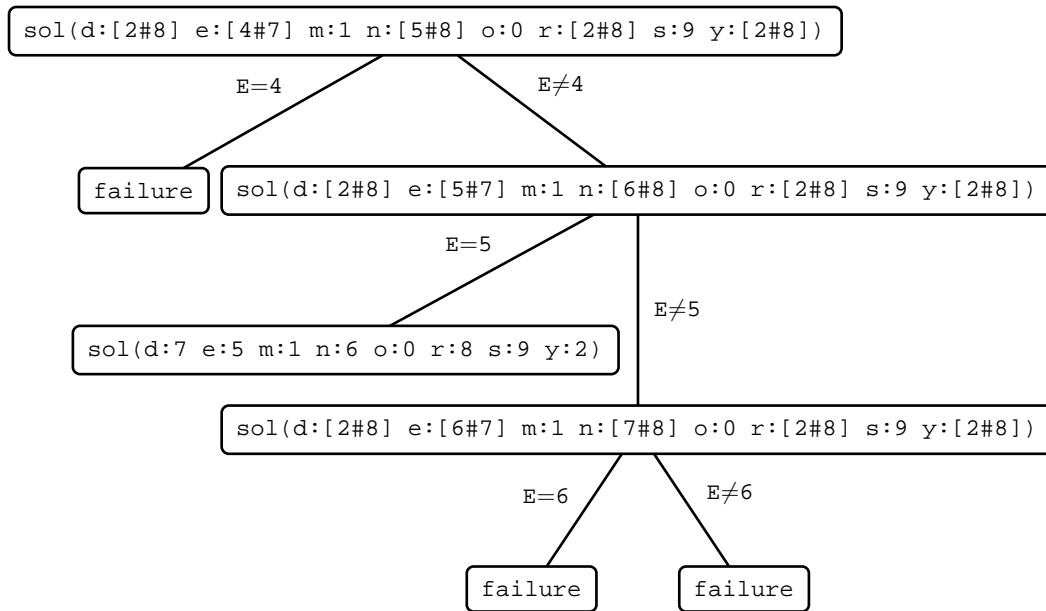
will compute and display the list of all solutions of the Send More Money Puzzle:

```
[sol(d:7 e:5 m:1 n:6 o:0 r:8 s:9 y:2)]
```

To understand the search process defined by `Money`, we need more information than just the list of solutions found. Obviously, it would be useful to see a graphical representation of the search tree. It would also be nice if we could see for every node of the search tree what information about the solution was accumulated in the constraint store when the respective space became stable. Finally, it would be nice to see for every arc of the tree with which constraint it was distributed.

Figure 3.2 shows the search tree explored by `Money` together with the information just mentioned. This gives us a good understanding of the search process defined by `Money`. Note that the search tree is quite small compared to the 10^8 candidates a naive generate and test method would have to consider.

Figure 3.2: The search tree explored by `Money`.



3.3 The Explorer

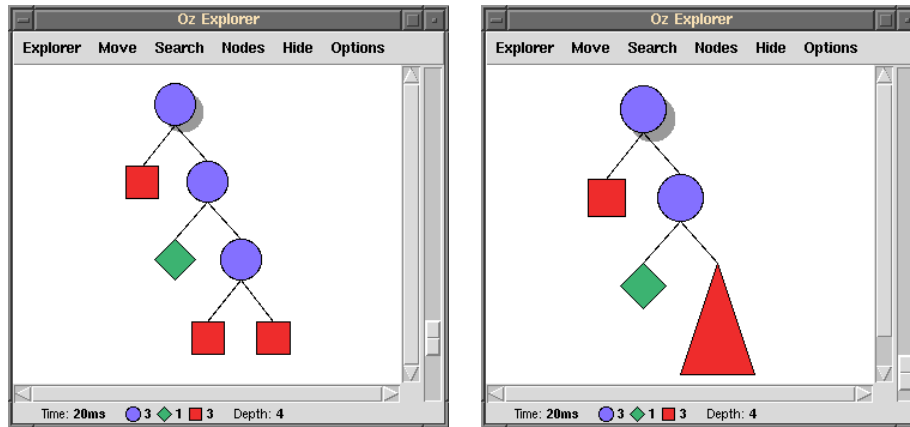
The Explorer is a graphical tool of the Mozart programming environment. It can run scripts and display the explored search trees. It can also display the information in the constraint stores associated with the nodes of the search tree.

The statement

```
{ExploreAll Money}
```

tells the Explorer to run the script `Money` and explore the entire search tree. The Explorer will pop up a window and display the explored nodes of the search tree (see left part of Figure 3.3). Choice nodes appear as blue circles, failure nodes as red boxes, and solution nodes as green diamonds. Fully explored subtrees not containing solution nodes are collapsed into a single red triangle.

You can select any node of the displayed search tree by clicking it with the left mouse button. Select the red triangle and type the command `h` (hide/unhide). This will replace the triangle with the actual nodes of the tree (see right part of Figure 3.3). You now see the full search tree of `Money`, which consists of three choice nodes, three failure nodes, and one solution node. Typing the command `h` once more will switch back to the compact representation of the failed subtree.

Figure 3.3: The Explorer with the search tree of `Money`.

double clicking nodes Next, double click the green solution node with the left mouse button.

This will display the unique solution

```
sol(d:7 e:5 m:1 n:6 o:0 r:8 s:9 y:2)
```

of the Money Puzzle in the Browser. You can also double click a blue choice node. This will display the information about the solution that was accumulated in the constraint store before the node was distributed. Double clicking the top node of the tree, for instance, will display

```
sol(d:_[2#8] e:_[4#7] m:1 n:_[5#8]
    o:0 r:_[2#8] s:9 y:_[2#8])
```

in the Browser. This way, the Explorer and the Browser can display the annotated search tree shown in Figure 3.2.

open and closed choice nodes The statement

```
{ExploreOne Money}
```

tells the Explorer to run the script `Money` until the first solution is found. This time the Explorer will show a partial search tree that contains the solution node in the rightmost position, and also contains an open choice node. An open choice node is a choice node for which not all direct descendents have been explored yet. A closed choice node is a choice node for which all direct descendents have been explored already. While closed choice nodes are displayed in dark blue, open choice nodes are displayed in light blue. Not yet explored descendents of an open choice node are not displayed.

To check whether there are further solutions, you can resume the search process by selecting the root node and typing the command `n` (next). This will resume the search until either the next solution is found or all nodes of the search tree are explored.

stopping exploration You can stop the exploring Explorer at any time by typing the command `C-g`.

resuming exploration You can resume the exploration of a partial search tree by selecting any choice node and typing the command `n` or `a`. The command `n` (next) will resume the exploration of the selected subtree until a further solution is found or the subtree is fully explored. The command `a` (all) will resume the exploration of the selected subtree until it is fully explored.

resetting the Explorer The command `C-r` will reset the Explorer and show only the root node of the search tree. By double clicking you can see in the Browser what is known about the solution before the first distribution step. You can request the exploration of the search tree by typing `n` or `a`.

hand-guided exploration You can guide the search of the Explorer by hand. Reset the Explorer by typing `C-r`. This will select the root node, which is an open choice node. Now type the command `o` (one) to compute the first descendent of the root. Select the root once more and type `o` again. This will compute the second and final descendent of the root. Note that the root has now changed from light blue indicating an open choice node to dark blue indicating a closed choice node.

zooming the search tree The right vertical scroll bar of the Explorer's window zooms the size of the displayed search tree. You can zoom the tree to fit the size of the window by clicking the zoom bar with the right mouse button.

Exercise 3.1 *With the Explorer it is easy to observe the effect of different distribution strategies. Replace the first-fail distribution strategy in `Money` with the naive strategy*

```
{FD.distribute naive Root}
```

which distributes on the leftmost undetermined variable and its least possible value. Draw the new search tree with the Explorer and observe that it is twice as large as the tree obtained with first-fail distribution.

Exercise 3.2 *Write a script that finds distinct digits for the letters A, B, D, E, G, L, N, O, R, and T such that the equation*

$$DONALD + GERALD = ROBERT$$

holds without leading zeros. Run the script with the Explorer and study the search tree. Try both first-fail and naive distribution. Observe that first-fail distribution yields a search tree that is by one order of magnitude smaller than the search tree obtained with naive distribution.

3.4 New Primitives

This section gives you an idea of the new Oz primitives needed to express search engines and finite domain scripts.

The new primitives come in two orthogonal groups. The first group provides the ability to create and distribute spaces and to explore search trees. This ability is essential for the implementation of search engines based on the propagate and distribute paradigm. As was mentioned before, this paradigm is general and applies also to constraints other than finite domain constraints.

The second group of primitives provides the ability to create finite domain propagators and to tell domain constraints to the constraint store. It also provides the ability to access the domain of a variable in the current constraint store, an expressivity needed for programming distribution strategies.

first-class spaces Oz provides spaces as first-class citizens that can be created, distributed, and killed, among other things. A first-class space is almost like Oz's unique top-level space, where regular computation takes place. Like the top-level space, first-class spaces have a constraint store, a procedure store, and a cell store and can host any number of threads. One important difference between the top-level and first-class spaces is the treatment of failure, which is considered an error at the top level and a regular event in first-class spaces (*this space has no solution*).

attributes of global objects cannot be assigned The parent of a first-class space is the space that created it. The constraint and the procedure store of a first-class space inherit all constraints and procedures in the respective stores of the parent space. However, a first-class space has no write access to the cell store of its parent space. Consequently, it is impossible to assign in a first-class space attributes of objects that belong to an ancestor space. On the other hand, a first-class space can create its own objects and apply them freely.

The details of first-class spaces need only concern programmers who want to implement new search engines. For finite domain problems, the necessary search engines are already available as predefined functionality (e.g., `SearchOne` and `SearchAll`).

All functionality related to finite domain constraints is provided through the procedures of the module `FD`. We have already seen `FD.distinct` (creates a propagator) and `FD.distribute` (creates a distributor) in the script for the Send More Money Puzzle (see Section 3.2).

infix notations For some of the procedures of the module `FD`, Oz provides special infix notations governed by the following operators:

`:: ::: =: \=: <: >: =<: >=:`

You have already seen examples of the use of `:::`, `\=:`, and `=:` in the script `Money`. An equivalent version of `Money` not using these notational conveniences appears in Figure 3.4

Figure 3.4: A script for Money that does not use the infix notations `==` and `\=:`.

```

proc {Money Root}
  S E N D M O R Y
in
  Root = sol(s:S e:E n:N d:D m:M o:O r:R y:Y)
  {FD.dom 0#9 Root}
  {FD.distinct Root}
  {FD.sum [S] '\=: ' 0}
  {FD.sum [M] '\=: ' 0}
  {FD.sumC
    [1000 100 10 1 1000 100 10 1 ~10000 ~1000 ~100 ~10 ~1]
    [ S E N D M O R E M O N E Y]
    '=:'
    0}
  {FD.distribute ff Root}
end

```

3.5 Watching Propagators

It is illuminating to watch the effect of one or several propagators with the Browser. Enter the following statements line by line and observe in the Browser the shrinking domains of the variables `x`, `y`, and `z`:

```

declare X Y Z
{Browse [X Y Z]}           % [X Y Z]
X :: 1#13                   % [X[1#13] Y Z]
Y :: 0#27                   % [X[1#13] Y[0#27] Z]
Z :: 1#12                   % [X[1#13] Y[0#27] Z[1#12]]
2*Y == Z                   % [X[1#13] Y[1#6] Z[2#12]]
X <: Y                     % [X[1#5] Y[2#6] Z[4#12]]
Z <: 7                     % [X[1#2] Y[2#3] Z[4#6]]
X \=: 1                    % [2 3 6]

```

The comments say what you will see in the Browser. Note that the statement `2*Y==Z` creates a propagator that performs interval rather than domain propagation.

3.6 Example: Safe

Problem Specification

The code of Professor Smart's safe is a sequence of 9 distinct nonzero digits C_1, \dots, C_9 such that the following equations and inequations are satisfied:

$$\begin{aligned} C_4 - C_6 &= C_7 \\ C_1 * C_2 * C_3 &= C_8 + C_9 \\ C_2 + C_3 + C_6 &< C_8 \\ C_9 &< C_8 \\ C_1 &\neq 1, \dots, C_9 \neq 9 \end{aligned}$$

Can you determine the code?

Model and Distribution Strategy

We choose the obvious model that has a variable for every digit C_1, \dots, C_9 . We distribute over these variables with the standard first-fail strategy.

Figure 3.5: A script for the Safe Puzzle.

```
proc {Safe C}
  {FD.tuple code 9 1#9 C}
  {FD.distinct C}
  C.4 - C.6 == C.7
  C.1 * C.2 * C.3 == C.8 + C.9
  C.2 + C.3 + C.6 < C.8
  C.9 < C.8
  {For 1 9 1 proc {$ I} C.I \=: I end}
  {FD.distribute ff C}
end
```

Script

Figure 3.5 shows a script for the Safe Puzzle. The statement

```
{FD.tuple code 9 1#9 C}
```

constrains the root variable `C` to a tuple with label `code` whose components are integers in the domain `1#9`. The statement

```
{For 1 9 1 proc {$ I} C.I \=: I end}
```

posts the constraint $c.i \neq i$ for every $i = 1, \dots, 9$.

The full search tree of `Safe` has 23 nodes and contains the unique solution:

```
code(4 3 1 8 9 2 6 7 5)
```

Elimination of Symmetries and Defined Constraints

In this section you will learn two basic constraint programming techniques. The first technique consists in eliminating symmetries in the model, which often leads to scripts with smaller search trees. The second technique introduces defined constraints, a means for writing modular and concise scripts.

4.1 Example: Grocery

This example illustrates that elimination of symmetries can dramatically reduce the size of search trees.

Problem Specification

A kid goes into a grocery store and buys four items. The cashier charges \$7.11, the kid pays and is about to leave when the cashier calls the kid back, and says ‘Hold on, I multiplied the four items instead of adding them; I’ll try again; Hah, with adding them the price still comes to \$7.11’. What were the prices of the four items?

Model

Our model has four variables A , B , C , and D , which stand for the prices of the four items. In order that the variables can be constrained to finite domains of integers, we assume that the prices are given in cents. To say that the sum of the four prices is 711, we impose the constraint $A + B + C + D = 711$, and to say that the product of the four prices is 711, we impose the constraint

$$A \cdot B \cdot C \cdot D = 711 \cdot 100 \cdot 100 \cdot 100$$

The model admits many different equivalent solutions since the prices of the items can be interchanged. We can eliminate these symmetries by imposing an order on the prices of the items, for instance,

$$A \leq B \leq C \leq D.$$

With these ordering constraints the model has a unique solution.

Distribution Strategy

For this problem it is advantageous to use a first-fail strategy that splits the domain of the selected variable and tries the upper part of the domain first. This strategy leads to a much smaller search tree than the standard first-fail strategy, which tries the least possible value of the selected variable first.

Figure 4.1: A script for the Grocery Puzzle.

```

proc {Grocery Root}
  A#B#C#D = Root
  S      = 711
in
  Root ::: 0#S
  A+B+C+D == S
  A*B*C*D == S*100*100*100
  %% eliminate symmetries
  A <=: B
  B <=: C
  C <=: D
  {FD.distribute generic(value:splitMax) Root}
end

```

Script

The script in Figure 4.1 spawns a search tree with 5039 nodes. It will explore 566 nodes before it finds the unique solution `120#125#150#316`. Without the ordering constraints the script explores more than three times as many nodes before finding a first solution. We learn that the elimination of symmetries may make it easier to find the first solution.

A Subtle Symmetry

There exists another symmetry whose elimination leads to a much smaller search tree. For this we observe that 711 has the prime factor 79 ($711 = 9 \cdot 79$). Since the product of the prices of the items is 711, we can assume without loss of generality that 79 is a prime factor of the price `A` of the first item. We adapt our script by replacing the statement `A <=: B` with

```
A == 79*{FD.decl}
```

The procedure `{FD.decl X}` constrains its argument to an integer in the finite domain `0#sup`, where `sup` stands for a large implementation-dependent integer (134217726 in Mozart on Linux or Sparcs).

The new propagator for `A == 79*X` reduces the search tree of `Grocery` to 357 nodes, one order of magnitude less than before. The solution of the problem is now found after exploring 44 nodes.

4.2 Example: Family

defined constraints This example illustrates the use of defined constraints. A defined constraint is a procedure

```
{DefinedConstraint X1 ... Xn}
```

posting constraints on the variables x_1, \dots, x_n . The reasons for introducing defined constraints are more or less the same as for introducing defined procedures in ordinary programming.

The script for the example will employ the procedures `FD.sum`, `FD.sumC`, and `FD.sumCN`, which create propagators for linear and nonlinear summation constraints.

Problem Specification

Maria and Clara are both heads of households, and both families have three boys and three girls. Neither family includes any children closer in age than one year, and all children are under age 10. The youngest child in Maria's family is a girl, and Clara has just given birth to a little girl.

In each family, the sum of the ages of the boys equals the sum of the ages of the girls, and the sum of the squares of the ages of the boys equals the sum of the the squares of ages of the girls. The sum of the ages of all children is 60.

What are the ages of the children in each family?

Model

We model a family as a record

```
Name(boys:[B1 B2 B3] girls:[G1 G2 G3])
```

where the variables $B1$, $B2$ and $B3$ stand for the ages of the boys in descending order (i.e., $B3$ is the age of the youngest boy in the family), and the variables $G1$, $G2$ and $G3$ stand for the ages of the girls, also in descending order. This representation of a family avoids possible symmetries. The constraints that must hold for a family F with name N will be posted by the defined constraint `{IsFamily N F}`.

A solution is a pair consisting of Maria's and Clara's family.

Distribution Stratgey

We distribute on the list of the ages of the children of the two families following a first-fail strategy. The strategy splits the domain of the selected variable and tries the lower part of the domain first.

Figure 4.2: A script for the Family Puzzle.

```

proc {Family Root}
  <Definition of IsFamily 29b>
  <Definition of AgeList 29a>
  Maria = {IsFamily maria}
  Clara = {IsFamily clara}
  AgeOfMariasYoungestGirl = {Nth Maria.girls 3}
  AgeOfClarasYoungestGirl = {Nth Clara.girls 3}
  Ages = {FoldR [Clara.girls Clara.boys Maria.girls Maria.boys]
            Append nil}
in
  Root = Maria#Clara
  {ForAll Maria.boys proc {$ A} A >: AgeOfMariasYoungestGirl end}
  AgeOfClarasYoungestGirl = 0
  {FD.sum Ages '=: ' 60}
  {FD.distribute split Ages}
end

```

Script

The script in Figure 4.2 introduces two defined constraints. The defined constraint

$$F = \{\text{IsFamily } \textit{Name}\}$$

imposes constraints saying that F is the representation of a family with name \textit{Name} (see Figure 4.3). The defined constraint

$$L = \{\text{AgeList}\}$$

imposes constraints saying that L is a list of three integers between 0 and 9 appearing in descending order (see Figure 4.3).

The statement

$$\{\text{FD.sumC Coeffs Ages '=: ' 0}\}$$

creates a propagator for the constraint

$$1 \cdot B_1 + 1 \cdot B_2 + 1 \cdot B_3 + (-1) \cdot G_1 + (-1) \cdot G_2 + (-1) \cdot G_3 = 0$$

saying that the sum of the ages of the boys equals the sum of the ages of the girls. The statement

$$\{\text{FD.sumCN Coeffs \{Map Ages fun \{ \$ A \} [A A] end\} '=: ' 0}\}$$

Figure 4.3: Defined constraints for the Family Puzzle.

29a **<Definition of AgeList 29a>**≡

```

proc {AgeList L}
  {FD.list 3 0#9 L}
  {Nth L 1} >: {Nth L 2}
  {Nth L 2} >: {Nth L 3}
end

```

29b **<Definition of IsFamily 29b>**≡

```

fun {IsFamily Name}
  Coeffs = [1 1 1 ~1 ~1 ~1]
  BoysAges = {AgeList}
  GirlsAges = {AgeList}
  Ages = {Append BoysAges GirlsAges}
in
  {FD.distinct Ages}
  {FD.sumC Coeffs Ages '=: ' 0}
  {FD.sumCN Coeffs {Map Ages fun {$ A} [A A] end} '=: ' 0}
  Name(boys:BoysAges girls:GirlsAges)
end

```

creates a propagator for the constraint

$$1 \cdot B_1 \cdot B_1 + 1 \cdot B_2 \cdot B_2 + 1 \cdot B_3 \cdot B_3 + (-1) \cdot G_1 \cdot G_1 + (-1) \cdot G_2 \cdot G_2 + (-1) \cdot G_3 \cdot G_3 = 0$$

saying that the sum of the squares of the ages of the boys equals the sum of the squares of the ages of the girls. The statement

```
{FD.sum Ages '=: ' 60}
```

creates a propagator for the constraint saying that the sum of the ages of all kids equals 60.

4.3 Example: Zebra Puzzle

This example shows a clever problem representation avoiding possible symmetries. It also illustrates the use of defined constraints.

Problem Specification

Five men with different nationalities live in the first five houses of a street. There are only houses on one side of the street. The men practice distinct professions, and each of them has a favorite drink and a favorite animal, all of them different. The five houses are painted with different colors. The following facts are known:

1. The Englishman lives in a red house.

2. The Spaniard owns a dog.
3. The Japanese is a painter.
4. The Italian drinks tea.
5. The Norwegian lives in the first house.
6. The owner of the green house drinks coffee.
7. The green house comes after the white one.
8. The sculptor breeds snails.
9. The diplomat lives in the yellow house.
10. Milk is drunk in the third house.
11. The Norwegian's house is next to the blue one.
12. The violinist drinks juice.
13. The fox is in the house next to that of the doctor.
14. The horse is in the house next to that of the diplomat.
15. The zebra is in the white house.
16. One of the men drinks water.

Who lives where?

Model

We number the houses from 1 to 5, where 1 is the first and 5 is last house in the street. There are 25 different properties (i.e. hosting an Englishman, being the green house, hosting a painter, hosting a dog, or hosting someone who drinks juice), and each of these properties must hold for exactly one house. The properties are partitioned into five groups of five members each, where the properties within one group must hold for different houses. The model has one variable for each of these properties, where the variable stands for the number of the house for which this property holds.

Distribution Strategy

We distribute on the variables for the properties with the standard first-fail strategy.

Script

Figure 4.4 shows a script based on the outlined model and distribution strategy. The script constrains the root variable `Nb` to a record that maps every property to a house number between 1 and 5.

The script introduces two defined constraints. The defined constraint

```
{Partition Group}
```

Figure 4.4: A script for the Zebra Puzzle.

```

proc {Zebra Nb}
  Groups      = [ [english spanish japanese italian norwegian]
                  [green red yellow blue white]
                  [painter diplomat violinist doctor sculptor]
                  [dog zebra fox snails horse]
                  [juice water tea coffee milk] ]
  Properties = {FoldR Groups Append nil}
  proc {Partition Group}
    {FD.distinct {Map Group fun {$ P} Nb.P end}}
  end
  proc {Adjacent X Y}
    {FD.distance X Y '=' 1}
  end
in
  %% Nb maps all properties to house numbers
  {FD.record number Properties 1#5 Nb}
  {ForAll Groups Partition}
  Nb.english = Nb.red
  Nb.spanish = Nb.dog
  Nb.japanese = Nb.painter
  Nb.italian = Nb.tea
  Nb.norwegian = 1
  Nb.green = Nb.coffee
  Nb.green >: Nb.white
  Nb.sculptor = Nb.snails
  Nb.diplomat = Nb.yellow
  Nb.milk = 3
  {Adjacent Nb.norwegian Nb.blue}
  Nb.violinist = Nb.juice
  {Adjacent Nb.fox Nb.doctor}
  {Adjacent Nb.horse Nb.diplomat}
  Nb.zebra = Nb.white
  {FD.distribute ff Nb}
end

```

says that the properties in the list `Group` must hold for pairwise distinct houses. The defined constraint

```
{Adjacent X Y}
```

says that the properties X and Y must hold for houses that are next to each other. The statement

```
{FD.distance X Y '=: ' 1}
```

creates a propagator for $|X - Y| = 1$.

The script defines a search tree with 33 nodes. The unique solution is the record

```
number(
  blue:2      coffee:5      diplomat:3  doctor:4
  dog:3       english:4     fox:5       green:5
  horse:4     italian:2     japanese:5  juice:1
  milk:3      norwegian:1   painter:5   red:4
  sculptor:2  snails:2      spanish:3   tea:2
  violinist:1 water:4       white:1     yellow:3
  zebra:1
)
```

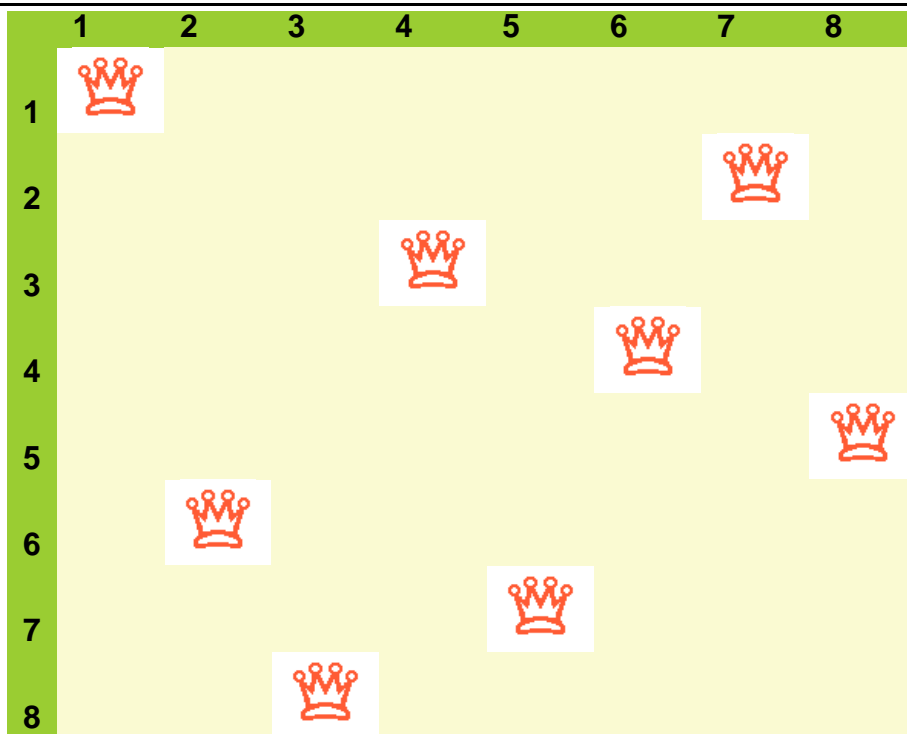
Parameterized Scripts

Combinatorial problems typically occur in a general form that can be instantiated with different data sets. This leads to parameterized scripts separating the general script from particular data sets.

5.1 Example: Queens

Problem Specification

Place N queens on an $N \times N$ chess board such that no two queens attack each other. The parameter of the problem is N . A solution for the 8-queens problem looks as follows:



Model

We will use a clever model avoiding possible symmetries and minimizing the number of propagators.

We assume that the queens are numbered from 1 to N , and that the k -th queen is always placed in the k -th column. For every queen i we have one variable R_i saying in which row the queen is placed. The model guarantees by construction that two queens are never placed in the same column. To ensure that two queens are never in the same row, we impose the constraint that the variables R_1, \dots, R_N are pairwise distinct.

To enforce that two queens are never in the same diagonal, we need to impose the constraints

$$R_i + (j - i) \neq R_j \quad \text{and} \quad R_i - (j - i) \neq R_j$$

for all i, j such that $1 \leq i < j \leq N$. Equivalently, we can impose the constraints

$$R_i - i \neq R_j - j \quad \text{and} \quad R_i + i \neq R_j + j$$

for all i, j such that $1 \leq i < j \leq N$. This is equivalent to saying that the sequences

$$R_1 - 1, \dots, R_N - N \quad \text{and} \quad R_1 + 1, \dots, R_N + N$$

are both nonrepetitive. Since Oz has a special propagator for the constraint stating the nonrepetitiveness of such sequences, this formulation requires only two propagators, one for each sequence.

Distribution Strategy

We distribute on the variables R_1, \dots, R_N using a first-fail strategy that tries the value in the middle of the domain of the selected variable first. This strategy works well even for large N .

Figure 5.1: A script for the N -queens Problem.

```

fun {Queens N}
  proc {$ Row}
    L1N={MakeTuple c N}
    LM1N={MakeTuple c N}
  in
    {FD.tuple queens N l#N Row}
    {For 1 N 1 proc {$ I}
      L1N.I=I LM1N.I=~I
    end}
    {FD.distinct Row}
    {FD.distinctOffset Row LM1N}
    {FD.distinctOffset Row L1N}
    {FD.distribute generic(value:mid) Row}
  end
end

```

Script

Figure 5.1 shows a parameterized script for the N -Queens Problem. The actual script is created by the procedure `Queens`, which takes `N` as parameter. The script constrains its root variable `Row` to a tuple having a component for every queen. This implicitly creates the variables R_1, \dots, R_N of the model.

The statements

```
{FD.distinct Row}
{FD.distinctOffset Row LM1N}
{FD.distinctOffset Row L1N}
```

create propagators for the constraints stating that the sequences

| | | |
|----------------------|------------------|----------------------|
| <code>Row.1</code> | <code>...</code> | <code>Row.N</code> |
| <code>Row.1-1</code> | <code>...</code> | <code>Row.N-N</code> |
| <code>Row.1+1</code> | <code>...</code> | <code>Row.N+N</code> |

be non repetitive.

5.2 Example: Changing Money

Problem Specification

Given bills and coins of different denominations and an amount A , select a minimal number of bill and coins to pay A . One instance of the problem assumes that we want to pay the amount of 1.42, and that we have 6 one dollar bills, 8 quarters (25 cents), 10 dimes (10 cents), 1 nickel (5 cents), and 5 pennies (1 cent).

Model

To avoid conversions, we assume that the amount to be paid and all denominations are specified in the same currency unit (e.g., cents). The data is specified by variables a_1, \dots, a_k specifying the available denominations d_i and the number a_i of available respective coins or bills.

The model has a variable C_i for ever available denomination saying how many of the corresponding bills or coins we will use to pay the amount. For all i , we must have $0 \leq C_i \leq a_i$ Moreover, we must satisfy the constraint

$$d_1 \cdot C_1 + d_2 \cdot C_2 + \dots + d_k \cdot C_k = \text{amount}$$

Distribution Strategy

We distribute on the variables C_1, C_2, \dots , where we give precedence to larger denominations and, with second priority, to larger values.

Figure 5.2: A script for changing money together with a data specification.

```

fun {ChangeMoney BillsAndCoins Amount}
  Available      = {Record.map BillsAndCoins fun {$ A#_} A end}
  Denomination   = {Record.map BillsAndCoins fun {$ _#D} D end}
  NbDenoms       = {Width Denomination}
in
  proc {$ Change}
    {FD.tuple change NbDenoms 0#Amount Change}
    {For 1 NbDenoms 1 proc {$ I} Change.I <=: Available.I end}
    {FD.sumC Denomination Change '=: ' Amount}
    {FD.distribute generic(order:naive value:max) Change}
  end
end

BillsAndCoins = bac(6#100 8#25 10#10 1#5 5#1)

```

Script

The procedure `ChangeMoney` in Figure 5.2 takes two parameters specifying the available bills and coins and the amount to be paid. It returns a script that enumerates the possible ways to pay the specified amount with the specified bills and coins. It is assumed that the bills and coins are specified in denomination decreasing order.

The statement

```
{Browse {SearchOne {ChangeMoney BillsAndCoins 142}}}
```

computes the list

```
[change(1 1 1 1 2)]
```

saying that we can pay \$1.42 with 1 one dollar bill, 1 quarter, 1 dime, 1 nickel, and 2 pennies. This payment uses the minimal number of bills and coins. The number of different possibilities to pay \$1.42 with the specified stock of bills and coins is 6 and can be computed with the statement

```
{Browse {Length {SearchAll {ChangeMoney BillsAndCoins 142}}}}
```

Minimizing a Cost Function

In many applications one is interested in solutions that minimize a given cost function. If the cost function is simple enough, we can obtain the minimization effect by employing a two-dimensional distribution strategy.

This section will present two examples, map coloring and conference scheduling, for which a two-dimensional distribution strategy suffices. For each of the two examples we will develop a parameterized script.

6.1 Example: Coloring a Map

Problem Specification

Given a map showing the West European countries Netherlands, Belgium, France, Spain, Portugal, Germany, Luxemburg, Switzerland, Austria, and Italy, find a coloring such that neighboring countries have different color and a minimal number of colors is used.

Model

We have a variable *NbColors* saying how many different colors we can use. Moreover, we have a variable for every country. For every pair A, B of countries having a border in common we impose the constraint $A \neq B$. We represent colors as numbers. Hence we constrain all variables for countries to integers in $0\#NbColors$.

Distribution Strategy

We first distribute on *NbColors*, trying the numbers $0, 1, 2, \dots$ in ascending order. After *NbColors* is determined, we distribute on the variables for the countries using the usual first-fail strategy.

Script

The script appears in Figure 6.1. It is parameterized with the specification of the map to be colored. The figure shows the specification of a map containing some European countries.

Figure 6.1: A script for the Map Coloring Problem together with a data specification.

```

fun {MapColoring Data}
  Countries = {Map Data fun {$ C#_} C end}
in
  proc {$ Color}
    NbColors = {FD.decl}
  in
    {FD.distribute naive [NbColors]}
    %% Color: Countries --> 1#NbColors
    {FD.record color Countries 1#NbColors Color}
    {ForAll Data
      proc {$ A#Bs}
        {ForAll Bs proc {$ B} Color.A \=: Color.B end}
      end}
    {FD.distribute ff Color}
  end
end

Data = [ austria      # [italy switzerland germany]
         belgium      # [france netherlands germany luxemburg]
         france       # [spain luxemburg italy]
         germany      # [austria france luxemburg netherlands]
         italy        # nil
         luxemburg    # nil
         netherlands # nil
         portugal     # nil
         spain        # [portugal]
         switzerland # [italy france germany austria] ]

```

The script first creates a local variable `NbColors` that specifies the number of different colors to be used for coloring the map. Then it distributes naively on `NbColors`. Recall that a distributor blocks its thread until it has done its job. After `NbColors` is determined by distribution, the variable `Color` is constrained to a record mapping the country names to integers in `1#NbColors`. This implicitly creates the variables for the Countries. Next the script creates a propagator

```
Color.A \=: Color.B
```

for every pair `A, B` of bordering countries. Finally, the script distributes on `Color` using the first-fail strategy.

The statement

```
{ExploreOne {MapColoring Data}}
```

will show the search tree explored to find the first solution, which looks as follows:

```

color(
  austria:    1    belgium:  3    france:    1
  germany:    2    italy:    2    luxemburg:  4
  netherlands: 1    portugal: 1    spain:      2
  switzerland: 3
)

```

The search tree of `MapColoring` is interesting. First, colorings with 0, 1, 2 and 3 colors are searched and not found. Then a coloring with 4 colors is searched. Now a solution is found quickly, without going through further failure nodes. There are many solutions using 4 colors since the particular color given to a country does not matter.

6.2 Example: Conference

This example will employ the constraint provided by `FD.atMost`.

Problem Specification

We want to construct the time table of a conference. The conference will consist of 11 sessions of equal length. The time table is to be organized as a sequence of slots, where a slot can take up to 3 parallel sessions. There are the following constraints on the timing of the sessions:

1. Session 4 must take place before Session 11.
2. Session 5 must take place before Session 10.
3. Session 6 must take place before Session 11.
4. Session 1 must not be in parallel with Sessions 2, 3, 5, 7, 8, and 10.
5. Session 2 must not be in parallel with Sessions 3, 4, 7, 8, 9, and 11.
6. Session 3 must not be in parallel with Sessions 5, 6, and 8.
7. Session 4 must not be in parallel with Sessions 6, 8, and 10.
8. Session 6 must not be in parallel with Sessions 7 and 10.
9. Session 7 must not be in parallel with Sessions 8 and 9.
10. Session 8 must not be in parallel with Session 10.

The time table should minimize the number of slots.

Model

The model has a variable *NbSlots* saying how many slots the conference has. For the given data, *NbSlots* can be constrained to the finite domain $4\#11$. The model also has a variable *Plan_i* for every session *i*, where *Plan_i* stands for the number of the slot in which Session *i* will take place. Every variable *Plan_i* can be constrained to the finite domain $1\#NbSlots$. The remaining constraints are obvious from the problem description.

Distribution Strategy

We use a two-dimensional distribution strategy. We first distribute on *NbSlots*, trying smaller values first. Once *NbSlots* is determined, we distribute on the variables $Plan_1, \dots, Plan_{11}$ using the standard first-fail strategy.

Script

The script in Figure 6.2 is parameterized with an argument *Data* specifying the conference to be organized. The figure also shows the specification of the conference described in the problem specification.

The script creates a local variable *NbSlots* specifying the number of slots used by the conference. It then distributes naively on *NbSlots*. After *NbSlots* is determined, it constrains its root variable *Plan* to a tuple mapping the session numbers 1, ..., 11 to integers in $1\#NbSlots$. This implicitly creates variables corresponding to the variables $Plan_i$ of the model.

The statement

```
{FD.atMost NbParSessions Plan Slot}
```

creates a propagator for a constraint saying that at most *NbParSessions* components of *Plan* can be equal to *Slot*.

The statement `{ForAll Constraints ... }` imposes the constraints of the conference to be scheduled.

The last statement distributes on *Plan* using the first-fail strategy.

The statement

```
{ExploreOne {Conference Data}}
```

will explore the search tree until the first solution is found. The first solution minimizes the number of slots and looks as follows:

```
plan(1 2 3 1 2 2 3 4 1 3 4)
```

This plan says that the conference requires 4 slots, where the sessions 1, 4 and 9 take place in slot 1, the sessions 2, 5 and 6 take place in slot 2, the sessions 3, 7 and 10 take place in slot 3, and the sessions 8 and 11 take place in slot 4.

Figure 6.2: A script for conference scheduling together with a data specification.

```

fun {Conference Data}
  NbSessions      = Data.nbSessions
  NbParSessions   = Data.nbParSessions
  Constraints      = Data.constraints
  MinNbSlots      = NbSessions div NbParSessions
in
  proc {$ Plan}
    NbSlots = {FD.int MinNbSlots#NbSessions}
  in
    {FD.distribute naive [NbSlots]}
    %% Plan: Session --> Slot
    {FD.tuple plan NbSessions 1#NbSlots Plan}
    %% at most NbParSessions per slot
    {For 1 NbSlots 1
      proc {$ Slot} {FD.atMost NbParSessions Plan Slot} end}
    %% impose constraints
    {ForAll Constraints
      proc {$ C}
        case C
        of before(X Y) then
          Plan.X <: Plan.Y
        [] disjoint(X Ys) then
          {ForAll Ys proc {$ Y} Plan.X \=: Plan.Y end}
        end
      end}
    {FD.distribute ff Plan}
  end
end

Data = data(nbSessions:11 nbParSessions:3
  constraints: [ before(4 11) before(5 10) before(6 11)
    disjoint(1 [2 3 5 7 8 10])
    disjoint(2 [3 4 7 8 9 11])
    disjoint(3 [5 6 8]) disjoint(4 [6 8 10])
    disjoint(6 [7 10]) disjoint(7 [8 9])
    disjoint(8 [10]) ] )

```

Propagators for Redundant Constraints

For some problems, the performance of a script can be drastically improved by introducing propagators for redundant constraints. Redundant constraints are constraints that are entailed by the constraints specifying the problem. Additional propagators for redundant constraints may decrease the size of the search tree by strengthening the propagation component of the script. They may also reduce the number of propagation steps needed to reach stability.

7.1 Example: Fractions

Problem Specification

The Fractions Puzzle consists in finding distinct nonzero digits such that the following equation holds:

$$\frac{A}{BC} + \frac{D}{EF} + \frac{G}{HI} = 1$$

Model

We have a variable for every letter, similar as in the Send More Money Puzzle. Since the three fractions are symmetric, we can impose the order

$$\frac{A}{BC} \geq \frac{D}{EF} \geq \frac{G}{HI}$$

From the order constraints we obtain the redundant constraints

$$3\frac{A}{BC} \geq 1 \quad \text{and} \quad 3\frac{G}{HI} \leq 1$$

The order constraints together with the redundant constraints reduce the size of the search tree by one order of magnitude.

Distribution Strategy

We distribute on the list of letters using the standard first-fail strategy.

Figure 7.1: A script for the Fractions Puzzle.

```

proc {Fractions Root}
  sol(a:A b:B c:C d:D e:E f:F g:G h:H i:I) = Root
  BC = {FD.decl}
  EF = {FD.decl}
  HI = {FD.decl}
in
  Root ::: 1#9
  {FD.distinct Root}
  BC =: 10*B + C
  EF =: 10*E + F
  HI =: 10*H + I
  A*EF*HI + D*BC*HI + G*BC*EF =: BC*EF*HI
  %% impose canonical order
  A*EF >=: D*BC
  D*HI >=: G*EF
  %% redundant constraints
  3*A >=: BC
  3*G <=: HI
  {FD.distribute ff Root}
end

```

Script

The script in Figure 7.1 constrains its root variable to a record having a field for every letter. Since Oz has no finite domain propagators for fractions, we eliminate the fractions by multiplying with the denominators. For every denominator we introduce an auxiliary variable. Since a finite domain propagator starts its work only after all variables of the constraint implemented by the propagator are constrained to finite domains in the constraint store, the script constrains the auxiliary variables for the denominators to the maximal finite domain using the procedure `FD.decl`.

7.2 Example: Pythagoras

Not all propagators exploit coreferences in products (e.g. $x \cdot x + y \cdot y = z \cdot z$). For the example of this section it will be essential to exploit such coreferences, and you will learn how to do it.

The example also illustrates the case where a propagator for a redundant constraint improves the performance of a script by decreasing the number of necessary propagation steps, but without significantly changing the search tree.

Problem Specification

How many triples (A, B, C) exist such that $A^2 + B^2 = C^2$ and $A \leq B \leq C \leq 1000$?

Model

The model has three variables A , B , and C . Each variable is constrained to the finite domain $1\#1000$. The model imposes the constraints $A^2 + B^2 = C^2$ and $A \leq B \leq C$.

The script will also create a propagator for the redundant constraint $2 \cdot B^2 \geq C^2$.

Distribution Strategy

We distribute on the variables A , B , C using the standard first-fail strategy.

Figure 7.2: A script that enumerates Pythagoras triples.

```

proc {Square X S}
  {FD.times X X S}
end

proc {Pythagoras Root}
  [A B C] = Root
  AA BB CC
in
  Root ::: 1#1000
  AA = {Square A}
  BB = {Square B}
  CC = {Square C}
  AA + BB == CC
  A <= B
  B <= C
  2*BB >= CC % redundant constraint
  {FD.distribute ff Root}
end

```

Script

Given the script in Figure 7.2, we can compute the number of different triples with the statement

```
{Browse {Length {SearchAll Pythagoras}}}
```

The script introduces a defined constraint

```
{Square X S}
```

saying that s is the square of x . This constraint is implemented with a propagator provided by

```
{FD.times X X S}
```

This propagator will start propagating as soon as the store constrains `x` to a finite domain. This in contrast to the propagator created by `x*x=:S`, which will start work only after both `x` and `S` are constrained to finite domains in the constraint store. To define `Square` with this constraint, we would have to write

```
proc {Square X S}
  {FD.decl S}
  X*X =: S
end
```

The propagator for the redundant constraint does not significantly reduce the size of the search tree. However, it reduces the number of propagation steps from about 1000000 to about 500000, which makes computation twice as fast.

statistics To find out about this, pop up the Oz Panel¹ and reset the statistics. Also switch on the status message feature and pop up the emulator buffer. Now enter the statement

```
{Browse {Length {SearchAll Pythagoras}}}
```

and print the statistics after the execution of the statement has finished. This will show something like

```
solutions: 881      Variables created:      3
clones:    1488     Propagators created:     7
failures:  608     Propagator invocations: 490299
```

in the emulator buffer. Now remove the propagator for the redundant constraint from the definition of the script, redefine it, reset the statistics, run the statement, and print the statistics. This time you will see something like

```
solutions: 881      Variables created:      3
clones:    1878     Propagators created:     6
failures:  998     Propagator invocations: 1190397
```

If we drop the redundant constraint, it seems sensible to not have separate propagators for the squares but simply have one propagator created with

```
A*A + B*B =: C*C
```

Unfortunately, this will dramatically increase the size of the search tree. The reason for this increase is the fact that the created propagator does not realize the coreferences in the constraint it implements, that is, it treats the two occurrences of `A`, say, as if they were independent variables.

¹“Oz Panel”

7.3 Example: Magic Squares

This example shows a smart representation of a matrix and a concomitant defined constraint of higher order. The model will eliminate symmetries by imposing order constraints. Propagation will be drastically improved by exploiting a redundant constraint.

Problem Specification

The Magic Square Puzzle consists in finding for given N an $N \times N$ -matrix such that:

- Every field of the matrix is an integer between 1 and N^2 .
- The fields of the matrix are pairwise distinct.
- The sums of the rows, columns, and the two main diagonals are all equal.

A magic square for $N = 3$ looks as follows:

$$\begin{array}{ccc} 2 & 7 & 6 \\ 9 & 5 & 1 \\ 4 & 3 & 8 \end{array}$$

p> The Magic Square Puzzle is extremely hard for large N . Even for $N = 5$, our script will have to explore almost 8000 nodes to find a solution.

Model

We model the problem by having a variable F_{ij} for every field (i, j) of the matrix. Moreover, we have one additional variable S and require that the sum of every row, column, and main diagonal equals S .

Without loss of generality, we can impose the following order constraints eliminating symmetries:

$$F_{11} < F_{NN}, \quad F_{N1} < F_{1N}, \quad F_{11} < F_{N1}.$$

Since the sum of the sums of the rows must equal the sum of all fields, we have the redundant constraint

$$\frac{N^2}{2} \cdot (N^2 + 1) = N \cdot S$$

To see this, note that sum of all fields equals

$$1 + 2 + \dots + N^2 = \frac{N^2}{2} \cdot (N^2 + 1)$$

and that the sum of each of the N rows must be S .

Distribution Strategy

We distribute on the variables F_{11}, \dots, F_{NN} with a first-fail strategy splitting the domain of the selected variable and trying the lower half first.

Figure 7.3: A script for the Magic Square Puzzle.

```

fun {MagicSquare N}
  NN = N*N
  L1N = {List.number 1 N 1} % [1 2 3 ... N]
in
  proc {$ Square}
    fun {Field I J}
      Square.((I-1)*N + J)
    end
    proc {Assert F}
      % {F 1} + {F 2} + ... + {F N} =: Sum
      {FD.sum {Map L1N F} '=: ' Sum}
    end
    Sum = {FD.decl}
  in
    {FD.tuple square NN 1#NN Square}
    {FD.distinct Square}
    %% Diagonals
    {Assert fun {$ I} {Field I I} end}
    {Assert fun {$ I} {Field I N+1-I} end}
    %% Columns
    {For 1 N 1
      proc {$ I} {Assert fun {$ J} {Field I J} end} end}
    %% Rows
    {For 1 N 1
      proc {$ J} {Assert fun {$ I} {Field I J} end} end}
    %% Eliminate symmetries
    {Field 1 1} <: {Field N N}
    {Field N 1} <: {Field 1 N}
    {Field 1 1} <: {Field N 1}
    %% Redundant: sum of all fields = (number rows) * Sum
    NN*(NN+1) div 2 =: N*Sum
    %%
    {FD.distribute split Square}
  end
end

```

Script

Figure 7.3 shows a script realizing the model and distribution strategy just discussed. The actual script is created by a procedure `MagicSquare` taking `N` as parameter.

The script represents the matrix as a tuple with N^2 elements. The tuple is the value of the root variable `Square`. The function

$$\{\text{Field } I \ J\}$$

returns the component of `Square` that represents the field at position (I, J) . The variable `Sum` takes the sum of the rows, columns, and main diagonals as value. The procedure

$$\{\text{Assert } F\}$$

takes a function F and posts the constraint

$$\{F \ 1\} + \{F \ 2\} + \dots + \{F \ N\} = \text{Sum}$$

Obviously, $\{\text{Assert } F\}$ is a defined constraint of higher order. With the help of this defined constraint it is straightforward to state that the sums of the rows, columns, and main diagonals are all equal to `Sum`.

With the Explorer you can find out that for `N=3` there is exactly one magic square satisfying the ordering constraints of our model. Without the ordering constraints there are 8 different solutions. Omitting the propagator for the redundant constraint will increase the search tree by an order of magnitude.

7.4 Exercises

Exercise 7.1 Magic Sequence

A magic sequence of length n is a sequence

$$x_0, x_1, \dots, x_{n-1}$$

of integers such that for every $i = 0, \dots, n-1$

- *x_i is an integer between 0 and $n-1$.*
- *the number i occurs exactly x_i times in the sequence.*

Write a parameterized script that, given n , can enumerate all magic sequences of length n .

The script should use the procedure

$$\{\text{FD.exactly } K \ S \ I\}$$

that creates a propagator for the constraint saying that exactly K fields of the record S are equal to the integer I .

You can drastically reduce the search space of the script by having propagators for the redundant constraints

$$x_0 + \dots + x_{n-1} = n$$

and

$$(-1) \cdot x_0 + \dots + (n-2) \cdot x_{n-1} = 0$$

Explain why these constraints are redundant.

Reified Constraints

In this section we will see a new class of constraints called reified constraints. Reified constraints make it possible to express constraints involving logical connectives such as disjunction, implication, and negation. Reified constraints also make it possible to solve overconstrained problems, for which only some of the stated constraints can be satisfied.

8.1 Getting Started

reification of a constraint The *reification of a constraint* C with respect to a variable x is the constraint

$$(C \leftrightarrow x = 1) \wedge x \in 0\#1$$

where it is assumed that x does not occur free in C .

The operational semantics of a propagator for the reification of a constraint C with respect to x is given by the following rules:

1. If the constraint store entails $x = 1$, the propagator for the reification reduces to a propagator for C .
2. If the constraint store entails $x = 0$, the propagator for the reification reduces to a propagator for $\neg C$.
3. If a propagator for C would realize that the constraint store entails C , the propagator for the reification tells $x = 1$ and ceases to exist.
4. If a propagator for C would realize that the constraint store is inconsistent with C , the propagator for the reification tells $x = 0$ and ceases to exist.

To understand these rules, you need to be familiar with the definitions in Section 2.2.

0/1-variables A *0/1-variable* is a variable that is constrained to the finite domain $0\#1$. The control variables of reified constraints are 0/1-variables.

posting reified constraints Here are examples for statements creating propagators for reified constraints:

- `(X<:Y)=B` creates a propagator for the reification of $X < Y$ with respect to B .
- `(X+Y+Z=:0)=B` creates a propagator for the reification of $X + Y + Z = 0$ with respect to B .
- `(X\=:Y)=B` creates a propagator for the reification of $X \neq Y$ with respect to B .
- `(X=:0#10)=B` creates a propagator for the reification of $X \in 0\#10$ with respect to B .
- `{FD.reified.distance X Y '=: ' Z B}` creates a propagator for the reification of $|X - Y| = Z$ with respect to B .

expressing equivalences With reified constraints it is straightforward to express equivalences of constraints. For instance, the equivalence

$$X < Y \leftrightarrow X < Z$$

can be posted with the statement

```
X<:Y = X<:Z
```

This statement is a notational convenience for

```
local B in
  X<:Y=B   X<:Z=B
end
```

and creates 2 propagators for reified constraints.

Boolean connectives We can define the Boolean connectives (e.g., conjunction or negation) by associating 0 with false and 1 with true. The respective Boolean constraints can be posted by means of the following procedures:

- `{FD.conj X Y Z}` posts the constraint $(X \wedge Y) = Z$.
- `{FD.disj X Y Z}` posts the constraint $(X \vee Y) = Z$.
- `{FD.impl X Y Z}` posts the constraint $(X \rightarrow Y) = Z$.
- `{FD.equi X Y Z}` posts the constraint $(X \leftrightarrow Y) = Z$.
- `{FD.nega X Y}` posts the constraint $\neg X = Y$.

Exercises

Exercise 8.1 Write a statement that posts the constraint

$$(X < Y \rightarrow X + Y = Z) \leftrightarrow (X \cdot Y = Z \vee Z \neq 5)$$

Exercise 8.2 Write a procedure `{Conj x y z}` that posts the constraints

$$(X \wedge Y) = Z, \quad X \in 0\#1, \quad Y \in 0\#1$$

The procedure should post the conjunction $(X \wedge Y) = Z$. by means of the reified form of the infix operator `=:`.

Write analogous procedures `Equi` and `Nega` posting equivalences and negations.

Write an analogous procedure `Dis` posting a disjunction $(X \vee Y) = Z$. Use the reified form of `<:` to post the disjunction.

How would you write a procedure posting an implication $(X \rightarrow Y) = Z$?

8.2 Example: Aligning for a Photo

We will now see an overconstrained problem for which it is impossible to satisfy all constraints. The problem specification will distinguish between primary and secondary constraints, and the goal is to find a solution that satisfies all primary constraints and as many of the secondary constraints as possible.

Problem Specification

Betty, Chris, Donald, Fred, Gary, Mary, and Paul want to align in one row for taking a photo. Some of them have preferences next to whom they want to stand:

1. Betty wants to stand next to Gary and Mary.
2. Chris wants to stand next to Betty and Gary.
3. Fred wants to stand next to Mary and Donald.
4. Paul wants to stand next to Fred and Donald.

Obviously, it is impossible to satisfy all preferences. Can you find an alignment that maximizes the number of satisfied preferences?

Model

The model has a variable A_p for every person, where A_p stands for the position p takes in the alignment. Since there are exactly 7 persons, we have $A_p \in 1\#7$ for every person p . Moreover, we have $A_p \neq A_q$ for every pair p, q of distinct persons. The model has a variable $S_i \in 0\#1$ for each of the 8 preferences, where $S_i = 1$ if and only if the i -th preference is satisfied. To express this constraint, we constrain the control variable S

of a preference “person p wants to stand next to person q ” by means of the reified constraint

$$(|A_p - A_q| = 1 \leftrightarrow S = 1) \wedge S \in 0\#1$$

Finally, there is a variable

$$Satisfaction = S_1 + \dots + S_8$$

denoting the number of satisfied preferences. We want to find a solution that maximizes the value of *Satisfaction*.

The experienced constraint programmer will note that we can eliminate a symmetry by picking two persons p and q and imposing the order constraint $A_p < A_q$.

Distribution Strategy.

To maximize *Satisfaction*, we employ a two-dimensional distribution strategy, which first distributes on *Satisfaction*, trying the values 8, 7, ..., 1 in this order. Once *Satisfaction* is determined, we distribute on the variables A_p using a first-fail strategy that splits the domain of the selected variable.

Script.

Figure 8.1: A script for the Photo Puzzle.

```

proc {Photo Root}
  Persons      = [betty chris donald fred gary mary paul]
  Preferences  = [betty#gary betty#mary  chris#betty chris#gary
                  fred#mary  fred#donald paul#fred  paul#donald]
  NbPersons    = {Length Persons}
  Alignment    = {FD.record alignment Persons 1#NbPersons}
  Satisfaction = {FD.decl}
  proc {Satisfied P#Q S}
    {FD.reified.distance Alignment.P Alignment.Q '=' 1 S}
  end
in
  Root = r(satisfaction: Satisfaction
          alignment:    Alignment)
  {FD.distinct Alignment}
  {FD.sum {Map Preferences Satisfied} '=' Satisfaction}
  Alignment.fred <: Alignment.betty      % breaking symmetries
  {FD.distribute generic(order:naive value:max) [Satisfaction]}
  {FD.distribute split Alignment}
end

```

The script in Figure 8.1 constrains its root variable to a record consisting of the number of satisfied preferences and a record mapping the names of the persons to their positions in the alignment. The fields of the record *Alignment* implement the variables A_p of the model.

Satisfied The script introduces the defined constraint `{Satisfied P#Q S}`, which implements the reification of the constraint “P stands next to Q” with respect to S.

The statement

```
{FD.sum {Map Preferences Satisfied} '=: ' Satisfaction}
```

constrains the variable `Satisfaction` to the number of satisfied preferences.

The statement `{ExploreOne Photo}` will run the script until a first solution is found. The first solution satisfies 6 preferences and looks as follows:

```
6 # alignment(betty:5  chris:6  donald:1  fred:3
              gary:7   mary:4   paul:2)
```

By construction of the script, this is the maximal number of preferences that can be satisfied simultaneously.

Exercises

Exercise 8.3 *Modify the script such that the first solution minimizes the number of preferences satisfied.*

8.3 Example: Self-referential Aptitude Test

This example illustrates three issues: expressing complex propositional formulas as reified constraints, improving performance and presentation by elimination of common subconstraints, and using the symbolic constraint posted by `FD.element`.

Problem Specification

The self-referential aptitude test (which is taken from [10]) consists of 10 multiple choice questions, referred to as 1 to 10. Each question allows for 5 possible answers, referred to as a to e. For each of the 50 possible answers, a condition is specified. For each question, exactly one of the conditions associated with its possible answers must hold. A solution of the test is a function assigning to every question a letter such that the condition selected by the assigned letter holds. Here are the questions and their possible answers:

1. The first question whose answer is b is question (a) 2; (b) 3; (c) 4; (d) 5; (e) 6.
2. The only two consecutive questions with identical answers are questions (a) 2 and 3; (b) 3 and 4; (c) 4 and 5; (d) 5 and 6; (e) 6 and 7.
3. The answer to this question is the same as the answer to question (a) 1; (b) 2; (c) 4; (d) 7; (e) 6.
4. The number of questions with the answer a is (a) 0; (b) 1; (c) 2; (d) 3; (e) 4.
5. The answer to this question is the same as the answer to question (a) 10; (b) 9; (c) 8; (d) 7; (e) 6.

6. The number of questions with answer a equals the number of questions with answer (a) b; (b) c; (c) d; (d) e; (e) none of the above.
7. Alphabetically, the answer to this question and the answer to the following question are (a) 4 apart; (b) 3 apart; (c) 2 apart; (d) 1 apart; (e) the same.
8. The number of questions whose answers are vowels is (a) 2; (b) 3; (c) 4; (d) 5; (e) 6.
9. The number of questions whose answer is a consonant is (a) a prime; (b) a factorial; (c) a square; (d) a cube; (e) divisible by 5.
10. The answer to this question is (a) a; (b) b; (c) c; (d) d; (e) e.

To understand the test, verify that

| | | | | |
|-----|-----|-----|-----|------|
| 1:c | 2:d | 3:e | 4:b | 5:e |
| 6:e | 7:d | 8:c | 9:b | 10:a |

is a correct set of answers for the test. In particular, convince yourself that for every question the remaining 4 possibilities to answer it are falsified. The script we are going to write will prove that there is no other set of correct answers.

Model

Our model has 0/1-variables A_i , B_i , C_i , and D_i for $i \in 1\#10$ such that:

1. $A_i + B_i + C_i + D_i + E_i = 1$.
2. $A_i = 1$ iff the answer to Question i is a.
3. $B_i = 1$ iff the answer to Question i is b.
4. $C_i = 1$ iff the answer to Question i is c.
5. $D_i = 1$ iff the answer to Question i is d.
6. $E_i = 1$ iff the answer to Question i is e.

To obtain a compact representation of the questions, we also have variables $Q_i \in 1\#5$ for $i \in 1\#10$ such that

$$\begin{array}{ll} Q_i = 1 \leftrightarrow A_i = 1 & Q_i = 2 \leftrightarrow B_i = 1 \\ Q_i = 3 \leftrightarrow C_i = 1 & Q_i = 4 \leftrightarrow D_i = 1 \\ Q_i = 5 \leftrightarrow E_i = 1 & \end{array}$$

The first question can now be expressed by means of five equivalences:

$$\begin{array}{ll} A_1 & = B_2 \\ B_1 & = (B_3 \wedge (B_2 = 0)) \\ C_1 & = (B_4 \wedge (B_2 + B_3 = 0)) \\ D_1 & = (B_5 \wedge (B_2 + B_3 + B_4 = 0)) \\ E_1 & = (B_6 \wedge (B_2 + B_3 + B_4 + B_5 = 0)) \end{array}$$

These equivalences can be expressed by reifying the nested equality constraints.

The second question can be expressed with the following constraints:

$$\begin{aligned} Q_1 \neq Q_2, \quad Q_7 \neq Q_8, \quad Q_8 \neq Q_9, \quad Q_9 \neq Q_{10} \\ A_2 = (Q_2 = Q_3), \quad B_2 = (Q_3 = Q_4), \quad C_2 = (Q_4 = Q_5) \\ D_2 = (Q_5 = Q_6), \quad E_2 = (Q_6 = Q_7) \end{aligned}$$

The third question can be expressed as follows:

$$\begin{aligned} A_3 = (Q_1 = Q_3), \quad B_3 = (Q_2 = Q_3), \quad C_3 = (Q_4 = Q_3) \\ D_3 = (Q_7 = Q_3), \quad E_3 = (Q_6 = Q_3) \end{aligned}$$

The fourth question can be elegantly expressed with the constraint

$$element(Q_4, (0, 1, 2, 3, 4)) = \sum_{i=1}^{10} A_i$$

where the function $element(k, x)$ yields the k -th component of the tuple x .

We choose this formulation since Oz provides a propagator `FD.element` for the constraint $element(k, x) = y$.

reified membership constraints The ninth question can be expressed with the following equations

$$\begin{aligned} S &= \sum_{i=1}^{10} (B_i + C_i + D_i) \\ A_9 &= (S \in \{2, 3, 5, 7\}) \\ B_9 &= (S \in \{1, 2, 6\}) \\ C_9 &= (S \in \{0, 1, 4, 9\}) \\ D_9 &= (S \in \{0, 1, 8\}) \\ E_9 &= (S \in \{0, 5, 10\}) \end{aligned}$$

where S is an existentially quantified auxiliary variable. This time we use reified membership constraints of the form $x \in D$.

Distribution Strategy

We distribute on the variables $Q_1, Q_2 \dots$ using the standard first-fail strategy.

Script

elimination of common subconstraints The script in Figure 8.2 implements the indexed variables A_i, B_i, C_i, D_i, E_i , and Q_i as tuples with 10 components each. The three procedures `Vector`, `Sum`, and `Assert` make it more convenient to state the constraints. For each sum occurring in the questions an auxiliary variable is introduced so that the corresponding summation constraint needs to be posted only once. This elimination of common subconstraints provides for a compact formulation of the script and also improves its performance.

The procedure `{FD.element K V X}` posts a propagator for the constraint saying that X is the K -th component of the vector V .

Note the use of `FD.decl` in the definition of the procedure `Sum` and in the representation of the seventh question. Telling an initial domain constraint for the respective variables is necessary so that the propagators depending on these variables can start their work.

Figure 8.2: A script for the self-referential aptitude test.

```

proc {SRAT Q}
  proc {Vector V} % V is a 0/1-vector of length 10
    {FD.tuple v 10 0#1 V}
  end
  proc {Sum V S} % S is the sum of the components of vector V
    {FD.decl S} {FD.sum V '=' S}
  end
  proc {Assert I U V W X Y}
    A.I=U B.I=V C.I=W D.I=X E.I=Y
  end
  A = {Vector} B = {Vector}
  C = {Vector} D = {Vector} E = {Vector}
  SumA = {Sum A} SumB = {Sum B} SumC = {Sum C}
  SumD = {Sum D} SumE = {Sum E}
  SumAE = {Sum [SumA SumE]} SumBCD = {Sum [SumB SumC SumD]}
in
  {FD.tuple q 10 1#5 Q}
  {For 1 10 1
    proc {$ I} {Assert I Q.I=:1 Q.I=:2 Q.I=:3 Q.I=:4 Q.I=:5} end}
  %% 1
  {Assert 1 B.2
    {FD.conj B.3 (B.2=:0)}
    {FD.conj B.4 (B.2+B.3=:0)}
    {FD.conj B.5 (B.2+B.3+B.4=:0)}
    {FD.conj B.6 (B.2+B.3+B.4+B.5=:0)}}
  %% 2
  {Assert 2 Q.2=:Q.3 Q.3=:Q.4 Q.4=:Q.5 Q.5=:Q.6 Q.6=:Q.7}
  Q.1\=:Q.2 Q.7\=:Q.8 Q.8\=:Q.9 Q.9\=:Q.10
  %% 3
  {Assert 3 Q.1=:Q.3 Q.2=:Q.3 Q.4=:Q.3 Q.7=:Q.3 Q.6=:Q.3}
  %% 4
  {FD.element Q.4 [0 1 2 3 4] SumA}
  %% 5
  {Assert 5 Q.10=:Q.5 Q.9=:Q.5 Q.8=:Q.5 Q.7=:Q.5 Q.6=:Q.5}
  %% 6
  {Assert 6 SumA=:SumB SumA=:SumC SumA=:SumD SumA=:SumE _}
  %% 7
  {FD.element Q.7 [4 3 2 1 0] {FD.decl}={FD.distance Q.7 Q.8 '='}}
  %% 8
  {FD.element Q.8 [2 3 4 5 6] SumAE}
  %% 9
  {Assert 9 SumBCD::[2 3 5 7] SumBCD::[1 2 6]
    SumBCD::[0 1 4 9] SumBCD::[0 1 8]
    SumBCD::[0 5 10]}
  %% 10
  {FD.distribute ff Q}
end

```

Exercises

Exercise 8.4 *The script in Figure 8.2 uses the statement*

```
{FD.element Q.7 [4 3 2 1 0]
 {FD.decl}={FD.distance Q.7 Q.8 '=: '}}
```

to post the constraints for the seventh question. It avoids one auxiliary variable by nesting two equated procedure applications. Give an equivalent statement in which the auxiliary variable is introduced and the nested procedure applications are unfolded.

8.4 Example: Bin Packing

This example features a nontrivial model involving reified constraints, a three-dimensional distribution strategy optimizing a cost function, and nontrivial defined constraints. The script will employ explicit thread creations to prevent blocking. To optimize performance, the script will implement certain implicative constraints with conditionals rather than reified constraints.

Problem Specification

Given a supply of components and bins of different types, compile a packing lists such that a minimal number of bins is used and given constraints on the contents of bins are satisfied.

In our example, there are 3 types of bins and 5 types of components. The bin types are red, blue, and green. The component types are glass, plastic, steel, wood, and copper.

The following constraints must hold for the contents of bins:

1. Capacity constraints:
 - (a) Red bins can take at most 3 components, and at most 1 component of type wood.
 - (b) Blue bins can take exactly 1 component.
 - (c) Green bins can take at most 4 components, and at most 2 components of type wood.
2. Containment constraints (what can go into what):
 - (a) Red bins can contain glass, wood, and copper.
 - (b) Blue bins can contain glass, steel, and copper.
 - (c) Green bins can contain plastic, wood, and copper.
3. Requirement and exclusion constraints applying to all bin types:
 - (a) Wood requires plastic.
 - (b) Glass excludes copper.
 - (c) Copper excludes plastic.

Compile a packing list for an order consisting of 1 glass component, 2 plastic components, 1 steel component, 3 wood components, and 2 copper components. The packing list should require as few bins as possible.

Model

One possibility for a model consists in having a variable for every component saying in which bin the component should be packed. The resulting model admits many symmetric solutions and does not lead to a satisfactory script.

We will use a dual model that has variables for bins but not for components. The model has a variable *NbBins* saying how many bins are used to pack the order. The individual bins are then referred to as first, second, and so on bin. For every $i \in 1\#NbBins$ we have 6 variables:

- $Type_i$ denoting the type of the i -th bin.
- $Glass_i$ denoting the number of glass components to be packed into the i -th bin.
- $Plastic_i$ denoting the number of plastic components to be packed into the i -th bin.
- $Steel_i$ denoting the number of steel components to be packed into the i -th bin.
- $Wood_i$ denoting the number of wood components to be packed into the i -th bin.
- $Copper_i$ denoting the number of copper components to be packed into the i -th bin.

Given these variables, the capacity and containment constraints are easy to express. The requirement and exclusion constraints are implications that can be expressed by means of reified constraints.

To reduce the size of the search tree, we exclude some of the symmetries in a packing list. We require that blue bins appear before red bins, and red bins appear before green bins. Moreover, if two consecutive bins have the same type, the first bin must contain at least as many glass components as the second bin.

Distribution Strategy

We will use a three-dimensional distribution strategy. First we distribute on *NbBins*, trying smaller values first. Then we distribute on the type variables $Type_1, Type_2, \dots$ with a naive strategy trying the values blue, red and green in this order. Finally, after the number and types of bins are determined, we distribute on the capacity variables

$$Glass_1, Plastic_1, Steel_1, Wood_1, Copper_1, Glass_2, Plastic_2, \dots$$

with the standard first-fail strategy.

Script

The script is shown in Figure 8.3. It takes as parameter the order for which a packing list is to be compiled. The statement

```
{Browse
  {SearchOne
    {BinPacking
      order(glass:2 plastic:4 steel:3 wood:6 copper:4)}}}
```

Figure 8.3: A script for the Bin Packing Problem.

```

fun {BinPacking Order}
  ComponentTypes = [glass plastic steel wood copper]
  MaxBinCapacity = 4
  <Definition of IsBin 62a>
  <Definition of IsPackList 63a>
  <Definition of Match 63b>
  <Definition of Distribute 64a>
in
  proc {$ PackList}
    {IsPackList PackList}
    {Match PackList Order}
    {Distribute PackList}
  end
end

```

will compute a packing list for the order that was given in the problem specification:

```

[ b(copper:0 glass:0 plastic:0 steel:1 type:0 wood:0)
  b(copper:0 glass:0 plastic:0 steel:1 type:0 wood:0)
  b(copper:0 glass:0 plastic:0 steel:1 type:0 wood:0)
  b(copper:0 glass:2 plastic:0 steel:0 type:1 wood:0)
  b(copper:4 glass:0 plastic:0 steel:0 type:2 wood:0)
  b(copper:0 glass:0 plastic:1 steel:0 type:2 wood:2)
  b(copper:0 glass:0 plastic:1 steel:0 type:2 wood:2)
  b(copper:0 glass:0 plastic:2 steel:0 type:2 wood:2) ]

```

From the printout we can see that the script represents a packing list as a list of packed bins. The types of the bins are coded as numbers, where 0 is blue, 1 is red, and 2 is green. The packed bin

```
b(copper:0 glass:0 plastic:1 steel:0 type:2 wood:2)
```

has type green and contains 1 plastic and 2 wood components.

The procedure {BinPacking Order} introduces three defined constraints `IsBin`, `IsPackList`, and `Match`. It also defines a procedure `Distribute` implementing the distribution strategy. Given these procedures, the script itself is straightforward.

IsBin The definition of the procedure {IsBin Bin} appears in Figure 8.4. It imposes constraints saying that `Bin` is a consistently packed bin. In fact, the procedure {IsBin Bin} implements all the capacity, containment, requirement, and exclusion constraints of the problem specification. The thread creation at the end of the procedure is needed so that the conditional does not block on the determination of `Type`.

Figure 8.4: The defined constraint `IsBin`.62a **<Definition of IsBin 62a>**≡

```

proc {IsBin Bin}
  [Blue Red Green] = [0 1 2]
  BinTypes         = [Blue Red Green]
  Capacity         = {FD.int [1 3 4]}      % capacity of Bin
  Type             = {FD.int BinTypes}     % type of Bin
  Components
  [Glass Plastic Steel Wood Copper] = Components
in
  Bin = b(type:Type    glass:Glass  plastic:Plastic
          steel:Steel  wood:Wood    copper:Copper)
  Components ::: 0#MaxBinCapacity
  {FD.sum Components '=<:' Capacity}
  {FD.impl Wood>:0 Plastic>:0 1}    % wood requires plastic
  {FD.impl Glass>:0 Copper=:0 1}    % glass excludes copper
  {FD.impl Copper>:0 Plastic=:0 1}  % copper excludes plastic
  thread
    case Type
    of !Red then Capacity=3 Plastic=0 Steel=0 Wood=<:1
    [] !Blue then Capacity=1 Plastic=0 Wood=0
    [] !Green then Capacity=4 Glass = 0 Steel=0 Wood=<:2
    end
  end
end
end

```

implementing implicative constraints with conditionals The conditional implements three implicative constraints. Implementing these implicative constraints with reified constraints would be much more expensive. For instance, the statement implementing the first implicative constraint would take the form

```

{FD.impl Type=:Red
 ((Capacity=:3) + (Plastic=:0)
  + (Steel=:0) + (Wood=<:1) =: 4)
 1}

```

and thus create 7 propagators. In contrast, the implementation of all three implicative constraints with a single conditional creates at most one propagator.

The reified implementation `{FD.impl A B 1}` of an implication $A \rightarrow B$ yields stronger propagation than the conditional implementation

```

if A=:1 then B=1 else B=0 end

```

since it will tell `A=0` once `B=0` is known. Given our distribution strategy, the backward propagation would not have much effect in our example.

IsPackList The procedure `{IsPackList Xs}` (see Figure 8.5) imposes constraints saying that `Xs` is a consistent packing list ordered as specified in the description of the model. The thread creation prevents `IsPackList` from blocking on the determination of the list structure of `Xs`.

Figure 8.5: The defined constraint `IsPackList` for the Bin Packing Problem.

63a **<Definition of IsPackList 63a>**≡

```

proc {IsPackList Xs}
  thread
    {ForAll Xs IsBin}
    {ForAllTail Xs % impose order
      proc {$ Ys}
        case Ys of A|B|_ then
          A.type =: B.type
          {FD.impl A.type=:B.type A.glass>=:B.glass 1}
        else skip
        end
      end}
    end
  end
end

```

Match The procedure `{Match PackList Order}` (see Figure 8.6) imposes constraints saying that the packing list `PackList` matches the order `Order`. Once more a thread creation is needed to prevent `Match` from blocking on the determination of the list structure of `PackList`.

Figure 8.6: The defined constraint `Match` for the Bin Packing Problem.

63b **<Definition of Match 63b>**≡

```

proc {Match PackList Order}
  thread
    {ForAll ComponentTypes
      proc {$ C}
        {FD.sum {Map PackList fun {$ B} B.C end} '=: ' Order.C}
      end}
    end
  end
end

```

Distribute The procedure `{Distribute PackList}` implements the distribution strategy (see Figure 8.7). It first computes a lower bound `min` for `NbBins` and then distributes naively on `NbBins`. After `NbBins` is determined, the variables `Types` and `Capacities` are constrained to the respective lists. Then the script first distributes on `Types` and afterwards on `Capacities`.

Figure 8.7: A distributor for the Bin Packing Problem.

64a **<Definition of Distribute 64a>**≡

```

proc {Distribute PackList}
  NbComps = {Record.foldR Order Number.'+' 0}
  Div     = NbComps div MaxBinCapacity
  Mod     = NbComps mod MaxBinCapacity
  Min     = if Mod==0 then Div else Div+1 end
  NbBins  = {FD.int Min#NbComps}
  Types
  Capacities

in
  {FD.distribute naive [NbBins]}
  PackList  = {MakeList NbBins} % blocks until NbBins is determined
  Types     = {Map PackList fun {$ B} B.type end}
  Capacities = {FoldR PackList
                fun {$ Bin Cs}
                  {FoldR ComponentTypes fun {$ T Ds} Bin.T|Ds end Cs}
                end
                nil}
  {FD.distribute naive Types}
  {FD.distribute ff Capacities}

end

```

Exercises

Exercise 8.5 The procedure {IsPackList} employs the statement

```
{FD.impl A.type=:B.type A.glass>=:B.glass 1}
```

to post an implicative constraint. This will create 3 propagators. Implement the implicative constraint with a conditional that creates only 1 propagator.

User-Defined Distributors

In this section we show how the user can program his or her own distributors.

9.1 A Naive Distribution Strategy

The distributor we program in this section implements a naive distribution strategy: choose the first not yet determined variable from a list and try the smallest possible value first. The distributor is shown in Figure 9.1.

Figure 9.1: A distributor for a naive distribution strategy.

```

proc {NaiveDistributor Is}
  {Space.waitStable}
  local
    Fs={Filter Is fun {$ I} {FD.reflect.size I}>1 end}
  in
    case Fs
    of nil then skip
    [] F|Fr then M={FD.reflect.min F} in
      choice F=M {NaiveDistributor Fr}
      []      F\=:M {NaiveDistributor Fs}
      end
    end
  end
end
end

```

choice-statements To maximize the information available for distribution we wait until the computation space becomes stable. A thread that executes `{Space.waitStable}` blocks until its hosting computation space S becomes stable. When S becomes stable, execution proceeds with the next statement.

Thus, the variable `Fs` in Figure 9.1 denotes the list of undetermined variables after S has become stable. To detect undetermined variables we use the procedure `FD.reflect.size` that returns the current size of a variable's domain. If the domain size is one, the variable is determined and consequently not included in the list `Fs`.

Then the least possible value for the first undetermined variable `F` is computed by

```
M={FD.reflect.min I}
```

binary choice-statements We now have to distribute. To this aim Oz provides a binary choice-statement. If a thread reaches the statement

```
choice S1
[] S2
end
```

the thread is blocked until its hosting computation space becomes stable.

If the space has become stable, the computation in the blocked thread is resumed and it is distributed. Distribution yields two spaces, one obtained by replacing the choice-statement by the statement `S1`, one obtained by replacing the choice-statement by the statement `S2`. All search engines in this tutorial will explore the space first which hosts `S1`.

In Figure 9.1, we distribute with the constraint that the selected variable is determined to the current least possible value. The distribution is done if no undetermined variables are left.

9.2 A Domain-Splitting Distributor

In this section we program a distributor for the domain-splitting strategy (see item 2.8 (page 12)). The program is shown in Figure 9.2. As in the previous section we first discard all determined variables. Then we select the variable `MinVar` which has the smallest domain (as it is done for the first-fail distribution strategy). For the selected variable we determine the value that is in the middle of the least and largest possible value by

```
Mid = {FD.reflect.mid MinVar}
```

After this is done we distribute with the constraint that `MinVar` should be smaller than or equal to `Mid`.

Figure 9.2: A distributor for a domain-splitting strategy.

```

proc {SplitDistributor Is}
  {Space.waitStable}
  local
    Fs={Filter Is fun {$ I} {FD.reflect.size I}>1 end}
  in
    case Fs
    of nil then skip
    [] F|Fr then
      MinVar#_ = {FoldL Fr fun {$ Var#Size X}
                    if {FD.reflect.size X}<Size then
                      X#{FD.reflect.size X}
                    else
                      Var#Size
                    end
                    end F#{FD.reflect.size F}}
      Mid = {FD.reflect.mid MinVar}
    in
      choice MinVar <: Mid then {SplitDistributor Fs}
      []      MinVar >: Mid  then {SplitDistributor Fs}
      end
    end
  end
end
end

```

Branch and Bound

In this chapter we focus on computing an optimal solution according to a given cost function. While we have searched for optimal solutions already in Section 8.2 and Section 8.4, we have used a rather ad-hoc strategy there. This strategy lacks generality and does not provide either for a proof of optimality.

branch and bound In this chapter we introduce a general schema to compute an optimal solution according to an arbitrary cost function and show how it is available in Oz. This schema is called branch and bound and is available by procedures like `ExploreBest` (see “Oz Explorer – Visual Constraint Programming Support” and Chapter *Search Engines: Search, (System Modules)* for more search engines). A typical application of `ExploreBest` for a script `Script` looks like

```
{ExploreBest Script Order}
```

The branch and bound schema works as follows. When a solution of `Script` is found, all the remaining alternatives in the search tree are constrained to be better with respect to an order available through the procedure `Order`. Usually `Order` applies a cost function to its arguments and creates a propagator imposing the ordering. The first argument of `Order` is the previous solution, and the second argument is an alternative solution we are searching for.

10.1 Example: Aligning for a Photo, Revisited

In Section 8.2 we have searched for a solution of the alignment problem which satisfies the maximal number of preferences. To this aim we have introduced a variable which counts the number of satisfied preferences. By distributing this variable with its maximal value first, we have guaranteed that the first solution found is indeed the optimal one.

In this section we replace this ad-hoc approach by branch and bound. We first restate the script for the problem by omitting the distribution code for the variable summing up the satisfied preferences. The resulting script is shown in Figure 10.1.

Next we define an ordering procedure which states that the overall sum of satisfied preferences in an alternative solution must be strictly greater than the corresponding sum in a previous solution:

Figure 10.1: The revised script for the Photo Puzzle.

```

proc {RevisedPhoto Root}
  Persons      = [betty chris donald fred gary mary paul]
  Preferences   = [betty#gary betty#mary  chris#betty chris#gary
                  fred#mary  fred#donald paul#fred  paul#donald]
  NbPersons     = {Length Persons}
  Alignment     = {FD.record alignment Persons 1#NbPersons}
  Satisfaction  = {FD.decl}
  proc {Satisfied P#Q S}
    {FD.reified.distance Alignment.P Alignment.Q '=' 1 S}
  end
in
  Root = r(satisfaction: Satisfaction
          alignment:    Alignment)
  {FD.distinct Alignment}
  {FD.sum {Map Preferences Satisfied} '=' Satisfaction}
  Alignment.fred <: Alignment.betty      % breaking symmetries
  {FD.distribute split Alignment}
end

```

```

proc {PhotoOrder Old New}
  Old.satisfaction <: New.satisfaction
end

```

The optimal solution can be found with the statement

```
{ExploreBest RevisedPhoto PhotoOrder}
```

We obtain the same solution as in (page 55). But now only 141 choice nodes are needed to find the optimal solution whereas 219 choice nodes were needed in Section 8.2. Furthermore, branch and bound allows us to prove in an efficient way that the last solution found is really the optimal one. The full search tree (including the proof of optimality) contains 169 choice nodes; still less than for the search for the best solution with the ad-hoc method. If we inspect the solutions, we observe that the first solution satisfies only a single preference. By imposition of the ordering procedure by the search engine, the next found solution must satisfy more preferences. Indeed, the second solution satisfies two preferences. Following this approach we finally arrive at the optimal solution with six satisfied preferences.

10.2 Example: Locating Warehouses

This example features branch and bound to compute an optimal solution, a non-trivial distribution strategy and symbolic constraints.

Problem Specification

Assume a company which wants to construct warehouses to supply stores with goods. Each warehouse to be constructed would have a certain capacity defining the largest number of stores which can be supplied by this warehouse. For the construction of a warehouse we have fixed costs. The costs for transportation from a warehouse to a store vary depending on the location of the warehouse and the supplied store. The aim is to determine which warehouses should be constructed and which stores should be supplied by the constructed warehouses such that the overall costs are minimized.

We assume the fixed costs of building a warehouse to be 50. We furthermore assume 5 warehouses W_1 through W_5 and 10 stores $Store_1$ through $Store_{10}$. The capacities of the warehouses are shown in Figure 10.2. The costs to supply a store by a warehouse are shown in Figure 10.3.

Figure 10.2: Capacities of warehouses.

| | W_1 | W_2 | W_3 | W_4 | W_5 |
|----------|-------|-------|-------|-------|-------|
| Capacity | 1 | 4 | 2 | 1 | 3 |

Figure 10.3: Costs for supplying stores.

| | W_1 | W_2 | W_3 | W_4 | W_5 |
|--------------|-------|-------|-------|-------|-------|
| $Store_1$ | 36 | 42 | 22 | 44 | 52 |
| $Store_2$ | 49 | 47 | 134 | 135 | 121 |
| $Store_3$ | 121 | 158 | 117 | 156 | 115 |
| $Store_4$ | 8 | 91 | 120 | 113 | 101 |
| $Store_5$ | 77 | 156 | 98 | 135 | 11 |
| $Store_6$ | 71 | 39 | 50 | 110 | 98 |
| $Store_7$ | 6 | 12 | 120 | 98 | 93 |
| $Store_8$ | 20 | 120 | 25 | 72 | 156 |
| $Store_9$ | 151 | 60 | 104 | 139 | 77 |
| $Store_{10}$ | 79 | 107 | 91 | 117 | 154 |

Model

We assume that the costs are given in the matrix *CostMatrix* defined by Figure 10.3. For the model of this problem we introduce the following variables.

- $Open_i, 1 \leq i \leq 5$, with domain $\{0,1\}$ such that $Open_i = 1$ if warehouse W_i does supply at least one store.
- $Supplier_i, 1 \leq i \leq 10$, with domain $\{1, \dots, 5\}$ such that $Supplier_i = j$ if store $Store_i$ is supplied by warehouse W_j .

- $Cost_i, 1 \leq i \leq 10$, such that the domain of $Cost_i$ is defined by the row $CostMatrix_i$. The variable $Cost_i$ denotes the costs of supplying store $Store_i$ by warehouse $W_{Supplier_i}$, i.e., $Cost_i = CostMatrix_{i,Supplier_i}$.

We have the additional constraint that the capacity of the warehouses must not be exceeded. To this aim we introduce auxiliary variables $Supplies_{i,j}$ with the domain 0#1 as follows.

$$\forall i \in \{1, \dots, 5\} \forall j \in \{1, \dots, 10\} : (Supplies_{i,j} = 1) \leftrightarrow (Supplier_j = i)$$

The capacity constraints can then be stated with

$$\forall i \in \{1, \dots, 5\} : Cap_i \geq \sum_{j=1}^{10} Supplies_{i,j}$$

where Cap_i is defined according to Figure 10.2.

Distribution Strategy

There are several possibilities to define a distribution strategy for this problem.

least regret We choose to determine the variables $Cost_i$ by distribution. Because no entry in a row of the cost matrix occurs twice, we immediately know which store is supplied by which warehouse. We select the variable $Cost_i$ first for which the difference between the smallest possible value and the next higher value is maximal. Thus, decisions are made early in the search tree where the difference between two costs by different suppliers are maximal. The distribution strategy will try the minimal value in the domain of $Cost_i$ first. In Operations Research this strategy is known as the principle of least regret.

Script

The script in Figure 10.4 constrains its root variable to a record containing the supplying warehouse for each store, the costs for each store to be supplied by the corresponding warehouse and the total costs.

The statement

```
{FD.element Supplier.St CostMatrix.St Cost.St}
```

connects the costs to supply a store with the supplier. Because no element in a row of the cost matrix occurs twice, the supplier for a store is known if its costs are determined and vice versa. Through this statement the constraint $Cost_i = CostMatrix_{i,Supplier_i}$ is imposed.

A propagator for the constraint that the capacity of a warehouse is not exceeded can be created by the statement

```
{FD.atMost Capacity.S Supplier S}
```

The statement

```
Open.S = {FD.reified.sum {Map Stores fun {$ St}
    Supplier.St =: S end} '>:' 0}
```

guarantees that a variable $Open_i$ in the model is constrained to 1 if warehouse W_i supplies at least one store.

The first solution of the problem can be found with the statement

```
{ExploreOne WareHouse}
```

To compute the solution with minimal costs we define the following ordering relation.

```
proc {Order Old New}
    Old.totalCost >: New.totalCost
end
```

The optimal solution with total cost 604 can now be computed with

```
{ExploreBest WareHouse Order}
```

Figure 10.4: A script for the warehouse problem.

```

Capacity    = supplier(3 1 4 1 4)
CostMatrix  = store(supplier(36 42 22 44 52)
                    supplier(49 47 134 135 121)
                    supplier(121 158 117 156 115)
                    supplier(8 91 120 113 101)
                    supplier(77 156 98 135 11)
                    supplier(71 39 50 110 98)
                    supplier(6 12 120 98 93)
                    supplier(20 120 25 72 156)
                    supplier(151 60 104 139 77)
                    supplier(79 107 91 117 154))

BuildingCost = 50
fun {Regret X}
  M = {FD.reflect.min X}
in
  {FD.reflect.nextLarger X M} - M
end
proc {WareHouse X}
  NbSuppliers = {Width Capacity}
  NbStores    = {Width CostMatrix}
  Stores      = {List.number 1 NbStores 1}
  Supplier    = {FD.tuple store NbStores 1#NbSuppliers}
  Open        = {FD.tuple supplier NbSuppliers 0#1}
  Cost        = {FD.tuple store NbStores 0#FD.sup}
  SumCost     = {FD.decl} = {FD.sum Cost '='}
  NbOpen      = {FD.decl} = {FD.sum Open '='}
  TotalCost   = {FD.decl}
in
  X = plan(supplier:Supplier cost:Cost totalCost:TotalCost)
  TotalCost =: SumCost + NbOpen*BuildingCost
  {For 1 NbStores 1
    proc {$ St}
      Cost.St :: {Record.toList CostMatrix.St}
      {FD.element Supplier.St CostMatrix.St Cost.St}
    end}
  {For 1 NbSuppliers 1
    proc {$ S}
      {FD.atMost Capacity.S Supplier S}
      Open.S = {FD.reified.sum {Map Stores fun {$ St}
                                Supplier.St =: S
                                end} '>:' 0}
    end}
  {FD.distribute
    generic(order: fun {$ X Y} {Regret X} > {Regret Y} end)
    Cost}
  end

```

Scheduling

In this section we will consider examples of scheduling problems. Scheduling in this tutorial means to compute a timetable for tasks competing for a given set of resources. We assume that the execution of a task can not be interrupted (that is, no preemption is allowed).

11.1 Building a House

We first consider the problem to build a house (in a simplified way). We will successively refine the problem specification, the model and the distribution strategy in order to solve more and more demanding problems.

Problem Specification

The task names, their description, duration (in days) and the company in charge are given in Figure 11.1. For example, **b** denotes the task involved with the carpentry for the roof. This task lasts for 3 days. Task **a** must be finished before the work for task **b** is started (indicated by the column *Predecessor*). The company in charge for task **b** is *House Inc.* The overall goal is to build the house as quickly as possible.

Figure 11.1: Building a house.

| Task | Description | Duration | Predecessor | Company |
|------|--------------------|----------|-------------|-------------------|
| a | Erecting Walls | 7 | none | Construction Inc. |
| b | Carpentry for Roof | 3 | a | House Inc. |
| c | Roof | 1 | b | House Inc. |
| d | Installations | 8 | a | Construction Inc. |
| e | Facade Painting | 2 | c, d | Construction Inc. |
| f | Windows | 1 | c, d | House Inc. |
| g | Garden | 1 | c, d | House Inc. |
| h | Ceilings | 3 | a | Construction Inc. |
| i | Painting | 2 | f, h | Builder Corp. |
| j | Moving in | 1 | i | Builder Corp. |

11.1.1 Building a House: Precedence Constraints

For the first model we do not consider the companies in charge for the tasks.

Model

The model introduces for each task a variable which stands for the start time of the task. In the sequel we will identify a task and its corresponding variable. The end time of each task is its start time plus its duration. For the time origin we assume 0. A trivial upper bound for the time to build the house can be obtained by summing up all durations of tasks. Here, we obtain 29.

precedence constraints From the predecessor relation we can derive a set of so-called precedence constraints:

$$\begin{array}{llll} A + 7 \leq B, & B + 3 \leq C, & A + 7 \leq D, & C + 1 \leq E, \\ D + 8 \leq E, & C + 1 \leq F, & D + 8 \leq F, & C + 1 \leq G, \\ D + 8 \leq G, & A + 7 \leq H, & F + 1 \leq I, & H + 3 \leq I, \\ & & I + 2 \leq J. & \end{array}$$

makespan For example, the constraint $A + 7 \leq B$ means that the earliest start time of `b` is 7 days after `a` has been started. We assume an additional task `pe` modeling the project end for the problem. All other tasks precede `pe`. The start time of `pe` is called the makespan of the schedule.

Distribution Strategy

If all propagators have become stable, it is sufficient to determine each variable to the current minimal value in its domain to obtain a solution. This is due to the fact that we only use constraints of the form $x + c \leq y$ where c is an integer. Hence, we do not need a distributor at all. Note that this fact remains true if we also consider constraints of the form $x + c = y$ (this will be needed later).

Script

The problem specification which is a direct implementation of Figure 11.1 is given in Figure 11.2. The field under the feature `tasks` contains the specification as a list of records. The label of each record gives the task name, the field at feature `dur` the duration, the field at feature `pre` the list of preceding tasks, and the field at feature `res` the resource name. The features `pre` and `res` are optional, if they are missing no preceding tasks and no resource are required. The task with name `pe` denotes the additional task representing the project end.

scheduling compiler Figure 11.3 shows a procedure that returns a script according to our scheduling specification. The used procedures `GetDur` and `GetStart` are shown in Figure 11.4. Such a procedure is called a scheduling compiler because it processes the problem specification and returns a script. Hence, the scheduling compiler *compiles* the problem specification into an *executable script*.

Figure 11.2: The specification to build a house.

```

House = house(tasks: [a(dur:7          res:constructionInc)
                      b(dur:3  pre:[a]   res:houseInc)
                      c(dur:1  pre:[b]   res:houseInc)
                      d(dur:8  pre:[a]   res:constructionInc)
                      e(dur:2  pre:[c d]  res:constructionInc)
                      f(dur:1  pre:[c d]  res:houseInc)
                      g(dur:1  pre:[c d]  res:houseInc)
                      h(dur:3  pre:[a]   res:constructionInc)
                      i(dur:2  pre:[f h]  res:builderCorp)
                      j(dur:1  pre:[i]   res:builderCorp)
                      pe(dur:0 pre:[j])])

```

Figure 11.3: Scheduling compiler.

```

fun {Compile Spec}
  TaskSpec = Spec.tasks
  Dur      = {GetDur TaskSpec}
in
  proc {$ Start}
    Start = {GetStart TaskSpec}
    <Post precedence constraints 78a>
    <Assign start times 78b>
  end
end

```

Figure 11.4: Procedures to compute duration and start records.

```

fun {GetDur TaskSpec}
  {List.toRecord dur {Map TaskSpec fun {$ T}
                                     {Label T}#T.dur
                                   end}}
end
fun {GetStart TaskSpec}
  MaxTime = {FoldL TaskSpec fun {$ Time T}
                                     Time+T.dur
                                   end 0}
  Tasks   = {Map TaskSpec Label}
in
  {FD.record start Tasks 0#MaxTime}
end

```

The durations and start times of tasks are stored in the records `Dur` and `Start`, respectively. The record `Start` is the root variable of the script returned by the function `Compile`. First, the propagators for the precedence constraints are created, which is shown in Figure 11.5. After the space executing the scheduling script has become stable, the start times are determined. This is shown in Figure 11.6.

Figure 11.5: Posting precedence constraints.

78a **⟨Post precedence constraints 78a⟩**≡

```
{ForAll TaskSpec
  proc {$ T}
    {ForAll {CondSelect T pre nil}
      proc {$ P}
        Start.P + Dur.P <=: Start.{Label T}
      end}
    end}
```

Figure 11.6: Assigning start times.

78b **⟨Assign start times 78b⟩**≡

```
{FD.assign min Start}
```

The statement

```
{ExploreOne {Compile House}}
```

runs the script. The makespan of the schedule is 19. By construction this solution is the one with the smallest makespan.

11.1.2 Building a House: Capacity Constraints

In this section we take the companies into account which are in charge for the tasks. We assume that each company cannot handle two tasks simultaneously. That is, the execution of two tasks handled by the same company must not overlap in time.

Model

For each company (which we also call a resource because the companies are consumed by a task) we must find a serialization of the handled tasks, i.e. for each task pair A, B we must decide whether A is finished before B starts or vice versa. Assume two tasks with start times S_1 and S_2 and the durations D_1 and D_2 , respectively.

capacity constraints Then the constraint

$$S_1 + D_1 \leq S_2 \quad \vee \quad S_2 + D_2 \leq S_1$$

states that the corresponding tasks do not overlap in time. Such a constraint is also known as a capacity constraint, because the capacity of the resource must not be exceeded. The capacity constraints can be modeled by reified constraints for each pair of tasks handled by the same resource (company). But this leads to a number of propagators which increases quadratically in the number of tasks on a resource. This is not a feasible approach for problems with many tasks. Thus, we will use a single propagator in the script providing the same propagation as the quadratic number of reified constraints.

Distribution Strategy

Because of the capacity constraints we have to provide a distribution strategy. We use the standard first-fail strategy.

Script

We extend the scheduling compiler in Figure 11.3 to extract the tasks handled by a common resource. The procedure `GetTasksOnResource` takes a task specification and returns a record that maps resource names to tasks. Its implementation is shown in Figure 11.7.

Figure 11.7: Extracting tasks on the same resource.

```

fun {GetTasksOnResource TaskSpec}
  D={Dictionary.new}
in
  {ForAll TaskSpec
    proc {$ T}
      if {HasFeature T res} then R=T.res in
        {Dictionary.put D R {Label T} | {Dictionary.condGet D R nil}}
      end
    end}
  {Dictionary.toRecord tor D}
end

```

The modified scheduling compiler is shown in Figure 11.8. The returned script uses

```
{Schedule.serializedDisj TasksOnRes Start Dur}
```

to create for each resource a single propagator for the capacity constraints as described in the model above.

Exercise 11.1 Write a procedure which implements the capacity constraints of the problem by reified constraints.

Figure 11.8: A scheduling compiler with resource constraints.

```

fun {Compile Spec}
  TaskSpec    = Spec.tasks
  Dur         = {GetDur TaskSpec}
  TasksOnRes  = {GetTasksOnResource TaskSpec}
in
  proc {$ Start}
    Start = {GetStart TaskSpec}
    {Schedule.serializedDisj TasksOnRes Start Dur}
    <Post precedence constraints 78a>
    {FD.distribute ff Start}
  end
end

```

But we are not only interested in the first solution but in the best solution. For our problem we are interested in the solution with the smallest makespan.

For our example we define the order relation

```

proc {Earlier Old New}
  Old.pe >: New.pe
end

```

stating that the makespan of the new alternative solution must be strictly smaller than the makespan of the already found solution. We assume that the refined scheduling compiler is the procedure `CompileHouse2`. Thus, the best solution for our problem can be found by the following statement.

```
{ExploreBest {Compile House} Earlier}
```

The first solution which is also the optimal one has a makespan of 21.

11.1.3 Building a House: Serializers

So far we have used only distribution strategies where a variable is selected first and then the domain is further restricted by a basic constraint. Scheduling applications lead to distribution strategies where we distribute not only with basic constraints but with propagators.

serializers In the previous section we have seen that it is necessary to serialize all tasks on a common resource to satisfy all capacity constraints. This leads to the idea to use a distributor to serialize the tasks. Such a distributor is called a serializer. Thus, we refine the scheduling compiler of the previous section by formulating a new distribution strategy.

Note that we have to refine the notion of distribution here. In Section 2.6 we have distributed a finite domain problem P only with constraints C and $\neg C$. But we can

refine the concept of distribution by distributing with constraints C_1 and C_2 whenever $P \models C_1 \vee C_2$ holds.

By this condition we are sure that no solution is lost. For a serializer we distribute with constraints $S_1 + D_1 \leq S_2$ and $S_2 + D_2 \leq S_1$ where we assume two tasks with start times S_1 and S_2 and the durations D_1 and D_2 , respectively. In the presence of capacity constraints the required condition holds by construction, i.e. $P \models S_1 + D_1 \leq S_2 \vee S_2 + D_2 \leq S_1$.

Ordering Tasks by Distribution

We replace the first-fail distribution strategy by a strategy consisting of two phases. In the first phase we serialize all tasks on common resources and in the second phase we determine the start times of the variables. The serialization is achieved by distributing for each pair of tasks T_1 and T_2 either with the constraint that task T_1 is finished before T_2 starts or that task T_2 is finished before task T_1 starts.

ordering of tasks If such a distribution step takes place we say that the two concerned tasks are ordered. After the serialization we have only constraints of the form $x + c \leq y$. Thus, it is sufficient for the second phase to determine each variable to the smallest value in its domain.

Script

The script for the third version of our problem refines the one in the previous section by replacing the first fail distributor by a distributor that orders task and assigning minimal start times. The distributor that orders tasks on resources is defined as follows:

```
81a  <Order tasks 81a>≡
      {Record.forAll TasksOnRes
        proc {$ Ts}
          {ForAllTail Ts
            proc {$ T1|Tr}
              {ForAll Tr
                proc {$ T2}
                  choice Start.T1 + Dur.T1 <=: Start.T2
                  []      Start.T2 + Dur.T2 <=: Start.T1
                  end
                end}
              end}
        end}
```

The complete scheduling compiler can be found in Figure 11.9. The optimal solution can be found by

```
{ExploreBest {Compile House} Earlier}
```

with 28 choice nodes and 3 solution nodes. Thus, for this problem the use of a serializer results in a larger search tree than the first-fail distributor. In the following section we will tackle a harder problem and we will show that the first-fail strategy as well as the naive serializer of this section completely fail to compute the optimal solution of this more difficult problem.

Figure 11.9: A scheduling compiler with task ordering.

```

fun {Compile Spec}
  TaskSpec    = Spec.tasks
  Dur         = {GetDur TaskSpec}
  TasksOnRes  = {GetTasksOnResource TaskSpec}
in
  proc {$ Start}
    Start = {GetStart TaskSpec}
    <Post precedence constraints 78a>
    {Schedule.serializedDisj TasksOnRes Start Dur}
    <Order tasks 81a>
    <Assign start times 78b>
  end
end

```

11.2 Constructing a Bridge

The following problem is taken from [4] and is used as a benchmark in the constraint programming community. The problem is to schedule the construction of the bridge shown in Figure 11.10.

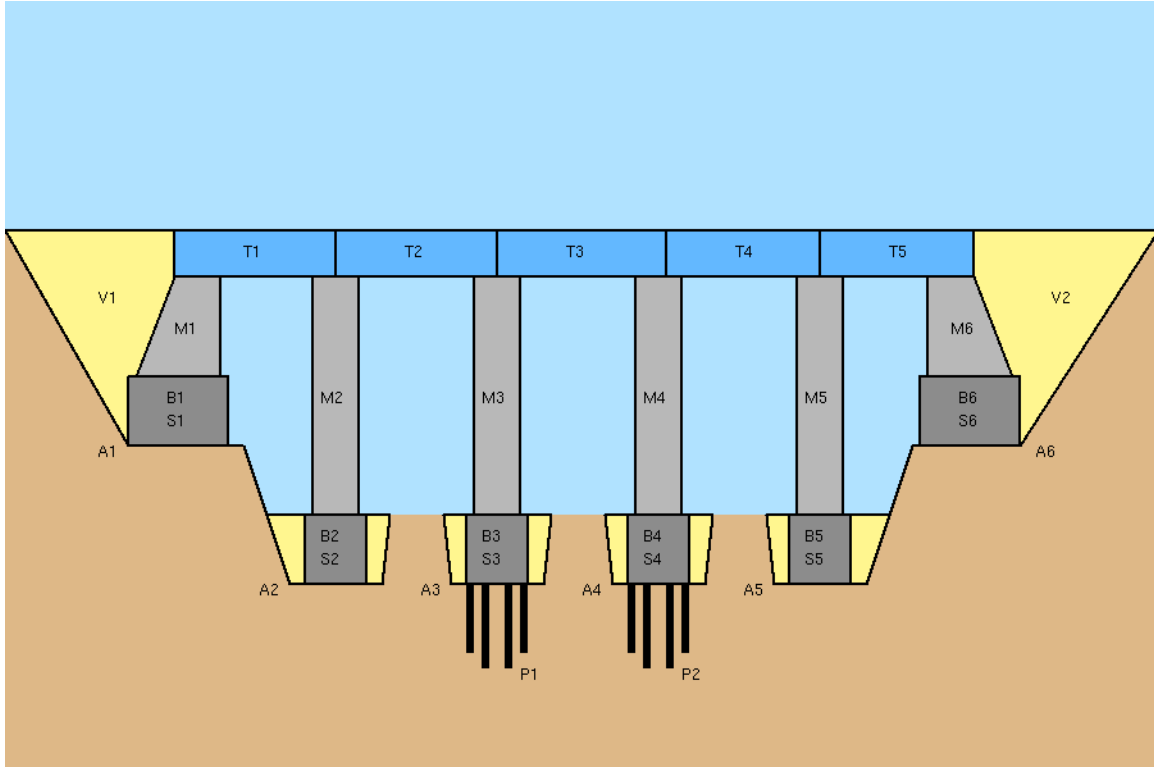
Problem Specification

The problem is specified as shown in Figure 11.11. From this table we derive precedence and capacity constraints as in the sections before. We also assume that a resource cannot handle more than one activity at a time. Such a kind of resource is also known as a *unary resource*.

unary resources Due to some peculiarities of the problem, we have the following additional constraints.

1. The time between the completion of the formwork and the completion of the corresponding concrete foundation is at most 4 days.
2. Between the end of a particular foundation and the beginning of the corresponding formwork can at most 3 days elapse.
3. The erection of the temporary housing must begin at least six days before each formwork.
4. The removal of the temporary housing can start at most two days before the end of the last masonry.
5. The delivery of the preformed bearers occurs exactly 30 days after the beginning of the project.

Figure 11.10: The Bridge Problem.



To deal with the additional constraints we refine the record containing the specification of the problem. We add a field under the feature `constraints` that contains a procedure parameterized by the records containing the start times and the durations of tasks (see Figure 11.12). This procedure will be applied by the scheduling script.

Model

A trivial upper bound of the makespan is the sum of all durations of the tasks. For the bridge construction problem we have 271 as the upper bound. We adopt the model of the house problem including capacity constraints. The additional constraints can be modeled with propagators for the following constraints over the problem variables ($\text{dur}(T)$ denotes the duration of a task T).

1.

$$(B_i + \text{dur}(B_i)) - (S_i + \text{dur}(S_i)) \leq 4, \quad 1 \leq i \leq 6$$

2.

$$S_i - (A_i + \text{dur}(A_i)) \leq 3, \quad i \in \{1, 2, 5, 6\}$$

$$S3 - (P1 + \text{dur}(P1)) \leq 3$$

$$S4 - (P2 + \text{dur}(P2)) \leq 3$$

Figure 11.11: Data for bridge construction.

| No | Na. | Description | Dur | Preds | Res |
|----|-----|------------------------------------|-----|------------------------|----------------|
| 1 | pa | beginning of project | 0 | - | noResource |
| 2 | a1 | excavation (abutment 1) | 4 | pa | excavator |
| 3 | a2 | excavation (pillar 1) | 2 | pa | excavator |
| 4 | a3 | excavation (pillar 2) | 2 | pa | excavator |
| 5 | a4 | excavation (pillar 3) | 2 | pa | excavator |
| 6 | a5 | excavation (pillar 4) | 2 | pa | excavator |
| 7 | a6 | excavation (abutment 2) | 5 | pa | excavator |
| 8 | p1 | foundation piles 2 | 20 | a3 | pile driver |
| 9 | p2 | foundation piles 3 | 13 | a4 | pile driver |
| 10 | ue | erection of temporary housing | 10 | pa | noResource |
| 11 | s1 | formwork (abutment 1) | 8 | a1 | carpentry |
| 12 | s2 | formwork (pillar 1) | 4 | a2 | carpentry |
| 13 | s3 | formwork (pillar 2) | 4 | p1 | carpentry |
| 14 | s4 | formwork (pillar 3) | 4 | p2 | carpentry |
| 15 | s5 | formwork (pillar 4) | 4 | a5 | carpentry |
| 16 | s6 | formwork (abutment 2) | 10 | a6 | carpentry |
| 17 | b1 | concrete foundation (abutment 1) | 1 | s1 | concrete mixer |
| 18 | b2 | concrete foundation (pillar 1) | 1 | s2 | concrete mixer |
| 19 | b3 | concrete foundation (pillar 2) | 1 | s3 | concrete mixer |
| 20 | b4 | concrete foundation (pillar 3) | 1 | s4 | concrete mixer |
| 21 | b5 | concrete foundation (pillar 4) | 1 | s5 | concrete mixer |
| 22 | b6 | concrete foundation (abutment 2) | 1 | s6 | concrete mixer |
| 23 | ab1 | concrete setting time (abutment 1) | 1 | b1 | noResource |
| 24 | ab2 | concrete setting time (pillar 1) | 1 | b2 | noResource |
| 25 | ab3 | concrete setting time (pillar 2) | 1 | b3 | noResource |
| 26 | ab4 | concrete setting time (pillar 3) | 1 | b4 | noResource |
| 27 | ab5 | concrete setting time (pillar 4) | 1 | b5 | noResource |
| 28 | ab6 | concrete setting time (abutment 2) | 1 | b6 | noResource |
| 29 | m1 | masonry work (abutment 1) | 16 | ab1 | bricklaying |
| 30 | m2 | masonry work (pillar 1) | 8 | ab2 | bricklaying |
| 31 | m3 | masonry work (pillar 2) | 8 | ab3 | bricklaying |
| 32 | m4 | masonry work (pillar 3) | 8 | ab4 | bricklaying |
| 33 | m5 | masonry work (pillar 4) | 8 | ab5 | bricklaying |
| 34 | m6 | masonry work (abutment 2) | 20 | ab6 | bricklaying |
| 35 | l | delivery of the preformed bearers | 2 | - | crane |
| 36 | t1 | positioning (preformed bearer 1) | 12 | m1, m2, l | crane |
| 37 | t2 | positioning (preformed bearer 2) | 12 | m2, m3, l | crane |
| 38 | t3 | positioning (preformed bearer 3) | 12 | m3, m4, l | crane |
| 39 | t4 | positioning (preformed bearer 4) | 12 | m4, m5, l | crane |
| 40 | t5 | positioning (preformed bearer 5) | 12 | m5, m6, l | crane |
| 41 | ua | removal of the temporary housing | 10 | - | noResource |
| 42 | v1 | filling 1 | 15 | t1 | caterpillar |
| 43 | v2 | filling 2 | 10 | t5 | caterpillar |
| 44 | pe | end of project | 0 | t2, t3, t4, v1, v2, ua | noResource |

Figure 11.12: Specification for bridge construction.

```

bridge(tasks:
  <Bridge task specification 103a>
constraints:
  proc {$ Start Dur}
    {ForAll [s1#b1 s2#b2 s3#b3 s4#b4 s5#b5 s6#b6]
      proc {$ A#B}
        (Start.B + Dur.B) - (Start.A + Dur.A) =:<: 4
      end}
    {ForAll [a1#s1 a2#s2 a5#s5 a6#s6 p1#s3 p2#s4]
      proc {$ A#B}
        Start.B - (Start.A + Dur.A) =:<: 3
      end}
    {ForAll [s1 s2 s3 s4 s5 s6]
      proc {$ A}
        Start.A >=: Start.ue + 6
      end}
    {ForAll [m1 m2 m3 m4 m5 m6]
      proc {$ A}
        (Start.A + Dur.A) - 2 =:<: Start.ua
      end}
    Start.l =: Start.pa + 30
    Start.pa = 0
  end)

```

3.

$$UE + 6 \leq Si, \quad 1 \leq i \leq 6$$

4.

$$(Mi + \text{dur}(Mi)) - 2 \leq UA, \quad 1 \leq i \leq 6$$

5.

$$L = PA + 30$$

Distribution Strategy

We first try the distribution strategy of Section 11.1.2, i.e. the first-fail strategy. The first solution of the problem is found with 97949 choice nodes and has a makespan of 133. After 500000 choice nodes no better solution is found. This is very unsatisfactory if we know that the optimal makespan is 104.

Thus, we try the distributor described in Section 11.1.3, i.e. the naive serializer. Now we find the first solution with makespan 120 with only 77 choice nodes. With 95 choice nodes we find a solution with makespan 112. After 500000 choice nodes no better solution is found.

In order to solve the problem we combine the ideas of first-fail and of serializers. The idea behind first-fail is to distribute first with a variable which has the smallest domain. This variable should occur in many constraints and should lead to much constraint propagation. The variable with the smallest domain acts as a bottleneck for the problem. This idea can be transferred to scheduling problems. A simple criterion for a resource to be a bottleneck is the sum of the durations of tasks to be scheduled on that resource. Hence, we will serialize first the tasks on a resource where the sum of durations is maximal.

Script

Figure 11.13: A scheduling compiler for the bridge problem.

```

fun {Compile Spec Capacity Serializer}
  TaskSpec    = Spec.tasks
  Constraints = <Extract additional constraints 86a>
  Dur         = {GetDur TaskSpec}
  TasksOnRes  = {GetTasksOnResource TaskSpec}
in
  proc {$ Start}
    Start = {GetStart TaskSpec}
    <Post precedence constraints 78a>
    {Constraints Start Dur}
    {Capacity    TasksOnRes Start Dur}
    {Serializer TasksOnRes Start Dur}
    <Assign start times 78b>
  end
end

```

Figure 11.13 shows the scheduling we will employ for the remaining problems. The variable `Constraints` refers to a binary procedure possibly containing additional constraints for a scheduling problem, it is computed as follows:

```

86a <Extract additional constraints 86a>≡
    if {HasFeature Spec constraints} then
      Spec.constraints
    else
      proc {$ _ _}
        skip
      end
    end
end

```

Note that we have parameterized the scheduling compiler with procedures to post the capacity constraints and the serializer. This makes it straightforward to solve the bridge problem with stronger techniques.

The procedure `DistributedSorted` orders the tasks on resources according to our bottleneck criterion (see Figure 11.14).

The optimal solution can be found by

Figure 11.14: Serializer that orders tasks by bottleneck criterion.

```

proc {DistributeSorted TasksOnRes Start Dur}
  fun {DurOnRes Ts}
    {FoldL Ts fun {$ D T}
      D+Dur.T
    end 0}
  end
in
  {ForAll {Sort {Record.toList TasksOnRes}
    fun {$ Ts1 Ts2}
      {DurOnRes Ts1} > {DurOnRes Ts2}
    end}
  proc {$ Ts}
    {ForAllTail Ts
      proc {$ T1|Tr}
        {ForAll Tr
          proc {$ T2}
            choice Start.T1 + Dur.T1 <=: Start.T2
            []      Start.T2 + Dur.T2 <=: Start.T1
          end
        end}
      end}
    end}
  end}
end

```

```

{ExploreBest {Compile Bridge
              Schedule.serializedDisj
              DistributeSorted}
  Earlier}

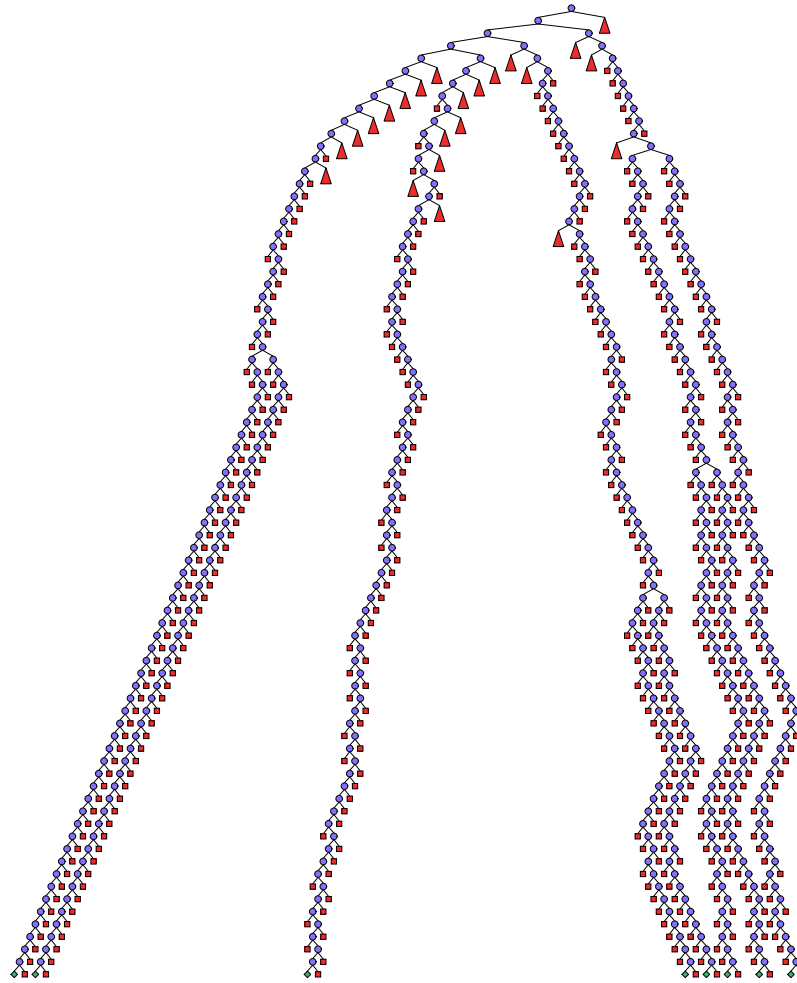
```

The full search tree consists of 1268 choice nodes and 8 solution nodes (see Figure 11.15).

The optimal solution can be visualized by a kind of Gantt-chart (see Figure 11.16). The makespan of the schedule (104 in this case) is indicated by a dashed line. Rectangles denote tasks. The left border of the rectangle indicates the start time of the task and the width of the rectangle indicates the duration of the task. Tasks scheduled on the same resource have the same texture.

The way we can solve scheduling problems by now seems to be satisfactory. But the current approach has two major flaws. First, the propagation of capacity constraint is rather weak. If we want to solve more demanding scheduling problems (like some benchmark problems from Operations Research) we need stronger propagation. Second, the bottleneck criterion of the serializer is rather coarse. We need more subtle techniques to solve more demanding problems. Furthermore, we need $(n \cdot (n - 1)) / 2$ ordering decisions for n tasks on the same resource which may result in deep search trees. This is not feasible for larger problems. Both problems will be solved in forthcoming sections.

Figure 11.15: A search tree for the bridge problem.



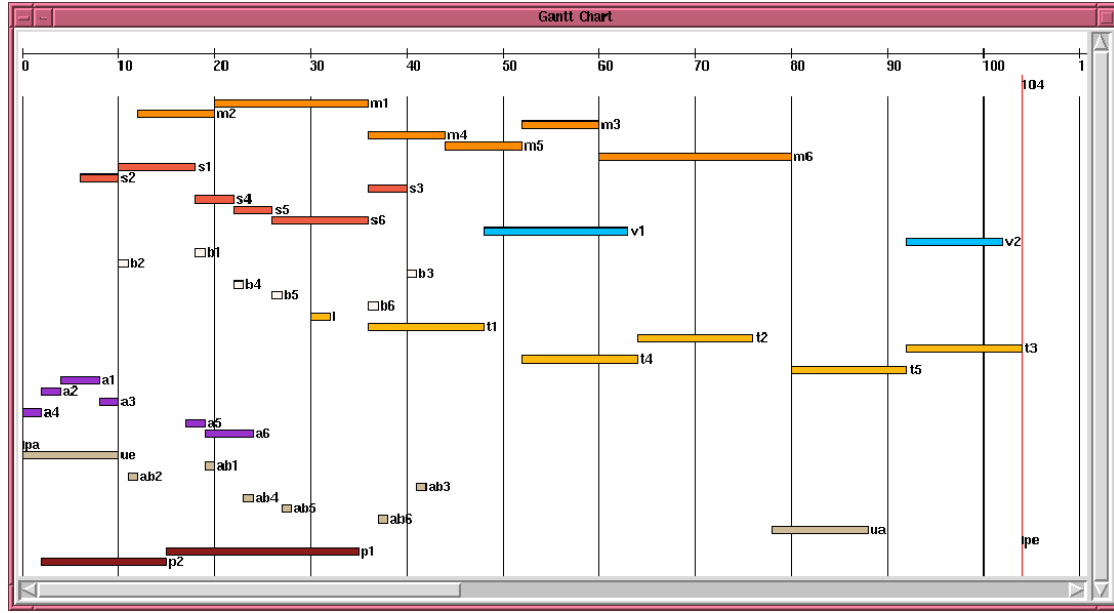
11.3 Strong Propagators for Capacity Constraints

In this section we introduce the ideas for stronger propagation employed for capacity constraints in Oz.

First we show the weakness of the propagators we have introduced so far. We consider three tasks A , B and C , each with duration 8 and with the domain $1\#10$. If we state for the pairs (A, B) , (A, C) and (B, C) that the contained tasks must not overlap in time by using reified constraints or by applying `Schedule.serializedDisj`, no further propagation will take place. This is due to the local reasoning on task pairs. For each pair no value in the corresponding domains can be discarded. On the other hand, the tasks must be scheduled between time point 1 and 18 (the latest completion time of either A , B or C). But because the overall duration is 24, this is impossible.

Hence, we will use stronger propagators reasoning simultaneously on the whole set of tasks on a resource. The principal ideas behind this reasoning are simple but very powerful. First, for an arbitrary set of tasks S to be scheduled on the same resource, the available time must be sufficient (see the example above). Furthermore, we check

Figure 11.16: The Gantt-chart for the bridge problem.



whether a task T in $\text{math}/S/$ must be scheduled as the first or last task of S (and analogously if T is not in S).

We introduce the following abbreviations for a task T .

| | |
|-----------------|---|
| $\text{est}(T)$ | least possible start time for T |
| $\text{lst}(T)$ | largest possible start time for T |
| $\text{ect}(T)$ | earliest completion time for T , i.e. $\text{ect}(T) = \text{est}(T) + \text{dur}(T)$ |
| $\text{lct}(T)$ | latest possible completion time for T , i.e., $\text{lct}(T) = \text{lst}(T) + \text{dur}(T)$ |

For a set S of tasks we define

$$\begin{aligned} \text{est}(S) &= \min(\{\text{est}(T) \mid T \in S\}) \\ \text{lct}(S) &= \max(\{\text{lct}(T) \mid T \in S\}) \\ \text{dur}(S) &= \sum_{T \in S} \text{dur}(T) \end{aligned}$$

If the condition

$$\text{lct}(S) - \text{est}(S) > \text{dur}(S)$$

holds, no schedule of the tasks in S can exist. A strong propagator for capacity constraints fails in this case.

Now we introduce some domain reductions by considering a task T and a set of tasks S where T does not occur in S . Assume that we can show that T cannot be scheduled after all tasks in S and that T cannot be scheduled between two tasks in S (if S contains at least two tasks). In this case we can conclude that T must be scheduled before all tasks in S .

More formally, if

$$\text{lct}(S) - \text{est}(S) < \text{dur}(S) + \text{dur}(T)$$

holds, T cannot be scheduled between $\text{lct}(S)$ and $\text{est}(S)$ (it cannot be scheduled between two tasks of S if S contains at least two tasks). If

$$\text{lct}(T) - \text{est}(S) < \text{dur}(S) + \text{dur}(T)$$

holds, T cannot be scheduled after all tasks in S . Hence, if both conditions hold, T must be scheduled before all tasks of S and corresponding propagators can be imposed, narrowing the domains of variables.

Analogously, if

$$\text{lct}(S) - \text{est}(S) < \text{dur}(S) + \text{dur}(T)$$

and

$$\text{lct}(S) - \text{est}(T) < \text{dur}(S) + \text{dur}(T)$$

holds, T must be last.

edge-finding Similar rules can be formulated if T is contained in S . For this kind of reasoning, the term edge-finding was coined in [2]. There are several variations of this idea in [5], [2], [6], [11] for the Operations Research community and in [13], [7], [3], [14] for the constraint programming community; they differ in the amount of propagation and which sets S are considered for edge-finding. The resulting propagators do a lot of propagation, but are also more expensive than e.g. reified constraints. Depending on the problem, one has to choose an appropriate propagator.

For unary resources Oz provides two propagators employing edge-finding to implement capacity constraints. The propagator `Schedule.serialize` is an improved version of an algorithm described in [11]. A single propagation step has complexity $O(n^2)$ where n is the number of tasks the propagator is reasoning on, i.e. the number of tasks on the resource considered by the propagator. Because the propagator runs until propagation reaches a fixed-point, we have the overall complexity of $O(k \cdot n^3)$ when k is the size of the largest domain of a task's start time (at most $k \cdot n$ values can be deleted from the domains of task variables).

The propagator `Schedule.taskIntervals` provides weaker propagation than described in [7] but provides stronger propagation than `Schedule.serialize`. While a single propagation step has complexity $O(n^3)$, the overall complexity is $O(k \cdot n^4)$.

Now we can solve the bridge construction problem with a propagator using edge-finding. By the statement

```
{ExploreBest {Compile Bridge
                Schedule.serialize
                DistributeSorted}
  Earlier}}
```

we compute the optimal solution in a full search tree with 508 choice nodes instead of 1268 as in the section before.

proof of optimality The improvement by strong propagation becomes even more dramatic if we constrain the bridge problem further by stating that the makespan must be strictly smaller than 104. Since we know that 104 is the optimal solution we, thus, prove optimality of this makespan. The modified problem specification is

```
OptBridge = {AdjoinAt Bridge constraints
  proc {$ Start Dur}
    {Bridge.constraints Start Dur}
    Start.pe <: 104
  end}
```

Solving the modified problem with the simple propagator by

```
{ExploreBest {Compile OptBridge
  Schedule.serializedDisj
  DistributeSorted}
  Earlier}
```

we obtain a search tree with 342 choice nodes. Using the edge-finding propagator `Schedule.serialized` instead we obtain a search tree with only 22 choice nodes. By using `Schedule.taskIntervals` the search tree shrinks further to the size of 17 choice nodes.

Note that for the proof of optimality the domains of the start times are rather narrow. If we start with an unconstrained problem, the domains are rather wide. But if the domains are more narrow compared to the durations of the tasks, the conditions we have described above are more likely to become true and propagation may take place. This is the reason why edge-finding turns out to be a stronger improvement for the proof of optimality.

11.4 Strong Serializers

So far we only have considered serializers which result in a search tree which depth grows quadratically in the number of tasks on a resource. In this section we will introduce a serializer where the depth of a search tree grows only linear in the number of tasks. In each choice node we will order several tasks not only two tasks, each choice node but several of them. This is done by stating that a single task must precede all other tasks on a resource.

A further disadvantage of the bottleneck serializer considered in Section 11.3 is its static bottleneck criterion. Instead we take the changing size of domains during run time into account to select a resource which should be serialized. This is also the approach chosen for the first-fail distribution strategy.

supply We start with a better criterion to select a resource. Let S be the set of tasks on a resource r . The available time to schedule all the tasks in S is $\text{lct}(S) - \text{est}(S)$. This value is called the supply of r . The overall time needed to schedule the tasks in S is $\text{dur}(S)$.

demand, global slack This value is called the demand of r . The difference between supply and demand is called global slack of r ($slack_g^r$) and denotes the free space on the resource. The smaller the value of the global slack the more critical is the resource (the tasks on it can be shifted only in a very limited way).

Hence, one could use the global slack as a criterion to select a resource. But a small example reveals that this criterion has still an important flaw: it is too coarse-grained. Assume S to be the set $\{A, B, C\}$ described in the following table.

| task | domain | dur(t) |
|------|--------|------------|
| A | 0#18 | 2 |
| B | 1#7 | 5 |
| C | 2#9 | 4 |

The global slack is $lct(S) - est(S) - dur(S) = 20 - 0 - 11 = 9$. But now consider the set $S' = \{B, C\}$. We obtain $lct(S') - est(S') - dur(S') = 13 - 1 - 9 = 3$. This means that for B and C we have far less free place to shift the tasks than it is indicated by the global slack. Thus, we refine our criterion as follows.

task interval Let T_1 and T_2 be two tasks on the same resource r and S the set of all tasks running on r . If $est(T_1) \leq est(T_2)$ and $lct(T_1) \leq lct(T_2)$, we call the set $I(T_1, T_2) = \{T \mid T \in S, est(T_1) \leq est(T), lct(T) \leq lct(T_2)\}$ the task interval defined by T_1 and T_2 (see also [7]). Intuitively, a task interval is the set of tasks which must be scheduled between $est(T_1)$ and $lct(T_2)$. Let I_r be the set of all task intervals on the resource r .

local slack The local slack of r ($slack_l^r$) is now defined as

$$\min(\{lct(I) - est(I) - dur(I) \mid I \in I_r\})$$

critical resource If two resources have the same local slack, we use the global slack to break this tie. Thus, we select the resource next for serialization which is minimal according to the lexicographic order $(slack_l^r, slack_g^r)$. The selected resource is called the critical resource. Note that a local slack of a resource with n tasks can be computed in $O(n^3)$ time.

Next we will determine the constraints to distribute with. Let $u(r)$ be the set of tasks on the critical resource r which are not ordered with all other tasks on r yet. Using the ideas of edge-finding we compute the set F of all tasks in $u(r)$ which can be scheduled first: $F = \{T \mid T \in u(r), lct(u(r) \setminus \{T\}) - est(T) \geq dur(u(r))\}$. In a distribution step each of the tasks in F , say T , may be scheduled before all others and T can be deleted from $u(r)$. The task in F which is smallest according to the lexicographic order $(est(T), lct(T))$ is first selected for distribution. By this choice we leave as much space for the remaining tasks to be scheduled on the resource. We now distribute with the constraints that T precedes all other tasks in $u(r)$: $\forall T' \in u(r) \setminus \{T\} : T + dur(T) \leq T'$. If this choice leads to failure, the next task in F is tried according to our criterion.

The overall strategy is as follows. We select a critical resource according to our criterion developed above. Then we serialize the critical resource by successively selecting

tasks to be scheduled before all others. After the critical resource is serialized, the next critical resource is selected for serialization. This process is repeated until all resources are serialized.

The described serializer follows the ideas of [3] which in turn adopts ideas of [5]. The serializer is available through `Schedule.firstsDist`.

We immediately apply our new serializer to the bridge problem.

```
{ExploreBest {Compile Bridge
                Schedule.serialized
                Schedule.firstsDist}
  Earlier}
```

The optimal solution can be found and its optimality can be proven with only 90 choice nodes. Now the proof of optimality (problem `OptBridge`) needs only 22 choice nodes.

But we can do better. In addition to the set F of tasks we can compute the set L of tasks which may be scheduled after all other tasks (see also Section 11.3). In this case the task T which is tried first to be scheduled after all the others is the one which is maximal according to the lexicographic order ($\text{lct}(T), \text{ect}(T)$). A further serializer computes both F and L . Then it selects the set which has the smallest cardinality. This serializer is available through `Schedule.firstsLastsDist`.

Using `Schedule.firstsLastsDist` we can find the optimal solution and prove its optimality with only 30 choice nodes (see Figure 11.17). Note that in contrast to Figure 11.15 where we have needed 8 solutions to reach the optimal one, we now find the optimal solution immediately. The size of the search tree is reduced by more than an order of magnitude.

The optimality of the problem can be proven with only 4 choice nodes.

Let m be the number of resources to consider in a scheduling problem and let n be the maximal number of tasks on a common resource. Then the described serializer has a run time complexity of $O(m \cdot n^3)$ if a resource has to be selected and $O(n)$ if only the set F or L has to be computed.

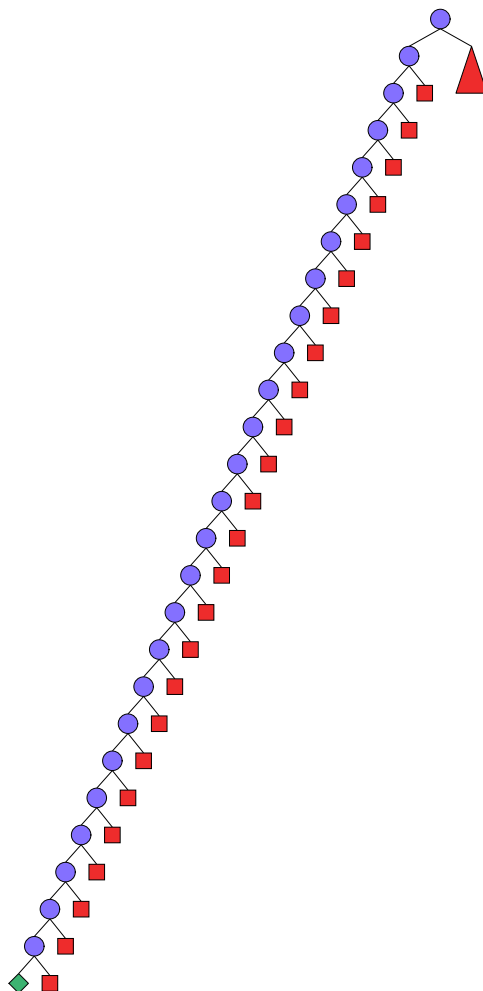
resource-oriented Because this kind of serializer successively serializes all resources, we call it resource-oriented serializer.

11.5 Solving Hard Scheduling Problems

In this section we tackle more difficult scheduling problems. To this aim we will also develop a new serializer.

We consider two problems in this section. Both are used as standard benchmark problems for scheduling. The first one is called ABZ6 and was introduced in [1]. The second one is the famous MT10 and was introduced in [12]. MT10 was considered as an especially hard problem for several years. It took more than 25 years that the optimality of a found makespan was proven [5].

Figure 11.17: A search tree for the bridge problem.



These problems belong to the class of so-called job-shop problems (see [9] or a good text book on scheduling). We slightly simplify the definition for our purposes. A job-shop problem consists of n jobs of tasks. Each job j consists of m tasks t_1^j through t_m^j such that each task of the job is scheduled on a different (unary) resource. Thus, we have m resources. Furthermore, we have the constraint $t_i^j + \text{dur}(t_i^j) \leq t_{i+1}^j$ for all tasks of job j , i.e. the tasks in a job are serialized. The latter constraints are already known as precedence constraints (see Section 11.1.1).

11.5.1 The Problem ABZ6

We will consider problem ABZ6 first. The specification is given in (page 104). The problem consists of 10 jobs and 10 resources. We first search for the optimal solution and prove its optimality:

```
{ExploreBest {Compile ABZ6
               Schedule.serialized
```

```
Schedule.firstsLastsDist}
Earlier}
```

The resulting search tree contains 2424 choice nodes. The optimal makespan is 943.

We now only want to prove the optimality of the makespan 943. To this aim we declare a modified problem as follows.

```
OptABZ6 = {AdjoinAt ABZ6 constraints
  proc {$ Start Dur}
    Start.pe <: 943
  end}
```

The proof of optimality needs 761 choice nodes.

Hence, the problem `ABZ6` seems to be rather easy to solve and we can try our previous bottleneck serializer `DistributeSorted`. To find the optimal solution and to prove its optimality a search tree is computed which contains more than 1.2 million choice nodes. Therefore, the problem *is* difficult for our simpler strategies and the gain by our new serializer is dramatic.

But we can do still better. To this aim we introduce a new serializer. This serializer will not serialize one resource after the other as the previous serializer. Instead a resource r is selected first. Then two tasks are selected which are running on r and it is distributed with a certain ordering. For the resource selection a criterion is used which combines the global slack and the local slack of each resource. For the task ordering the sets F and L are computed as shown in the previous section. From these sets two tasks are selected according to a subtle criterion (see [7]). After an ordering decision is made by distribution the process is repeated until all resources are serialized. In contrast to the strategy in Section 11.4, a task pair on a resource may be ordered without that the resource which was previously considered needs to be serialized.

task-oriented Thus, we call such a serializer a task-oriented serializer. The strategy implemented in Oz is very similar to the one suggested in [7].

Since we have to compute local slacks, the serializer has a run time complexity of $O(m \cdot n^3)$ in each step. Thus, it is more expensive than the resource-oriented serializer of the previous section. Furthermore, the use of this serializer might result in very deep search trees because we order only two tasks at each choice node. But the presented task-oriented serializer has a very important operational behavior besides the fact that it is used for distribution. While it is computing the local slacks of the resources it additionally employs edge-finding for the task intervals considered during this computation. In this way, the serializer may detect several orderings which must hold by the edge-finding rules presented in Section 11.3. This information is exploited at each choice node by additionally creating the corresponding propagators. Thus, the serializer orders two tasks by distribution and simultaneously adds orderings which are detected deterministically. By this approach the search tree may be reduced dramatically if edge-finding can be applied. As we have seen before, this is the case when the domains are rather narrow, i.e. for example when we want to prove optimality.

Oz provides the serializer `Schedule.taskIntervalsDistP` which has the described behavior. To prove optimality for `ABZ6` we now only need 145 choice nodes.

This serializer is especially designed for proving optimality. Hence, do not use this strategy when you want to find the optimal solution from scratch. If we search for the optimal solution the search tree becomes rather deep (a depth larger than 450) (including the proof of optimality) and the full tree contains more than 47000 choice nodes.

A variant of this task-oriented serializer is especially designed to find good solutions. To this aim Oz provides `Schedule.taskIntervalsDistO` (see also [8]). To find the optimal solution and to prove its optimality with this strategy we need 2979 choice nodes. But be aware that the use of this strategy may also lead to deep search trees which result in high memory consumption.

11.5.2 The MT10 Problem

In this section we tackle the famous `MT10` problem (the data specification is (page 105)). From the literature we know that 930 is the optimal makespan and we can define a script (compiled from `OptMT10`, see (page 106)) which can be used for proving optimality. The proof of optimality can be done with 1850 choice nodes:

```
{ExploreBest {Compile OptMT10
                Schedule.serialized
                Schedule.taskIntervalsDistP}
  Earlier}
```

Note that the depth of the search tree is only 39. This emphasizes the fact that many orderings can be determined by edge-finding which is employed by the task-oriented serializer.

To find the optimal solution we better use the serializer `Schedule.firstsLastsDist`:

```
{ExploreBest {Compile MT10
                Schedule.serialized
                Schedule.firstsLastsDist}
  Earlier}
```

The full search tree to find the optimal solution and to prove its optimality contains 16779 choice nodes and has depth 91.

Traps and Pitfalls

This section lists traps and pitfalls that beginners typically fall into when writing their first finite domain problem scripts in Oz.

Ordinary Arithmetic Blocks

There is a big difference between the statement

```
X+Y ==: Z
```

and the statement

```
X+Y = Z
```

The first statement creates a concurrent finite domain propagator and never blocks. The second statement creates an addition task that blocks until its arguments `x` and `y` are known. Blocking means that the statements following the addition statement will not be executed.

This pitfall can be particularly malicious if the infix expressions `(X mod Y)` or `(X div Y)` are used. For instance,

```
X mod Y ==: Z
```

is equivalent to

```
local A in
  X mod Y = A
  A ==: Z
end
```

and will thus block until `x` and `y` are determined. In contrast, a statement like

```
U + X*(Y-Z) ==: ~Y
```

is fine since the operations `+`, `*`, `-`, and `~` are implemented by the created propagator. The general rule behind this is simple: The infix operators `==:`, `\==:`

:

, `>:`, `=<:`, and `>=:` absorb the arithmetic operators `+`, `-`, `*`, and `~`, and no others.

Incidentally, interval and domain propagators for the modulo constraint can be created with the procedures `{FD.modI X Y Z}` and `{FD.modD X Y Z}`, respectively (see Section 2.4).

There is an easy way to check whether a statement in a script blocks: Just insert as last statement

```
{Browse 'End of script reached'}
```

and check in the Browser. If `'End of script reached'` appears in the Browser when you execute the script (e.g. with the Explorer), no statement in the script can have blocked, except for those that have been explicitly parallelized with `thread ... end`.

Delay of Propagators

Almost all propagators start their work only after all variables occurring in the implemented constraint are constrained to finite domains in the constraint store. For instance, the propagator created by

```
X*47 =: _
```

will never start propagation since it will wait forever that the anonymous variable created by the wildcard symbol `_` is constrained to a finite domain. This problem can easily be avoided by writing

```
X*47 =: {FD.decl}
```

The procedure `{FD.decl X}` constrains its argument to the largest finite domain possible (i.e. `0#sup`).

The Operators `=:` and `::` don't Introduce Pattern Variables

The statement

```
local X =: Y+Z in ... end
```

does not declare `X` as local variable, which is in contrast to the statement

```
local X = Y+Z in ... end
```

which however does not create a propagator. The desired effect can be obtained by writing

```
local X = {FD.decl} in X =: Y+Z ... end
```

A related pitfall is the wrong assumption that a statement

```
local X :: 4#5 in ... end
```

declares `X` as local variable. This is not the case. To obtain the desired effect, you can write

```
local X = {FD.int 4#5} in ... end
```

Delay of Domain Specifications

A domain specification like `X::L#U` constrains `X` only after both `L` and `U` are determined. Thus

```
L :: 5#13
U :: 14#33
X :: L#U
```

will constrain `X` only after both `L` and `U` have been determined.

Coreferences are not Always Realized

The propagator created by

```
A*A + B*B == C*C
```

provides much less propagation than the four propagators created by

```
{FD.times A A} + {FD.times B B} == {FD.times C C}
```

The reason is that the first propagator does not realize the coreferences in the constraint it implements, that is, it treats the two occurrences of `A`, say, as if they were independent variables. On the other hand, the propagator created by `{FD.times A A $}` exploits this coreference to provide better propagation. The Pythagoras Puzzle (see Section 7.2) is a problem, where exploiting coreferences is essential).

Large Numbers

There is an implementation-dependent upper bound for the integers that can occur in a finite domain stored in the constraint store. This upper bound is available as the value of `FD.sup`. In Mozart, `FD.sup` is 134 217 726 on Linux and Sparc platforms.

The same restriction applies to constants appearing in propagators. For instance, the creation of a propagator

```
X*Y <: 900*1000*1000
```

will result in a run-time error since the constant 900 000 000 computed by the compiler is larger than `FD.sup`. There is a trick that solves the problem for some cases. The trick consists in giving a large number as a product involving an auxiliary variable:

```
local A = 900 in
  X*Y <: A*1000*1000
end
```

The trick exploits that propagators can compute internally with integers larger than `FD.sup`, and that the compiler does not eliminate the auxiliary variable. The Grocery Puzzle in Section 4.1 uses this trick.

Golden Rules

We offer the following rules for the design of efficient constraint programs.

Analyze and Understand your Script

The first script for a difficult problem usually does not show satisfactory performance, even if you are expert. To improve it, you need to analyze and understand the search tree. The Explorer is a powerful tool for doing this. Use the statistics feature of the Panel to analyse the performance of your script: how many variables and propagators have been created? How often where the propagators invoked?

Experiment

Once you have analyzed the search tree and performance of your script, start to experiment with different models, different distribution strategies, and propagators for redundant constraints.

Have as much Constraint Propagation as Possible

More constraint propagation results in smaller search trees. Try to design a model that yields strong propagation. Try to eliminate symmetries by imposing canonical orders. Finally, try to find redundant constraints that result in stronger propagation when imposed as propagators.

Find a Good Distribution Strategy

A good distribution strategy can reduce the size of the search trees dramatically. Usually, it's a good idea to start with a first-fail strategy. The Grocery Puzzle (see Section 4.1) is an example where domain splitting is much better than trying the least possible value. Our script for the Queens Puzzle (see Section 5.1) can solve the puzzle even for large N 's by using a first-fail distribution strategy that tries the value in the middle of the domain of the selected variable first.

Keep the Number of Variables and Propagators Low

The memory consumption of a script depends very much on the number of propagators and finite domain variables created. Models that keep these numbers low usually lead to more efficient scripts. The model for the Queens Problem in Section 5.1 is particularly successful in keeping the number of propagators low.

Use the statistics feature of the Panel to find out how many variables and propagators were created.

This rule conflicts with the rule asking for maximization of constraint propagation. Extra propagators for redundant constraints will improve performance if they reduce significantly either the size of search tree or the number of propagation steps (for the latter, see the Pythagoras Example in Section 7.2).

Eliminate Symmetries

It is always a good idea to design a model such that symmetries are avoided as much as possible. The model for the Queens Puzzle (see Section 5.1) avoids possible symmetries by having a minimal number of variables. The models for the Grocery and Family Puzzles (see Section 4.1 and Section 4.2) eliminate symmetries by imposing a canonical order on the variables by means of additional constraints. The model of the Grocery Puzzle eliminates a subtle symmetry by stating that the price of the first item must have a large prime factor in common with the product of the prices of the items. The Fraction Puzzle (see Section 7.1) eliminates symmetries by imposing an order on the three occurring fractions.

Introduce Propagators for Redundant Constraints

Propagators for redundant constraints can often strengthen a script's propagation. A redundant constraint is a constraint that is logically entailed by the constraints specifying the problem. Try to find redundant constraints that yield nonredundant propagators. The models for the Fraction and Magic Square puzzles (see Section 7.1 and Section 7.3) feature good examples for nonredundant propagators for redundant constraints.

Use Recomputation if Memory Consumption is a Problem

Scripts which create a large number of variables or propagators or scripts for which the search tree is very deep might use too much memory to be feasible. Search engines described in Chapter *Search Engines: Search*, (*System Modules*) and the Explorer (see “*Oz Explorer – Visual Constraint Programming Support*”) feature support for so-called recomputation. Recomputation reduces the space requirements for these problems in that it trades space for time.

Example Data

The following appendix features some data specifications omitted in the chapters' text.

Scheduling

103a **Bridge task specification 103a** ≡

```
[pa(dur: 0)
a1(dur: 4 pre:[pa] res:excavator)
a2(dur: 2 pre:[pa] res:excavator)
a3(dur: 2 pre:[pa] res:excavator)
a4(dur: 2 pre:[pa] res:excavator)
a5(dur: 2 pre:[pa] res:excavator)
a6(dur: 5 pre:[pa] res:excavator)
p1(dur:20 pre:[a3] res:pileDriver)
p2(dur:13 pre:[a4] res:pileDriver)
ue(dur:10 pre:[pa])
s1(dur: 8 pre:[a1] res:carpentry)
s2(dur: 4 pre:[a2] res:carpentry)
s3(dur: 4 pre:[p1] res:carpentry)
s4(dur: 4 pre:[p2] res:carpentry)
s5(dur: 4 pre:[a5] res:carpentry)
s6(dur:10 pre:[a6] res:carpentry)
b1(dur: 1 pre:[s1] res:concreteMixer)
b2(dur: 1 pre:[s2] res:concreteMixer)
b3(dur: 1 pre:[s3] res:concreteMixer)
b4(dur: 1 pre:[s4] res:concreteMixer)
b5(dur: 1 pre:[s5] res:concreteMixer)
b6(dur: 1 pre:[s6] res:concreteMixer)
ab1(dur:1 pre:[b1])
ab2(dur:1 pre:[b2])
ab3(dur:1 pre:[b3])
ab4(dur:1 pre:[b4])
ab5(dur:1 pre:[b5])
ab6(dur:1 pre:[b6])
m1(dur:16 pre:[ab1] res:bricklaying)
m2(dur: 8 pre:[ab2] res:bricklaying)
m3(dur: 8 pre:[ab3] res:bricklaying)
```

```

m4(dur: 8 pre:[ab4] res:bricklaying)
m5(dur: 8 pre:[ab5] res:bricklaying)
m6(dur:20 pre:[ab6] res:bricklaying)
l(dur: 2 res:crane)
t1(dur:12 pre:[m1 m2 l] res:crane)
t2(dur:12 pre:[m2 m3 l] res:crane)
t3(dur:12 pre:[m3 m4 l] res:crane)
t4(dur:12 pre:[m4 m5 l] res:crane)
t5(dur:12 pre:[m5 m6 l] res:crane)
ua(dur:10)
v1(dur:15 pre:[t1] res:caterpillar)
v2(dur:10 pre:[t5] res:caterpillar)
pe(dur: 0 pre:[t2 t3 t4 v1 v2 ua])]
```

104a **<ABZ6 Specification 104a>**≡

```

abz6(tasks:
[pa(dur: 0)
a1(dur:62 pre:[pa] res:m7) a2(dur:24 pre:[a1] res:m8)
a3(dur:25 pre:[a2] res:m5) a4(dur:84 pre:[a3] res:m3)
a5(dur:47 pre:[a4] res:m4) a6(dur:38 pre:[a5] res:m6)
a7(dur:82 pre:[a6] res:m2) a8(dur:93 pre:[a7] res:m0)
a9(dur:24 pre:[a8] res:m9) a10(dur:66 pre:[a9] res:m1)
b1(dur:47 pre:[pa] res:m5) b2(dur:97 pre:[b1] res:m2)
b3(dur:92 pre:[b2] res:m8) b4(dur:22 pre:[b3] res:m9)
b5(dur:93 pre:[b4] res:m1) b6(dur:29 pre:[b5] res:m4)
b7(dur:56 pre:[b6] res:m7) b8(dur:80 pre:[b7] res:m3)
b9(dur:78 pre:[b8] res:m0) b10(dur:67 pre:[b9] res:m6)
c1(dur:45 pre:[pa] res:m1) c2(dur:46 pre:[c1] res:m7)
c3(dur:22 pre:[c2] res:m6) c4(dur:26 pre:[c3] res:m2)
c5(dur:38 pre:[c4] res:m9) c6(dur:69 pre:[c5] res:m0)
c7(dur:40 pre:[c6] res:m4) c8(dur:33 pre:[c7] res:m3)
c9(dur:75 pre:[c8] res:m8) c10(dur:96 pre:[c9] res:m5)
d1(dur:85 pre:[pa] res:m4) d2(dur:76 pre:[d1] res:m8)
d3(dur:68 pre:[d2] res:m5) d4(dur:88 pre:[d3] res:m9)
d5(dur:36 pre:[d4] res:m3) d6(dur:75 pre:[d5] res:m6)
d7(dur:56 pre:[d6] res:m2) d8(dur:35 pre:[d7] res:m1)
d9(dur:77 pre:[d8] res:m0) d10(dur:85 pre:[d9] res:m7)
e1(dur:60 pre:[pa] res:m8) e2(dur:20 pre:[e1] res:m9)
e3(dur:25 pre:[e2] res:m7) e4(dur:63 pre:[e3] res:m3)
e5(dur:81 pre:[e4] res:m4) e6(dur:52 pre:[e5] res:m0)
e7(dur:30 pre:[e6] res:m1) e8(dur:98 pre:[e7] res:m5)
e9(dur:54 pre:[e8] res:m6) e10(dur:86 pre:[e9] res:m2)
f1(dur:87 pre:[pa] res:m3) f2(dur:73 pre:[f1] res:m9)
f3(dur:51 pre:[f2] res:m5) f4(dur:95 pre:[f3] res:m2)
f5(dur:65 pre:[f4] res:m4) f6(dur:86 pre:[f5] res:m1)
f7(dur:22 pre:[f6] res:m6) f8(dur:58 pre:[f7] res:m8)
f9(dur:80 pre:[f8] res:m0) f10(dur:65 pre:[f9] res:m7)
g1(dur:81 pre:[pa] res:m5) g2(dur:53 pre:[g1] res:m2)
g3(dur:57 pre:[g2] res:m7) g4(dur:71 pre:[g3] res:m6)
```

```

g5(dur:81 pre:[g4] res:m9) g6(dur:43 pre:[g5] res:m0)
g7(dur:26 pre:[g6] res:m4) g8(dur:54 pre:[g7] res:m8)
g9(dur:58 pre:[g8] res:m3) g10(dur:69 pre:[g9] res:m1)
h1(dur:20 pre:[pa] res:m4) h2(dur:86 pre:[h1] res:m6)
h3(dur:21 pre:[h2] res:m5) h4(dur:79 pre:[h3] res:m8)
h5(dur:62 pre:[h4] res:m9) h6(dur:34 pre:[h5] res:m2)
h7(dur:27 pre:[h6] res:m0) h8(dur:81 pre:[h7] res:m1)
h9(dur:30 pre:[h8] res:m7) h10(dur:46 pre:[h9] res:m3)
i1(dur:68 pre:[pa] res:m9) i2(dur:66 pre:[i1] res:m6)
i3(dur:98 pre:[i2] res:m5) i4(dur:86 pre:[i3] res:m8)
i5(dur:66 pre:[i4] res:m7) i6(dur:56 pre:[i5] res:m0)
i7(dur:82 pre:[i6] res:m3) i8(dur:95 pre:[i7] res:m1)
i9(dur:47 pre:[i8] res:m4) i10(dur:78 pre:[i9] res:m2)
j1(dur:30 pre:[pa] res:m0) j2(dur:50 pre:[j1] res:m3)
j3(dur:34 pre:[j2] res:m7) j4(dur:58 pre:[j3] res:m2)
j5(dur:77 pre:[j4] res:m1) j6(dur:34 pre:[j5] res:m5)
j7(dur:84 pre:[j6] res:m8) j8(dur:40 pre:[j7] res:m4)
j9(dur:46 pre:[j8] res:m9) j10(dur:44 pre:[j9] res:m6)
pe(dur:0 pre:[a10 b10 c10 d10 e10 f10 g10 h10 i10 j10]))

```

105a **⟨MT10 Specification 105a⟩**≡

```

mt10(tasks:
[pa(dur:0)
a1(dur:29 pre:[pa] res:m1) a2(dur:78 pre:[a1] res:m2)
a3(dur: 9 pre:[a2] res:m3) a4(dur:36 pre:[a3] res:m4)
a5(dur:49 pre:[a4] res:m5) a6(dur:11 pre:[a5] res:m6)
a7(dur:62 pre:[a6] res:m7) a8(dur:56 pre:[a7] res:m8)
a9(dur:44 pre:[a8] res:m9) a10(dur:21 pre:[a9] res:m10)
b1(dur:43 pre:[pa] res:m1) b2(dur:90 pre:[b1] res:m3)
b3(dur:75 pre:[b2] res:m5) b4(dur:11 pre:[b3] res:m10)
b5(dur:69 pre:[b4] res:m4) b6(dur:28 pre:[b5] res:m2)
b7(dur:46 pre:[b6] res:m7) b8(dur:46 pre:[b7] res:m6)
b9(dur:72 pre:[b8] res:m8) b10(dur:30 pre:[b9] res:m9)
c1(dur:91 pre:[pa] res:m2) c2(dur:85 pre:[c1] res:m1)
c3(dur:39 pre:[c2] res:m4) c4(dur:74 pre:[c3] res:m3)
c5(dur:90 pre:[c4] res:m9) c6(dur:10 pre:[c5] res:m6)
c7(dur:12 pre:[c6] res:m8) c8(dur:89 pre:[c7] res:m7)
c9(dur:45 pre:[c8] res:m10) c10(dur:33 pre:[c9] res:m5)
d1(dur:81 pre:[pa] res:m2) d2(dur:95 pre:[d1] res:m3)
d3(dur:71 pre:[d2] res:m1) d4(dur:99 pre:[d3] res:m5)
d5(dur: 9 pre:[d4] res:m7) d6(dur:52 pre:[d5] res:m9)
d7(dur:85 pre:[d6] res:m8) d8(dur:98 pre:[d7] res:m4)
d9(dur:22 pre:[d8] res:m10) d10(dur:43 pre:[d9] res:m6)
e1(dur:14 pre:[pa] res:m3) e2(dur: 6 pre:[e1] res:m1)
e3(dur:22 pre:[e2] res:m2) e4(dur:61 pre:[e3] res:m6)
e5(dur:26 pre:[e4] res:m4) e6(dur:69 pre:[e5] res:m5)
e7(dur:21 pre:[e6] res:m9) e8(dur:49 pre:[e7] res:m8)
e9(dur:72 pre:[e8] res:m10) e10(dur:53 pre:[e9] res:m7)
f1(dur:84 pre:[pa] res:m3) f2(dur: 2 pre:[f1] res:m2)

```

```

f3(dur:52 pre:[f2] res:m6) f4(dur:95 pre:[f3] res:m4)
f5(dur:48 pre:[f4] res:m9) f6(dur:72 pre:[f5] res:m10)
f7(dur:47 pre:[f6] res:m1) f8(dur:65 pre:[f7] res:m7)
f9(dur: 6 pre:[f8] res:m5) f10(dur:25 pre:[f9] res:m8)
g1(dur:46 pre:[pa] res:m2) g2(dur:37 pre:[g1] res:m1)
g3(dur:61 pre:[g2] res:m4) g4(dur:13 pre:[g3] res:m3)
g5(dur:32 pre:[g4] res:m7) g6(dur:21 pre:[g5] res:m6)
g7(dur:32 pre:[g6] res:m10) g8(dur:89 pre:[g7] res:m9)
g9(dur:30 pre:[g8] res:m8) g10(dur:55 pre:[g9] res:m5)
h1(dur:31 pre:[pa] res:m3) h2(dur:86 pre:[h1] res:m1)
h3(dur:46 pre:[h2] res:m2) h4(dur:74 pre:[h3] res:m6)
h5(dur:32 pre:[h4] res:m5) h6(dur:88 pre:[h5] res:m7)
h7(dur:19 pre:[h6] res:m9) h8(dur:48 pre:[h7] res:m10)
h9(dur:36 pre:[h8] res:m8) h10(dur:79 pre:[h9] res:m4)
i1(dur:76 pre:[pa] res:m1) i2(dur:69 pre:[i1] res:m2)
i3(dur:76 pre:[i2] res:m4) i4(dur:51 pre:[i3] res:m6)
i5(dur:85 pre:[i4] res:m3) i6(dur:11 pre:[i5] res:m10)
i7(dur:40 pre:[i6] res:m7) i8(dur:89 pre:[i7] res:m8)
i9(dur:26 pre:[i8] res:m5) i10(dur:74 pre:[i9] res:m9)
j1(dur:85 pre:[pa] res:m2) j2(dur:13 pre:[j1] res:m1)
j3(dur:61 pre:[j2] res:m3) j4(dur: 7 pre:[j3] res:m7)
j5(dur:64 pre:[j4] res:m9) j6(dur:76 pre:[j5] res:m10)
j7(dur:47 pre:[j6] res:m6) j8(dur:52 pre:[j7] res:m4)
j9(dur:90 pre:[j8] res:m5) j10(dur:45 pre:[j9] res:m8)
pe(dur:0 pre:[a10 b10 c10 d10 e10 f10 g10 h10 i10 j10]))

```

106a **Definition of OptMT10 106a**≡

```

OptMT10 = {AdjoinAt MT10 constraints
  proc {$ Start Dur}
    Start.pe <: 930
  end}

```

Answers To Exercises

Answer to exercise 3.1. This does not really require to be answered. Just try it.

Answer to exercise 3.2.

```

proc {Donald Root}
  sol(a:A b:B d:D e:E g:G l:L n:N o:O r:R t:T) = Root
in
  Root ::: 0#9
  {FD.distinct Root}
  D\=:0 R\=:0 G\=:0
    100000*D + 10000*O + 1000*N + 100*A + 10*L + D
  + 100000*G + 10000*E + 1000*R + 100*A + 10*L + D
  =: 100000*R + 10000*O + 1000*B + 100*E + 10*R + T
  {FD.distribute split Root}
end

```

Answer to exercise 7.1. The first redundant constraint follows from the fact that the total number of occurrences in the sequence is n , and that no numbers but those between 0 and $n - 1$ occur in the sequence.

The second redundant constraint follows from the fact that

$$0 \cdot x_0 + \dots + (n-1) \cdot x_{n-1} = x_0 + \dots + x_{n-1}$$

Here is a parametrized script for the Magic Sequence Puzzle:

```

fun {MagicSequence N}
  Cs = {List.number ~1 N-2 1}
in
  proc {$ S}
    {FD.tuple sequence N 0#N-1 S}
    {For 0 N-1 1
      proc {$ I} {FD.exactly S.(I+1) S I} end}
    %% redundant constraints
    {FD.sum S '=: ' N}
    {FD.sumC Cs S '=: ' 0}
  end
end

```

```

%%
{FD.distribute ff S}
end
end

```

Answer to exercise 8.1.

```
{FD.impl X<:Y X+Y=:Z} = {FD.disj X*Y=:Z Z\=:5}
```

Answer to exercise 8.2.

```

proc {Conj X Y Z}
  X::0#1 Y::0#1
  (X+Y=:2) = Z
end
proc {Equi X Y Z}
  X::0#1 Y::0#1
  (X=:Y) = Z
end
proc {Nega X Z}
  X::0#1
  (X=:0) = Z
end
proc {Disj X Y Z}
  X::0#1 Y::0#1
  (X+Y>:0) = Z
end
proc {Impl X Y Z}
  X::0#1 Y::0#1
  (X=:0) + (Y>:0) = Z
end

```

Answer to exercise 8.3. To minimize the value of `Satisfaction`, we modify the distributor for `Satisfaction` such that it tries smaller values first:

```
{FD.distribute generic(order:naive value:min) [Satisfaction]}
```

It turns out that the persons can be aligned such that no preference is satisfied.

Answer to exercise 8.4.

```

local Aux in
  {FD.decl Aux}
  {FD.distance Q.7 Q.8 '=: ' Aux}
  {FD.element Q.7 [4 3 2 1 0] Aux}
end

```

Answer to exercise 8.5.

```

thread
  if A.type==B.type then
    A.glass >=: B.glass
  end
end

```

Answer to exercise 11.1. A possible solution is as follows.

```

proc {CapacityConstraints TasksOnRes Start Dur}
  {Record.forAll TasksOnRes
    proc {$ Ts}
      {ForAllTail Ts
        proc {$ T1|Tr}
          {ForAll Tr
            proc {$ T2}
              (Start.T1 + Dur.T1 <=: Start.T2) +
              (Start.T2 + Dur.T2 <=: Start.T1) =: 1
            end}
          end}
        end}
      end}
    end}
  end
end

```

Bibliography

- [1] J. Adams, E. Balas, and D. Zawack. The shifting bottleneck procedure for job shop scheduling. *Management science*, 34(3):391–401, 1988.
- [2] David Applegate and William Cook. A computational study of the job-shop scheduling problem. *Operations Research Society of America, Journal on Computing*, 3(2):149–156, 1991.
- [3] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. Incorporating efficient operations research algorithms in constraint-based scheduling. In *First International Joint Workshop on Artificial Intelligence and Operations Research*, 1995.
- [4] Martin Bartusch. *Optimierung von Netzplänen mit Anordnungsbeziehungen bei knappen Betriebsmitteln*. PhD thesis, Universität Passau, Fakultät für Mathematik und Informatik, 1983.
- [5] J. Carlier and E. Pinson. An algorithm for solving the job-shop problem. *Management Science*, 35(2):164–176, 1989.
- [6] J. Carlier and E. Pinson. Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, 78:146–161, 1994.
- [7] Yves Caseau and François Laburthe. Improved CLP scheduling with task intervals. In *International Conference on Logic Programming*, pages 369–383, 1994.
- [8] Yves Caseau and François Laburthe. Disjunctive scheduling with task intervals. LIENS Technical Report 95-25, Laboratoire d’Informatique de l’Ecole Normale Supérieure, 1995.
- [9] Michael R. Garey and David S. Johnson. *Computers and Intractability*. W.H. Freeman and company, 1979.
- [10] Martin Henz. Don’t be puzzled! In *Proceedings of the Workshop on Constraint Programming Applications, in conjunction with the Second International Conference on Principles and Practice of Constraint Programming (CP96)*, Cambridge, Massachusetts, USA, August 1996.
- [11] Paul Martin and David B. Shmoys. A new approach to computing optimal schedules for the job shop scheduling problem. In *International Conference on Integer Programming and Combinatorial Optimization, Vancouver*, pages 389–403, 1996.
- [12] J. F. Muth and G. L. Thompson. *Industrial Scheduling*. Prentice Hall, Englewood Cliffs, NJ, USA, 1963.

- [13] Wim Nuijten. *Time and resource constrained scheduling*. PhD thesis, Technical University Eindhoven, 1994.
- [14] Jörg Würtz. Oz Scheduler: A workbench for scheduling problems. In M.G. Randle, editor, *Eighth International Conference on Tools with Artificial Intelligence*, pages 149–156, Toulouse, France, 1996. IEEE, IEEE Computer Society Press.