FICO® Xpress Kalis Module for Mosel

13.4.2

USER GUIDE

FICO® Xpress Optimization

Artelys
solutions en optimisation

FICO®

Xpress Kalis 13.4.2 (FICO® Xpress 9.7)
Deliverable Version: A
Last Revised: 29 July, 2025

How to Contact the Xpress Team

Sales and Maintenance

If you need information on other Xpress Optimization products, or you need to discuss maintenance contracts or other sales-related items, contact FICO by:

- Phone: +1 (408) 535-1500 or +44 207 940 8718
- Web: www.fico.com/optimization and use the available contact forms

Product Support

Customer Self Service Portal (online support): www.fico.com/en/product-support

Email: Support@fico.com (Please include *'Xpress'* in the subject line)

For the latest news and Xpress software and documentation updates, please visit the Xpress website at http://www.fico.com/xpress or subscribe to our mailing list.

How to Contact Artelys

Artelys SA

Information and Sales: info-kalis@artelys.com
Licensing and Product Support: support-kalis@artelys.com

Tel: +33 1 44 77 89 00
Fax: +33 1 42 96 22 61

12, rue du Quatre Septembre
75002 Paris Cedex
France

For the latest news about Kalis, training course programs, and examples, please visit the Artelys website at http://www.artelys.com.

Contents

CHAPTER 1

Introduction

Constraint Programming is an approach to problem solving that has been particularly successful for dealing with nonlinear constraint relations over discrete variables. In the following we therefore often use 'Constraint Programming' or 'CP' synonymously to 'finite domain Constraint Programming'.

In the past, CP has been successfully applied to such different problem types as production scheduling (with or without resource constraints), sequencing and assignment of tasks, workforce planning and timetabling, frequency assignment, loading and cutting, and also graph coloring.

The strength of CP lies in its use of a high-level semantics for stating the constraints that preserves the original meaning of the constraint relations (such high-level constraints are referred to as *global constraints*). It is not necessary to translate constraints into an arithmetic form—a process whereby sometimes much of the problem structure is lost. The knowledge about a problem inherent to the constraints is exploited by the solution algorithms, rendering them more efficient.

1.1 Xpress Kalis Module for Mosel

The *Xpress Kalis Module for Mosel*, or short *Kalis for Mosel*, provides access to the FICO® Xpress Kalis Constraint Programming solver by Artelys from Mosel via the module *kalis*. Through the *kalis* module, the Constraint Programming functionality of Kalis becomes available in the Mosel environment, allowing the user to formulate and solve CP models in the Mosel language. Xpress Kalis combines a finite domain solver and a solver over continuous (floating point) variables. To aid the formulation of scheduling problems, the software defines specific aggregate modeling objects representing tasks and resources that will automatically setup certain (implicit) constraint relations and trigger the use of built-in search strategies specialized for this type of problem. Standard scheduling problems may thus be defined and solved simply by setting up the corresponding task and resource objects. Additionally, Xpress Kalis provides automatic linear relaxations of its constraints to ease the formulation and solving of a wide range of optimization problems.

All data handling facilities of the Mosel environment, including data transfer in memory (using the Mosel IO drivers) and ODBC access to databases (through the module *mmodbc*) can be used with *kalis* without any extra effort.

The Mosel language supports typical programming constructs, such as loops, subroutines, *etc.*, that may be required to implement more complicated algorithms. Mosel can also be used as a platform for combining different solvers, in particular Xpress Kalis with Xpress Optimizer for joint CP – LP/MIP problem solving[1]. This manual explains the basics on modeling and programming with Mosel and, where necessary, also some more advanced features. For a complete documentation and a more thorough introduction to its use the reader is referred to the Mosel Language Reference Manual and the Mosel User Guide.

[1] See the Xpress Whitepapers Multiple models and parallel solving with Mosel and Hybrid MIP/CP solving with Xpress Optimizer and Xpress Kalis

1.1.1 Note on product versions

Most examples in this manual have originally been developed using the release 2007.1.0 of Xpress Kalis, their implementations have been updated to release 13.2 of Xpress Kalis with the Xpress Release 8.14 of Mosel (6.0). If they are run with other product versions the output obtained may look different. In particular, improvements to the algorithms in the CP solver or modifications to the default settings in Xpress Kalis may influence the behavior of the constraints or the search. Xpress Workbench screenshots have been updated to Xpress Release 8.7.

1.2 Software installation

To be able to work with Xpress Kalis, the Xpress Mosel software and Xpress Kalis must be installed and licensed. Users may find it convenient to install (in addition) the graphical environment Xpress Workbench for working with CP models, but this is not a prerequisite: Mosel models can be edited with any text editor and simply be executed from the command line. The examples of Chapter 6 of this manual can only be executed if Xpress Optimizer is also present and licensed.

Follow the installation instructions provided with the Xpress distribution for installing Mosel (and Xpress Workbench) and the required solvers on your computer by selecting the Xpress Kalis add-on option during the installation process.

1.3 Basic concepts of Constraint Programming

A *Constraint Programming (CP) problem* is defined by its *decision variables* with their *domains* and *constraints* over these variables. The problem definition is usually completed by a *branching strategy* (also referred to as *enumeration* or *search strategy*).

CP makes active use of the concept of *variable domains*, that is, the set out of which a decision variable takes its values. In *finite domain Constraint Programming* these are sets or intervals of integer numbers.

Each *constraint* in CP comes with its own (set of) solution algorithm(s), typically based on results from other areas, such as graph theory. Once a constraint has been established it maintains its set of variables in a solved state, *i.e.*, its solution algorithm removes any values that it finds infeasible from the domains of the variables.

The constraints in a CP problem are linked by a mechanism called *constraint propagation*: whenever the domain of a variable is modified this triggers a re-evaluation of all constraints on this variable which in turn may cause modifications to other variables or further reduction of the domain of the first variable as shown in the example in Figure 1.1 (the original domains of the variables are reduced by the addition of two constraints; in the last step the effect of the second constraint is propagated to the first constraint, triggering its re-evaluation).

A CP problem is built up *incrementally* by adding constraints and bounds on its variables. The solving of a CP problem starts with the statement of the first constraint—values that violate the constraint relation are removed from the domains of the involved variables. Since the effect of a newly added constraint is propagated immediately to the entire CP problem it is generally not possible to modify or delete this constraint from the problem later on.

In some cases the combination of constraint solving and the propagation mechanism may be sufficient to prove that a problem instance is infeasible, but most of the time it will be necessary to add an *enumeration* for reducing all variable domains to a single value (*consistent instantiation* or *feasible solution*) or proving that no such solution exists. In addition it is possible to define an *objective function* (or *cost function*) and search for a feasible solution with the best objective function value (*optimal solution*).

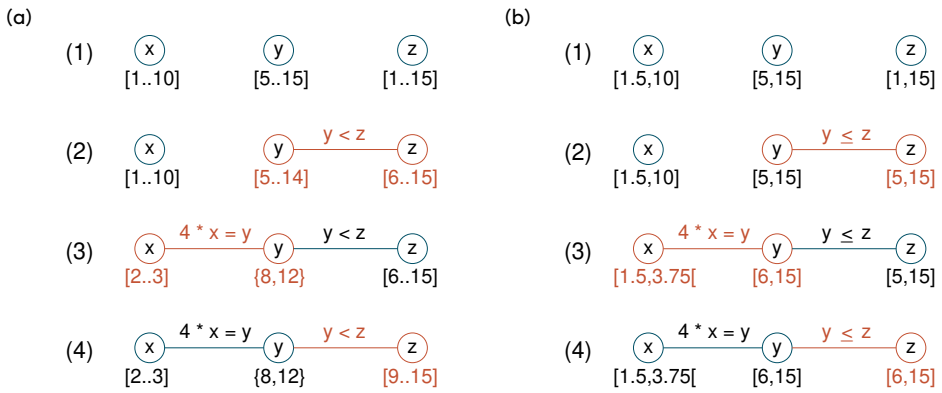

Figure 1.1: Example of constraint propagation, (a) finite domain (discrete) variables, (b) continuous variables.

1.4 Contents of this document

This document gives an introduction to working with Xpress Kalis from Mosel. Basic notions of Constraint Programming are explained but it is expected that the reader has some general understanding of CP techniques. Since we are working with the *kalis* module for Xpress Mosel this document also describes features of the Mosel language where this appears neccessary to the understanding of the presented examples.

By the means of example models we illustrate the use of Xpress Kalis, presenting the new types that are defined by the *kalis* module, namely

- Decision variables: finite domain and floating point variables
- Constraints: absolute value and distance, all-different, element, generic binary, linear, maximum and minimum, occurrence, and logical relations
- Enumeration: predefined branching schemes and user search strategies
- Scheduling: aggregate modeling objects representing tasks and resources

The first chapter deals with the basics of writing CP models with Mosel. It explains the general structure of CP models and some basics on working with Mosel, including data handling. Then follows a series of small problems illustrating the use of the different constraint relation types in Xpress Kalis. The next chapter introduces in a more systematic way the different possibilities of defining enumeration strategies, some of which already appear in the preceding model examples. The chapter dedicated to the topic of scheduling introduces the modeling objects 'task' and 'resource' that simplify the formulation of scheduling problems. The following chapter familiarizes the reader with the concept of linear relaxations and shows how to work with them.

Apart from the initial examples, every example is presented as follows:

1. Example description
2. Formulation as a CP model
3. Implementation with Mosel using the *kalis* module: code listing and explanations
4. Results

All example models of this document are included with the set of examples that is provided as part of the Xpress Kalis distribution.

I. Working with Xpress Kalis in Mosel

CHAPTER 2

Modeling basics

This chapter shows how to

- start working with Mosel,
- create and solve a simple CP model using Xpress Kalis from Mosel,
- understand and analyze the output produced by the software,
- extend the model with data handling,
- define an objective function, and
- modify the default branching strategy.

2.1 A first model

Consider the following problem: we wish to schedule four meetings A, B, C, and D in three time slots (numbered 1 to 3). Some meetings are attended by the same persons, meaning that they may not take place at the same time: meeting A cannot be held at the same time as B or D, and meeting B cannot take the same time slot as C or D.

More formally, we may write down this problem as follows, where $\mathrm{plan}_{\mathrm{m}}$ ($\mathrm{m} \in \mathrm{MEETINGS} = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}\}$) denotes the time slot for meeting m—these are the *decision variables* of our problem.

$$\forall \mathrm{m} \in \mathrm{MEETINGS} : \mathrm{plan}_{\mathrm{m}} \in \{1, 2, 3\}$$
$$\mathrm{plan}_{\mathrm{A}} \neq \mathrm{plan}_{\mathrm{B}}$$
$$\mathrm{plan}_{\mathrm{A}} \neq \mathrm{plan}_{\mathrm{D}}$$
$$\mathrm{plan}_{\mathrm{B}} \neq \mathrm{plan}_{\mathrm{C}}$$
$$\mathrm{plan}_{\mathrm{B}} \neq \mathrm{plan}_{\mathrm{D}}$$

2.1.1 Implementation with Mosel

The following code listing implements and solves the problem described above.

```
model "Meeting"
 uses "kalis"

 declarations
  MEETINGS = {'A','B','C','D'}        ! Set of meetings
  TIME = 1..3                         ! Set of time slots
  plan: array(MEETINGS) of cpvar      ! Time slot per meeting
 end-declarations
```

```
 forall(m in MEETINGS) setdomain(plan(m), TIME)

! Respect incompatibilities
 plan('A') <> plan('B')
 plan('A') <> plan('D')
 plan('B') <> plan('C')
 plan('B') <> plan('D')

! Solve the problem
 if not cp_find_next_sol then
  writeln("Problem is infeasible")
  exit(1)
 end-if

! Solution printing
 forall(m in MEETINGS)
  writeln("Meeting ", m, ": ", getsol(plan(m)))

end-model
```

This Mosel model is saved as a text file with the name `meeting.mos`. Let us now take a closer look at what we have just written.

2.1.1.1 General structure

Every Mosel program starts with the keyword `model`, followed by a model name chosen by the user. The Mosel program is terminated with the keyword `end-model`.

As Mosel is itself not a solver, we specify that the Kalis constraint solver is to be used with the statement

```
 uses "kalis"
```

at the begin of the model.

All objects must be declared in a `declarations` section, unless they are defined unambiguously through an assignment. For example, `i := 1` defines `i` as an integer and assigns to it the value 1. There may be several such `declarations` sections at different places in a model.

In the present case, we define two *sets*, and one array:

- `MEETINGS` is a *set of strings*.
- `TIME` is a so-called *range set*—i.e., a set of consecutive integers (here: from 1 to 3).
- `plan` is an array of decision variables of type `cpvar` (finite domain CP variables; a second decision variable type of *kalis* is `cpfloatvar` for continuous variables), indexed by the set `MEETINGS`.

The model then defines the domains of the variables using the *kalis* procedure `setdomain`. The decision variables are indeed created at their declaration with a large default domain and `setdomain` reduces these domains to the intersection of the default domain with the indicated values. As in the mathematical model, we use a `forall` *loop* to enumerate all the indices in the set `MEETINGS`.

This is followed by the statement of the constraints, in this model we have four disequality constraints.

2.1.1.2 Solving and solution output

With the function `cp_find_next_sol`, we call Kalis to solve the problem (find a feasible assignment of values to all decision variables). We test the return value of this function: if no solution is found it returns `false` and we stop the model execution at this point (by calling the Mosel procedure `exit`), otherwise we print out the solution.

To solve the problem Xpress Kalis uses its built-in default search strategies. We shall see later how to modify these strategies.

The solution for a CP variable is obtained with the function `getsol`. To write several items on a single line use `write` instead of `writeln` for printing the output.

2.1.1.3 Formatting

Indentation, spaces, and empty lines in our model have been added to increase readability. They are skipped by Mosel.

Line breaks: It is possible to place several statements on a single line, separating them by semicolons, as such:

```
plan('A') <> plan('B'); plan('A') <> plan('D')
```

But since there are no special 'line end' or continuation characters, every line of a statement that continues over several lines must end with an operator (+, >=, etc.) or characters like ', ' that make it obvious that the statement is not terminated.

As shown in the example, single line *comments* in Mosel are preceded by `!`. Multiple line comments start with `(!` and terminate with `!)`.

2.1.2 Running the model

You may choose among three different methods for running your Mosel models:

1. From the *Mosel command line*: this method can be used on all platforms for which Mosel is available. It is especially useful if you wish to execute a (batch) sequence of model runs—for instance, with different parameter settings. The full Mosel functionality, including its debugger, is accessible in this run mode.
2. Within the graphical environment *Xpress Workbench*: available to Windows users. Workbench is a complete modeling and optimization development environment with a built-in text editor for working with Mosel models and a number of displays that help analyze models and solution algorithms in the development phase. Models can be modified and re-run interactively.
3. From within an *application program*: Mosel models may be executed and accessed from application programs (C/C++, Java, VBA, C#). This functionality is typically used for the deployment of Mosel models, integrating them into a company's information system.

In this manual we shall use the first two methods for running the models we develop. For further detail on embedding models into application programs the user is referred to the Mosel User Guide.

2.1.2.1 Working from the Mosel command line

When you have entered the complete model into the file `meeting.mos`, we can proceed to the solution to our problem. We start Mosel at the command prompt by typing the following command

```
mosel execute meeting.mos
```

and we will see output something like that below.

```
Meeting A: 1
Meeting B: 2
Meeting C: 1
Meeting D: 3
```

The Mosel command for executing the model can be abbreviated to

```
mosel exec meeting
```

or simply

```
mosel meeting
```

The model execution performed by the command `execute` comprises three stages:

1. Compiling `meeting.mos`
2. Loading the compiled model
3. Running the model we have just loaded.

Instead of using `execute`, you may choose to explicitly generate the compiled model file `chess.bim`

```
mosel compile meeting.mos
```

followed by

```
mosel run meeting.bim
```

to load and run the compiled model.

2.1.2.2 Using Xpress Workbench

To execute the model file `meeting.mos` with Workbench you need to carry out the following steps.

- Start up Workbench: if you have followed the standard installation procedure for Xpress Workbench, start the program by double clicking the icon on the desktop or selecting *Start* ≫ *Programs* ≫ *Xpress* ≫ *Xpress Workbench*. Otherwise, you may also start up Workbench by double clicking a model file (file with extension `.mos`).
- Open the model file by choosing *File* ≫ *Open*. The model source is then displayed in the central window (the *Workbench Editor*).
- Click the *Run* button or, alternatively, choose menu entry *Run* for the desired model.

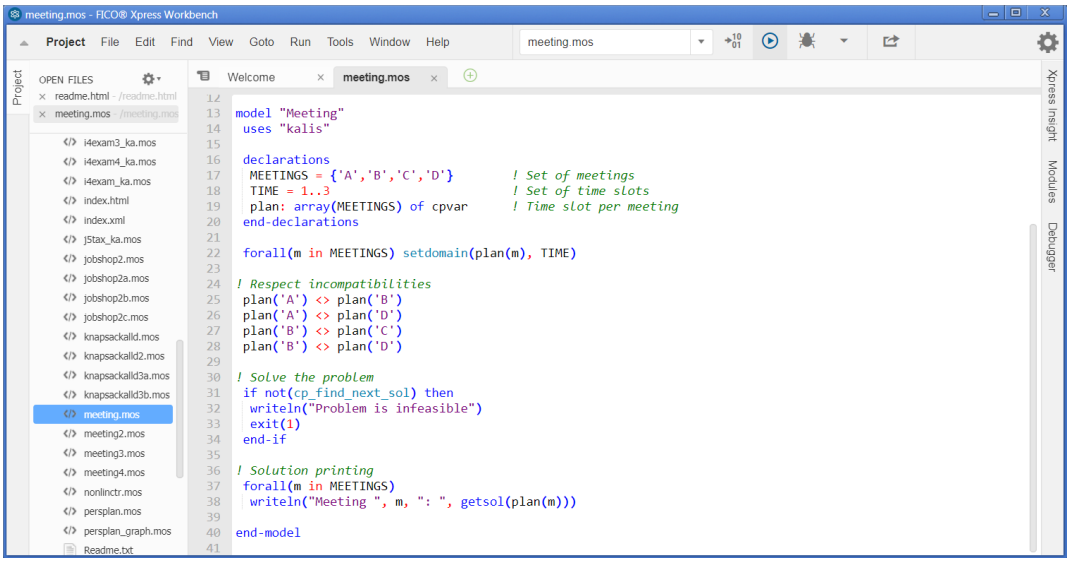

Figure 2.1: Workbench workspace after opening a model

The *Build* pane at the bottom of the workspace displays the model execution status messages from Mosel and any output generated by it. If syntax errors are found in the model they are displayed here,

with details of the line and character position where the error was detected and a description of the problem, if available.

Workbench makes all information about the model available for inspection through the *Debugger* pane in the right hand window if the model is run in debug mode (bug icon ).

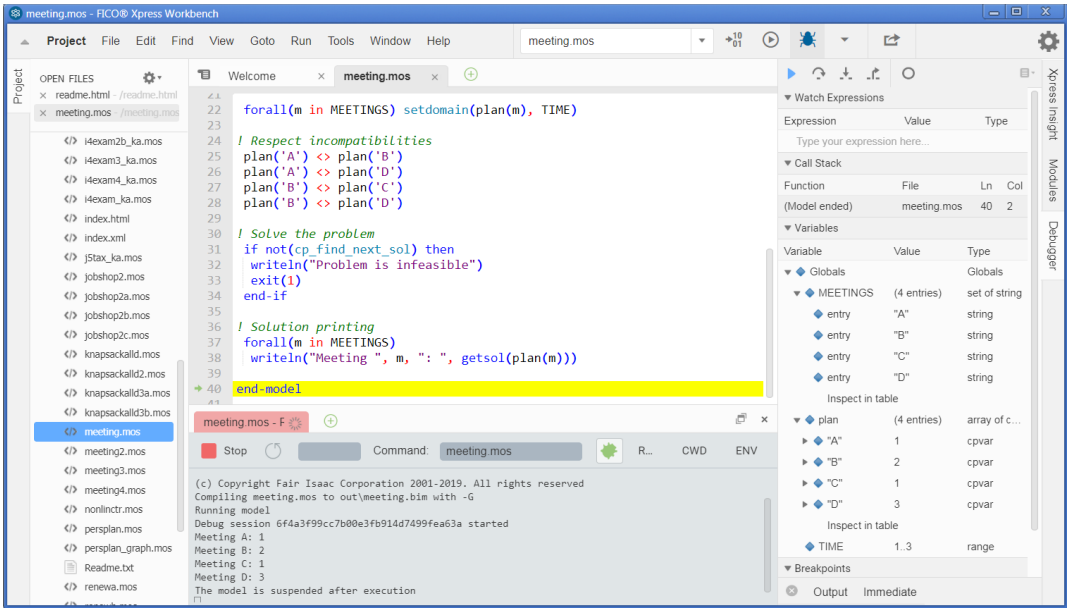

Figure 2.2: Workbench workspace after model execution in debug mode

2.1.2.3 Debugging a model

A first step for debugging a model certainly is to add additional output. In our model, we could, for instance, print out the definition of the decision variables by adding the line

```
writeln(plan)
```

after the definition of the variables' domains, or even print out the complete problem definition with the procedure `cp_show_prob`. To obtain a more easily readable output for debugging the user may give names to the decision variables of his problem. For example:

```
forall(m in MEETINGS) setname(plan(m), "Meeting "+m)
```

Notice that we have used the '+' sign to concatenate strings.

Calling the procedure `cp_show_stats` will display summary statistics of the CP solving.

To obtain detailed information about run-time errors with the command line version models need to be compiled with the flag `-g`, for example,

```
mosel exec -g meeting
```

For using the Mosel debugger (please see the Mosel Language Reference Manual for further detail) the compilation flag `-G` needs to be used instead.

Workbench by default compiles models in standard mode, debug+trace information (option `-G`) is automatically enabled when launching a debugging run (the debugger is started by clicking the 'debug' button  or via the *Debug* entry of the *Run* menu).

2.2 Data input from file

We now extend the previous example in order to use it with different data sets. If we wish to run a model

with different data sets, it would be impractical and error-prone having to edit the model file with every change of the data set. Instead, we are now going to see how to read data from a text file.

This is a description of the problem we want to solve (example taken from Section 14.4 of the book 'Applications of optimization with Xpress-MP').

A technical university needs to schedule the exams at the end of the term for a course with several optional modules. Every exam lasts two hours. Two days have been reserved for the exams with the following time periods: 8:00–10:00, 10:15–12:15, 14:00–16:00, and 16:15–18:15, resulting in a total of eight time periods. For every exam the set of incompatible exams that may not take place at the same time because they have to be taken by the same students is shown in Table 2.1.

Table 2.1: Incompatibilities between different exams

	DA	NA	C++	SE	PM	J	GMA	LP	MP	S	DSE
DA	–	X	–	–	X	–	X	–	–	X	X
NA	X	–	–	–	X	–	X	–	–	X	X
C++	–	–	–	X	X	X	X	–	X	X	X
SE	–	–	X	–	X	X	X	–	–	X	X
PM	X	X	X	X	–	X	X	X	X	X	X
J	–	–	X	X	X	–	X	–	X	X	X
GMA	X	X	X	X	X	X	–	X	X	X	X
LP	–	–	–	–	X	–	X	–	–	X	X
MP	–	–	X	–	X	X	X	–	–	X	X
S	X	X	X	X	X	X	X	X	X	–	X
DSE	X	X	X	X	X	X	X	X	X	X	–

2.2.1 Model formulation

The CP model has the same structure as the previous one, with the difference that we now introduce a data array INCOMP indicating incompatible pairs of exams and define the disequality constraints in a loop instead of writing them out one by one.

$$\forall e \in \mathrm{EXAM} : \mathrm{plan}_e \in \{1, \ldots, 8\}$$
$$\forall d, e \in \mathrm{EXAM}, \mathrm{INCOMP}_{de} = 1 : \mathrm{plan}_d \neq \mathrm{plan}_e$$

2.2.2 Implementation

The Mosel model now looks as follows.

```
model "I-4 Scheduling exams (CP)"
 uses "kalis"

 declarations
  EXAM = 1..11                        ! Set of exams
  TIME = 1..8                         ! Set of time slots
  INCOMP: array(EXAM,EXAM) of integer ! Incompatibility between exams
  EXAMNAME: array(EXAM) of string

  plan: array(EXAM) of cpvar          ! Time slot for exam
 end-declarations

 EXAMNAME:: (1..11)["DA","NA","C++","SE","PM","J","GMA","LP","MP","S","DSE"]

 initializations from 'Data/i4exam.dat'
  INCOMP
 end-initializations

 forall(e in EXAM) setdomain(plan(e), TIME)
```

```
! Respect incompatibilities
 forall(d,e in EXAM | d<e and INCOMP(d,e)=1)  plan(d) <> plan(e)

! Solve the problem
 if not cp_find_next_sol then
  writeln("Problem is infeasible")
  exit(1)
 end-if

! Solution printing
 forall(t in TIME) do
  write("Slot ", t, ": ")
  forall(e in EXAM | getsol(plan(e))=t) write(EXAMNAME(e)," ")
  writeln
 end-do

end-model
```

The values of the array INCOMP are read in from the file `i4exam.dat` in an `initializations` block. In the definition of the disequality constraints we check the value of the corresponding array entry—conditions on the indices for loops, sums, and other aggregate operators are marked by a vertical bar.

The data file has the following contents.

```
INCOMP: [0 1 0 0 1 0 1 0 0 1 1
         1 0 0 0 1 0 1 0 0 1 1
         0 0 0 1 1 1 1 0 1 1 1
         0 0 1 0 1 1 1 0 0 1 1
         1 1 1 1 0 1 1 1 1 1 1
         0 0 1 1 1 0 1 0 1 1 1
         1 1 1 1 1 1 0 1 1 1 1
         0 0 0 0 1 0 1 0 0 1 1
         0 0 1 0 1 1 1 0 0 1 1
         1 1 1 1 1 1 1 1 1 0 1
         1 1 1 1 1 1 1 1 1 1 0]
```

2.2.3 Results

The model prints out the following results. Only the first seven time slots are used for scheduling exams.

```
Slot 1: DA C++ LP
Slot 2: NA SE MP
Slot 3: PM
Slot 4: GMA
Slot 5: S
Slot 6: DSE
Slot 7: J
Slot 8:
```

2.2.4 Data-driven model

In the model shown above, we have read the incompatibility data from a file but part of the data (namely the index set EXAM and the number of time slots available) are still hard-coded in the model. To obtain a fully flexible model that can be run with arbitrary data sets we need to move all data definitions from the model to the data file.

The new data file `i4exam2.dat` not only defines the data entries, it also defines the index tuples for the array INCOMP. Every data entry is preceded by its index tuple (in brackets). There is no need to write out explicitly the contents of the set EXAM—Mosel will automatically populate this set with the index values read in for the array INCOMP. In addition, the data file now contains a value for the number of time periods NT.

```
INCOMP: [("DA" "NA") 1 ("DA" "PM") 1 ("DA" "GMA") 1 ("DA" "S") 1 ("DA" "DSE") 1
         ("NA" "DA") 1 ("NA" "PM") 1 ("NA" "GMA") 1 ("NA" "S") 1 ("NA" "DSE") 1
         ("C++" "SE") 1 ("C++" "PM") 1 ("C++" "J") 1 ("C++" "GMA") 1
  ("C++" "MP") 1 ("C++" "S") 1 ("C++" "DSE") 1
         ("SE" "C++") 1 ("SE" "PM") 1 ("SE" "J") 1 ("SE" "GMA") 1 ("SE" "S") 1
  ("SE" "DSE") 1
         ("PM" "DA") 1 ("PM" "NA") 1 ("PM" "C++") 1 ("PM" "SE") 1 ("PM" "J") 1
  ("PM" "GMA") 1 ("PM" "LP") 1 ("PM" "MP") 1 ("PM" "S") 1 ("PM" "DSE") 1
         ("J" "C++") 1 ("J" "SE") 1 ("J" "PM") 1 ("J" "GMA") 1 ("J" "MP") 1
  ("J" "S") 1 ("J" "DSE") 1
         ("GMA" "DA") 1 ("GMA" "NA") 1 ("GMA" "C++") 1 ("GMA" "SE") 1
  ("GMA" "PM") 1 ("GMA" "J") 1 ("GMA" "LP") 1 ("GMA" "MP") 1
  ("GMA" "S") 1 ("GMA" "DSE") 1
         ("LP" "PM") 1 ("LP" "GMA") 1 ("LP" "S") 1 ("LP" "DSE") 1
         ("MP" "C++") 1 ("MP" "PM") 1 ("MP" "J") 1 ("MP" "GMA") 1 ("MP" "S") 1
  ("MP" "DSE") 1
         ("S" "DA") 1 ("S" "NA") 1 ("S" "C++") 1 ("S" "SE") 1 ("S" "PM") 1
  ("S" "J") 1 ("S" "GMA") 1 ("S" "LP") 1 ("S" "MP") 1 ("S" "DSE") 1
         ("DSE" "DA") 1 ("DSE" "NA") 1 ("DSE" "C++") 1 ("DSE" "SE") 1
  ("DSE" "PM") 1 ("DSE" "J") 1 ("DSE" "GMA") 1 ("DSE" "LP") 1
  ("DSE" "MP") 1 ("DSE" "S") 1 ]

NT: 8
```

Our model also needs to undergo a few changes: the sets EXAM and TIME are now declared by stating their types, which turns them into *dynamic sets* (as opposed to their previous constant definition by stating their values). As a consequence, the array of decision variables `plan` is declared before the indexing set EXAM is known and Mosel creates this array as a *dynamic array*, meaning that the declaration results in an empty array and its elements need to be created explicitly (using the Mosel procedure `create`) once the indices are known. Before creating the variables, we modify the default bounds of Xpress Kalis to the values corresponding to the set TIME, thus replacing the call to `setdomain`.

The array INCOMP is explicitly declared as a dynamic array. The `initializations` block will assign values to just those entries that are listed in the data file (with the previous, constant declaration, all entries were defined) and the explicit `dynamic` marker means that Mosel's automatic finalization is not applied—it would try to transform the array into a static array. This makes it possible to reformulate the condition on the loop defining the disequality constraints: we now simply test for the existence of an entry instead of comparing all data values. With larger data sets, using the keyword `exists` may greatly reduce the execution time of loops involving sparse arrays (multidimensional data arrays with few entries different from 0).

```
model "I-4 Scheduling exams (CP) - 2"
 uses "kalis"

 declarations
  NT: integer                          ! Number of time slots
  EXAM: set of string                  ! Set of exams
  TIME: set of integer                 ! Set of time slots
  INCOMP: dynamic array(EXAM,EXAM) of integer ! Incompatibility between exams

  plan: array(EXAM) of cpvar           ! Time slot for exam
 end-declarations

 initializations from 'Data/i4exam2.dat'
  INCOMP NT
 end-initializations

 TIME:= 1..NT

 setparam("kalis_default_lb", 1); setparam("kalis_default_ub", NT)
 forall(e in EXAM) create(plan(e))

! Respect incompatibilities
 forall(d,e in EXAM | exists(INCOMP(d,e)) and d<e)  plan(d) <> plan(e)
```

```
! Solve the problem
 if not cp_find_next_sol then
  writeln("Problem is infeasible")
  exit(1)
 end-if

! Solution printing
 forall(t in TIME) do
  write("Slot ", t, ": ")
  forall(e in EXAM | getsol(plan(e))=t) write(e," ")
  writeln
 end-do

end-model
```

Running this fully data-driven model produces the same solution as the previous version.

An alternative to the explicit creation of the decision variables plan is to move their declaration after the initialization of the data as shown in the code extract below. In this case, it is important to `finalize` the indexing set EXAM, which turns it into a constant set with its current contents and allows Mosel to create any subsequently declared arrays indexed by this set as static arrays.

```
 declarations
  NT: integer                          ! Number of time slots
  EXAM: set of string                  ! Set of exams
  TIME: set of integer                 ! Set of time slots
  INCOMP: dynamic array(EXAM,EXAM) of integer ! Incompatibility between exams
 end-declarations

 initializations from 'Data/i4exam2.dat'
  INCOMP NT
 end-initializations

 finalize(EXAM)
 TIME:= 1..NT

 setparam("kalis_default_lb", 1); setparam("kalis_default_ub", NT)
 declarations
  plan: array(EXAM) of cpvar           ! Time slot for exam
 end-declarations
```

2.3 Optimization and enumeration

2.3.1 Optimization

Since running our model `i4exam_ka.mos` in Section 2.2.2 has produced a solution to the problem that does not use all time slots one might wonder which is the minimum number of time slots that are required for this problem. This question leads us to the formulation of an *optimization problem*.

We introduce a new decision variable $\mathrm{numslot}$ over the same value range as the plan_i variables and add the constraints that this variable is greater or equal to every plan_i variable. A simplified formulation is to say that the variable $\mathrm{numslot}$ equals the *maximum value* of all plan_i variables.

The objective then is to minimize the value of $\mathrm{numslot}$, which results in the following model.

```
model "I-4 Scheduling exams (CP) - 3"
 uses "kalis"

 declarations
  NT: integer                          ! Number of time slots
  EXAM: set of string                  ! Set of exams
  TIME: set of integer                 ! Set of time slots
  INCOMP: dynamic array(EXAM,EXAM) of integer ! Incompatibility between exams

  plan: array(EXAM) of cpvar           ! Time slot for exam
```

```
  numslot: cpvar                        ! Number of time slots used
 end-declarations

 initializations from 'Data/i4exam2.dat'
  INCOMP NT
 end-initializations

 finalize(EXAM)
 TIME:= 1..NT

 setparam("kalis_default_lb", 1); setparam("kalis_default_ub", NT)
 forall(e in EXAM) create(plan(e))

! Respect incompatibilities
 forall(d,e in EXAM | exists(INCOMP(d,e)) and d<e)  plan(d) <> plan(e)

! Calculate number of time slots used
 numslot = maximum(plan)

! Solve the problem
 if not cp_minimize(numslot) then
  writeln("Problem is infeasible")
  exit(1)
 end-if

! Solution printing
 forall(t in TIME) do
  write("Slot ", t, ": ")
  forall(e in EXAM | getsol(plan(e))=t) write(e," ")
  writeln
 end-do

end-model
```

Instead of `cp_find_next_sol` we now use `cp_minimize` with the objective function variable `numslot` as function argument.

This program also generates a solution using seven time slots, thus proving that this is the least number of slots required to produce a feasible schedule.

2.3.2 Enumeration

When comparing the problem statistics obtained by adding a call to `cp_show_stats` to the end of the different versions of our model, we can see that switching from finding a feasible solution to optimization considerably increases the number of nodes explored by the CP solver.

So far we have simply relied on the default enumeration strategies of Xpress Kalis. We shall now try to see whether we can reduce the number of nodes explored and hence shorten the time spent by the search for proving optimality.

The default strategy of Kalis for enumerating finite domain variables corresponds to adding the statement

```
cp_set_branching(assign_var(KALIS_SMALLEST_DOMAIN, KALIS_MIN_TO_MAX))
```

before the start of the search. `assign_var` denotes the *branching scheme* ('a branch is formed by assigning the next chosen value to the branching variable'), `KALIS_SMALLEST_DOMAIN` is the *variable selection strategy* ('choose the variable with the smallest number of values remaining in its domain'), and `KALIS_MIN_TO_MAX` the *value selection strategy* ('from smallest to largest value').

Since we are minimizing the number of time slots, enumeration starting with the smallest value seems to be a good idea. We therefore keep the default value selection criterion. However, we may try to change the variable selection heuristic: replacing `KALIS_SMALLEST_DOMAIN` by `KALIS_MAX_DEGREE` results in a reduction of the tree size and search time to less than half of its default size.

Here follows once more the complete model.

```
model "I-4 Scheduling exams (CP) - 4"
 uses "kalis"

 declarations
  NT: integer                          ! Number of time slots
  EXAM: set of string                  ! Set of exams
  TIME: set of integer                 ! Set of time slots
  INCOMP: dynamic array(EXAM,EXAM) of integer ! Incompatibility between exams

  plan: array(EXAM) of cpvar           ! Time slot for exam
  numslot: cpvar                       ! Number of time slots used
 end-declarations

 initializations from 'Data/i4exam2.dat'
  INCOMP NT
 end-initializations

 finalize(EXAM)
 TIME:= 1..NT

 setparam("kalis_default_lb", 1); setparam("kalis_default_ub", NT)
 forall(e in EXAM) create(plan(e))

! Respect incompatibilities
 forall(d,e in EXAM | exists(INCOMP(d,e)) and d<e)  plan(d) <> plan(e)

! Calculate number of time slots used
 numslot = maximum(plan)

! Setting parameters of the enumeration
 cp_set_branching(assign_var(KALIS_MAX_DEGREE, KALIS_MIN_TO_MAX))

! Solve the problem
 if not cp_minimize(numslot) then
  writeln("Problem is infeasible")
  exit(1)
 end-if

! Solution printing
 forall(t in TIME) do
  write("Slot ", t, ": ")
  forall(e in EXAM | getsol(plan(e))=t) write(e," ")
  writeln
 end-do

 cp_show_stats

end-model
```

NB: In the model versions without optimization we may try to obtain a more evenly distributed schedule by choosing values randomly, that is, by using the value selection criterion `KALIS_RANDOM_VALUE` instead of `KALIS_MIN_TO_MAX`.

Further detail on the definition of branching strategies is given in Chapter 4.

2.4 Continuous variables

All through this chapter we have worked with the decision variable type `cpvar` (discrete variables). A second variable type in Xpress Kalis are continuous variables (Mosel type `cpfloatvar`). Such variables are used in a similar way to what we have seen above for discrete variables, for example:

```
 setparam("KALIS_DEFAULT_CONTINUOUS_LB", 0)
 setparam("KALIS_DEFAULT_CONTINUOUS_UB", 10)
```

```
declarations
 x,y: cpfloatvar
end-declarations

x >= y                    ! Define a constraint
                          ! Retrieve information about continuous variables
writeln(getname(x), ":", getsol(x))
writeln(getlb(y), " ", getub(y))
```

A few differences in the use of the two decision variable types exist:

- Constraints involving `cpfloatvar` cannot be strict inequalities (that is, only the operators <=, >=, and = may be used).
- Most global constraints (see Chapter 3) only apply to `cpvar`; also applicable to `cpfloatvar` are `maximum` and `minimum` relations.
- Search strategies enumerating the values in a variable's domain can only be used with `cpvar`; with `cpfloatvar` domain splitting must be used (see Chapter 4).
- Access functions for enumerating domain values such `getnext` are not applicable to `cpfloatvar`.

CHAPTER 3

Constraints

This chapter contains a collection of examples demonstrating the use of Xpress Kalis from Mosel for solving different types of (optimization) problems. The first section shows different ways of defining and posting constraints for simple linear constraints. The following sections each introduce a new constraint type. Since most examples use a combination of different constraints, the following list may help in finding examples of the use of a certain constraint type quickly.

- arithmetic: linear disequality: 2.1 (`meeting.mos`), 2.2 (`i4exam*.mos`), 3.7 (`persplan.mos`); linear equality/inequality: 3.5 (`b4seq*_ka.mos`), 3.6 (`i1assign_ka.mos`), 3.8 (`b5paint*_ka.mos`), 3.9 (`j5tax_ka.mos`), 4.4 (`i1assign_ka.mos`), 5.2 (`b1stadium_ka.mos`); nonlinear: 3.2
- `all_different`: 3.3 (`sudoku_ka.mos`), 3.5 (`b4seq_ka.mos`), 3.8 (`b5paint*_ka.mos`), 3.4 (`freqasgn.mos`), 3.7 (`persplan.mos`), 3.11 (`eulerkn*.mos`), 4.4 (`i1assign_ka.mos`)
- `abs` / `distance`: 3.4 (`freqasgn.mos`)
- `cumulative`: 5.4 (`d4backup2_ka.mos`)
- `cycle`: 3.10 (`b5paint3_ka.mos`)
- `disjunctive`: 3.5 (`b4seq2_ka.mos`)
- `element`: 3.5 (`b4seq_ka.mos`), 3.6 (`a4sugar_ka.mos`), 3.9 (`j5tax_ka.mos`), 3.8 (`b5paint_ka.mos`), 4.4 (`i1assign_ka.mos`), 2D: 3.8 (`b5paint2_ka.mos`)
- `distribute` / `occurrence`: 3.6 (`a4sugar_ka.mos`), 3.7 (`persplan.mos`), 3.9 (`j5tax_ka.mos`), 3.12 (`scenesalloc.mos`)
- `maximum` / `minimum`: 2.3 (`i4exam3.mos`, `i4exam4.mos`), 3.5 (`b4seq2_ka.mos`), 3.4 (`freqasgn.mos`), 4.4 (`i1assign_ka.mos`), 3.12 (`scenesalloc.mos`)
- `implies` / `equiv`: 3.8 (`b5paint_ka.mos`), 3.7 (`persplan.mos`), 3.9 (`j5tax_ka.mos`), 3.11 (`eulerkn2.mos`), 3.12 (`scenesalloc.mos`)
- generic binary: 3.11 (`eulerkn.mos`)

3.1 Constraint handling

In this section we shall work once more with the introductory problem of scheduling meetings from Section 2.1. This model only contains simple arithmetic constraints over discrete variables. However, all that is said here equally applies to all other constraint types of Xpress Kalis, including constraint relations over continuous decision variables (that is, variables of the type `cpfloatvar`).

We now wish to state a few more constraints for this problem.

1. Meeting B must take place before day 3.
2. Meeting D cannot take place on day 2.
3. Meeting A must be scheduled on day 1.

3.1.1 Model formulation

The three additional constraints translate into simple (unary) linear constraints. We complete our model as follows:

$$\forall m \in \text{MEETINGS} : \text{plan}_m \in \{1, 2, 3\}$$
$$\text{plan}_B \leqslant 2$$
$$\text{plan}_D \neq 2$$
$$\text{plan}_A = 1$$
$$\text{plan}_A \neq \text{plan}_B$$
$$\text{plan}_A \neq \text{plan}_D$$
$$\text{plan}_B \neq \text{plan}_C$$
$$\text{plan}_B \neq \text{plan}_D$$

3.1.2 Implementation

The Mosel implementation of the new constraints is quite straightforward.

```
model "Meeting (2)"
 uses "kalis"

 declarations
  MEETINGS = {'A','B','C','D'}        ! Set of meetings
  TIME = 1..3                         ! Set of time slots
  plan: array(MEETINGS) of cpvar      ! Time slot per meeting
 end-declarations

 forall(m in MEETINGS) do
  setdomain(plan(m), TIME)
  setname(plan(m), "plan"+m)
 end-do
 writeln("Original domains: ", plan)

 plan('B') <= 2                      ! Meeting B before day 3
 plan('D') <> 2                      ! Meeting D not on day 2
 plan('A') = 1                       ! Meeting A on day 1
 writeln("With constraints: ", plan)

! Respect incompatibilities
 plan('A') <> plan('B')
 plan('A') <> plan('D')
 plan('B') <> plan('C')
 plan('B') <> plan('D')

! Solve the problem
 if not cp_find_next_sol then
  writeln("Problem is infeasible")
  exit(1)
 end-if

! Solution printing
 forall(m in MEETINGS)
  writeln("Meeting ", m, ": ", getsol(plan(m)))

end-model
```

3.1.3 Results

As the reader may have noticed, we have added printout of the variables $\mathrm{plan_m}$ at several places in the model. The output generated by the execution of this model therefore is the following.

```
Original domains: [planA[1..3],planB[1..3],planC[1..3],planD[1..3]]
With constraints: [planA[1],planB[1..2],planC[1..3],planD[1,3]]
Meeting A: 1
Meeting B: 2
Meeting C: 1
Meeting D: 3
```

As can be seen from this output, immediately after stating the constraints, the domains of the concerned variables have been reduced. The constraints are *immediately and automatically posted* to the solver and their effects are propagated to the whole problem.

3.1.4 Naming constraints

Sometimes it may be necessary to access constraints later on, after their definition, in particular if they are to become part of logic relations or if they are to be branched on (typically the case for disjunctive constraints). To this aim *kalis* defines a new type, `cpctr`, that can be used to *declare constraints* giving them a name that can used after their definition. We *define constraints* by assigning to them a constraint relation.

Naming constraints has the secondary effect that the constraints are *not automatically posted* to the solver. This needs to be done by writing the name of a constraint as a statement on its own (after its definition) as shown in the following model.

```
model "Meeting (3)"
 uses "kalis"

 declarations
  MEETINGS = {'A','B','C','D'}        ! Set of meetings
  TIME = 1..3                         ! Set of time slots
  plan: array(MEETINGS) of cpvar      ! Time slot per meeting
  Ctr: array(range) of cpctr
 end-declarations

 forall(m in MEETINGS) do
  setdomain(plan(m), TIME)
  setname(plan(m), "plan"+m)
 end-do
 writeln("Original domains: ", plan)

 Ctr(1):= plan('B') <= 2             ! Meeting B before day 3
 Ctr(2):= plan('D') <> 2             ! Meeting D not on day 2
 Ctr(3):= plan('A') = 1              ! Meeting A on day 1
 writeln("After definition of constraints:\n ", plan)

 forall(i in 1..3) Ctr(i)
 writeln("After posting of constraints:\n ", plan)

! Respect incompatibilities
 plan('A') <> plan('B')
 plan('A') <> plan('D')
 plan('B') <> plan('C')
 plan('B') <> plan('D')

! Solve the problem
 if not cp_find_next_sol then
  writeln("Problem is infeasible")
  exit(1)
 end-if
```

```
! Solution printing
 forall(m in MEETINGS) writeln("Meeting ", m, ": ", getsol(plan(m)))

end-model
```

From the output produced by this model, we can see that the mere definition of named constraints does not have any effect on the domains of the variables (the constraints are defined in Mosel but not yet sent to the Kalis solver). Only after stating the names of the constraints (that is, sending them to the solver) we obtain the same domain reductions as with the previous version of the model.

```
Original domains: [planA[1..3],planB[1..3],planC[1..3],planD[1..3]]
After definition of constraints:
 [planA[1..3],planB[1..3],planC[1..3],planD[1..3]]
After posting of constraints:
 [planA[1],planB[1..2],planC[1..3],planD[1,3]]
Meeting A: 1
Meeting B: 2
Meeting C: 1
Meeting D: 3
```

3.1.5 Explicit posting of constraints

In all previous examples, we have silently assumed that posting the constraints does not lead to a failure (infeasibility detected by the solver). In practice, this may not always be a reasonable assumption. *kalis* therefore defines the function `cp_post` that explicitly posts a constraint to the solver and returns the status of the constraint system after its addition (`true` for feasible and `false` for infeasible). This functionality may be of special interest for dealing with over-constrained problems to stop the addition of constraints once an infeasibility has been detected and to report back to the user which constraint has made the problem infeasible.

To use the explicit posting with `cp_post`, in the previous model we replace the line

```
 forall(i in 1..3) Ctr(i)
```

with the following code.

```
 forall(i in 1..3)
  if not cp_post(Ctr(i)) then
   writeln("Constraint ", i, " makes problem infeasible")
   exit(1)
  end-if
```

The return value of the every constraint posting is checked and the program is stopped if the addition of a constraint leads to an infeasibility.

The output produced by this model version is exactly the same as what we have seen in the previous section.

3.1.6 Explicit constraint propagation

The behavior of constraints can also be influenced in a different way. If we turn automated constraint propagation off,

```
 setparam("KALIS_AUTO_PROPAGATE", false)
```

then constraints posted to the solver are not propagated. In this case, constraint propagation will only be launched by a call to `cp_propagate` or by starting the enumeration (subroutines `cp_find_next_sol`, `cp_minimize`, *etc.*).

3.2 Arithmetic constraints

In the previous sections we have already seen several examples of linear constraints over finite domain variables. Linear constraints may be regarded as a special case of arithmetic constraints, that is, equations or inequality relations involving expressions over decision variables formed with the operators +, −, /, *,^, `sum`, `prod` and arithmetic functions like `abs` or `ln`. For a complete list of arithmetic functions supported by the solver the reader is refered to the Xpress Kalis Mosel Reference Manual.

Arithmetic constraints in Xpress Kalis may be defined over finite domain variables (type `cpvar`), continuous variables (type `cpfloatvar`), or mixtures of both. Notice, however, that arithmetic constraints involving continuous variables cannot be defined as strict inequalities, that means, only the relational operators >=, <=, and = may be used.

Here are a few examples of (nonlinear) arithmetic constraints that may be defined with Xpress Kalis in Mosel.

```
model "Nonlinear constraints"
 uses "kalis"

 setparam("KALIS_DEFAULT_LB", 0)
 setparam("KALIS_DEFAULT_UB", 5)
 setparam("KALIS_DEFAULT_CONTINUOUS_LB", -10)
 setparam("KALIS_DEFAULT_CONTINUOUS_UB", 10)

 declarations
  a,b,c: cpvar
  x,y,z: cpfloatvar
 end-declarations

 x = ln(y)
 y = abs(z)
 x*y <= z^2
 z = -a/b
 a*b*c^3 >= 150

 while (cp_find_next_sol)
  writeln("a:", getsol(a), ", b:", getsol(b), ", c:", getsol(c),
         ", x:", getsol(x), ", y:", getsol(y), ", z:", getsol(z))

end-model
```

3.3 `all_different`: Sudoku

Sudoku puzzles, originating from Japan, have recently made their appearance in many western newspapers. The idea of these puzzles is to complete a given, partially filled 9 × 9 board with the numbers 1 to 9 in such a way that no line, column, or 3 × 3 subsquare contains a number more than once. The tables 3.1 and 3.2 show two instances of such puzzles. Whilst sometimes tricky to solve for a human, these puzzles lend themselves to solving by a CP approach.

3.3.1 Model formulation

As in the examples, we denote the columns of the board by the set $XS = \{A, B, ..., I\}$ and the rows by $YS = \{1, 2, ..., 9\}$. For every x in XS and y in YS we define a decision variable v_{xy} taking as its value the number at the position (x, y).

The only constraints in this problem are
(1) all numbers in a row must be different,
(2) all numbers in a column must be different,
(3) all numbers in a 3 × 3 subsquare must be different.

Table 3.1: Sudoku ('The Times', 26 January, 2005)

	A	B	C	D	E	F	G	H	I
1					4	3		6	
2		6	5				7		
3	8			7					3
4		5				1	3		7
5	1	2						8	4
6	9		7	5				2	
7	4					5			9
8			9				4	5	
9		3		4	6				

Table 3.2: Sudoku ('The Guardian', 29 July, 2005)

	A	B	C	D	E	F	G	H	I
1	8					3			
2		5					4		
3	2				7			6	
4				1					5
5			3				9		
6	6					4			
7		7			2				3
8			4					1	
9				9					8

These constraints can be stated with Xpress Kalis's `all_different` relation. This constraint ensures that all variables in the relation take different values.

$$\forall x \in XS, y \in YS : v_{xy} \in \{1, ..., 9\}$$
$$\forall x \in XS : \text{all-different}(v_{x1}, ..., v_{x9})$$
$$\forall y \in YS : \text{all-different}(v_{Ay}, ..., v_{Iy})$$
$$\text{all-different}(v_{A1}, ..., v_{C3})$$
$$\text{all-different}(v_{A4}, ..., v_{C6})$$
$$\text{all-different}(v_{A7}, ..., v_{C9})$$
$$\text{all-different}(v_{D1}, ..., v_{F3})$$
$$\text{all-different}(v_{D4}, ..., v_{F6})$$
$$\text{all-different}(v_{D7}, ..., v_{F9})$$
$$\text{all-different}(v_{G1}, ..., v_{I3})$$
$$\text{all-different}(v_{G4}, ..., v_{I6})$$
$$\text{all-different}(v_{G7}, ..., v_{I9})$$

In addition, certain variables v_{xy} are fixed to the given values.

3.3.2 Implementation

The Mosel implementation for the Sudoku puzzle in Table 3.2 looks as follows.

```
model "sudoku (CP)"
 uses "kalis"

 forward procedure print_solution(numsol: integer)

 setparam("kalis_default_lb", 1)
 setparam("kalis_default_ub", 9)         ! Default variable bounds

 declarations
  XS = {'A','B','C','D','E','F','G','H','I'}   ! Columns
  YS = 1..9                              ! Rows
  v: array(XS,YS) of cpvar               ! Number assigned to cell (x,y)
 end-declarations

! Data from "The Guardian", 29 July, 2005. http://www.guardian.co.uk/sudoku
 v('A',1)=8; v('F',1)=3
 v('B',2)=5; v('G',2)=4
 v('A',3)=2; v('E',3)=7; v('H',3)=6
 v('D',4)=1; v('I',4)=5
 v('C',5)=3; v('G',5)=9
 v('A',6)=6; v('F',6)=4
 v('B',7)=7; v('E',7)=2; v('I',7)=3
 v('C',8)=4; v('H',8)=1
 v('D',9)=9; v('I',9)=8


! All-different values in rows
 forall(y in YS) all_different(union(x in XS) {v(x,y)})

! All-different values in columns
 forall(x in XS) all_different(union(y in YS) {v(x,y)})

! All-different values in 3x3 squares
 forall(s in {{'A','B','C'},{'D','E','F'},{'G','H','I'}}, i in 0..2)
  all_different(union(x in s, y in {1+3*i,2+3*i,3+3*i}) {v(x,y)})

! Solve the problem
 solct:= 0
```

```
 while (cp_find_next_sol) do
  solct+=1
  print_solution(solct)
 end-do

 writeln("Number of solutions: ", solct)
 writeln("Time spent in enumeration: ",
         getparam("KALIS_COMPUTATION_TIME"), "sec")
 writeln("Number of nodes: ", getparam("KALIS_NODES"))

!************************************************************
! Solution printing
 procedure print_solution(numsol: integer)
  writeln(getparam("KALIS_COMPUTATION_TIME"), "sec: Solution ", numsol)
  writeln("   A B C   D E F   G H I")
  forall(y in YS) do
   write(y, ": ")
   forall(x in XS)
    write(getsol(v(x,y)), if(x in {'C','F'}, " | ", " "))
   writeln
   if y mod 3 = 0 then
    writeln("   ---------------------")
   end-if
  end-do
 end-procedure

end-model
```

In this model, the call to `cp_find_next_sol` is embedded in a `while` loop to search all feasible solutions. At every loop execution the procedure `print_solution` is called to print out the solution found nicely formatted. Subroutines in Mosel may have `declarations` blocks for local declarations and they may take any number of arguments. Since, in our model, the call to the procedure occurs before its definition, we need to declare it before the first call using the keyword `forward`.

For selecting the information that is to be printed by the subroutine we use two different versions of Mosel's `if` statement: the inline `if` and `if-then` that includes a block of statements.

At the end of the model run we retrieve from the solver the run time measurement (parameter `KALIS_COMPUTATION_TIME`) and the number of nodes explored by the search (parameter `KALIS_NODES`).

To obtain a complete list of parameters defined by the *kalis* module type the command

```
mosel exam -p kalis
```

(for a listing of the complete functionality of *kalis* leave out the flag `-p`).

3.3.3 Results

The model shown above generates the following output; this puzzle has only one solution, as is usually the case for Sudoku puzzles.

```
0.16sec: Solution 1
   A B C   D E F   G H I
1: 8 6 9 | 2 4 3 | 1 5 7
2: 3 5 7 | 6 1 9 | 4 8 2
3: 2 4 1 | 8 7 5 | 3 6 9
   ---------------------
4: 4 9 8 | 1 3 2 | 6 7 5
5: 7 1 3 | 5 8 6 | 9 2 4
6: 6 2 5 | 7 9 4 | 8 3 1
   ---------------------
7: 1 7 6 | 4 2 8 | 5 9 3
8: 9 8 4 | 3 5 7 | 2 1 6
9: 5 3 2 | 9 6 1 | 7 4 8
   ---------------------
```

```
Number of solutions: 1
Time spent in enumeration: 0.41sec
Number of nodes: 2712
```

The `all_different` relation takes an optional second argument that allows the user to specify the *propagation algorithm* to be used for evaluating the constraint. If we change from the default setting (`KALIS_FORWARD_CHECKING`) to the more aggressive strategy `KALIS_GEN_ARC_CONSISTENCY` by adding this choice as the second argument, for example,

```
  forall(y in YS)
   all_different(union(x in XS) {v(x,y)}, KALIS_GEN_ARC_CONSISTENCY)
```

we observe that the number of nodes is reduced to a single node—the problem is solved by simply posting the constraints. Whereas the time spent in the search is down to zero, the constraint posting now takes 4-5 times longer (still just a fraction of a second) due to the larger computational overhead of the generalized arc consistency algorithm. Allover, the time for problem definition and solving is reduced to less than a tenth of the previous time.

As a general rule, the generalized arc consistency algorithm achieves stronger pruning (*i.e.*, it removes more values from the domains of the variables). However, due to the increase in computation time its use is not always justified. The reader is therefore encouraged to try both algorithm settings in his models.

3.4 `abs` and `distance`: Frequency assignment

The area of telecommunications, and in particular mobile telecommunications, gives rise to many different variants of frequency assignment problems.

We are given a network of cells (nodes) with requirements of discrete frequency bands. Each cell has a given demand for a number of frequencies (bands). Figure 3.1 shows the structure of the network. Nodes linked by an edge are considered as neighbors. They must not be assigned the same frequencies to avoid interference. Furthermore, if a cell uses several frequencies they must all be different by at least 2. The objective is to minimize the total number of frequencies used in the network.

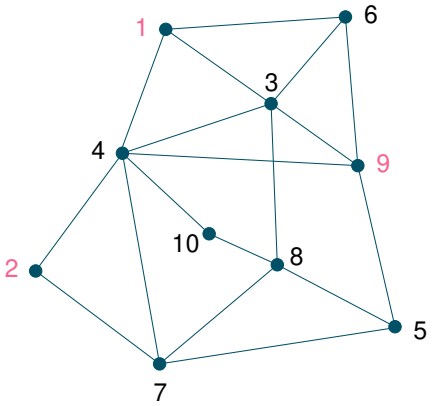

Figure 3.1: Telecommunications network

Table 3.3 lists the number of frequency demands for every cell.

Table 3.3: Frequency demands at nodes

Cell	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Demand	4	5	2	3	2	4	3	4	3	2

3.4.1 Model formulation

Let NODES be the set of all nodes in the network and DEM_n the demand of frequencies at node $n \in \mathrm{NODES}$. The network is given as a set of edges LINKS. Furthermore, let $\mathrm{DEMANDS} = \{1, 2, ..., \mathrm{NUMDEM}\}$ be the set of frequencies, numbered consecutively across all nodes where the upper bound NUMDEM is given by the total number of demands. The auxiliary array INDEX_n indicates the starting index in $\mathrm{DEMANDS}$ for node n.

For representing the frequency assigned to every demand $d \in \mathrm{DEMANDS}$ we introduce the variables use_d that take their values from the set $\{1, 2, ..., \mathrm{NUMDEM}\}$.

The two sets of constraints (different frequencies assigned to neighboring nodes and minimum distance between frequencies within a node) can then be modeled as follows.

$$\forall (n, m) \in \mathrm{LINKS} : \text{all-different}\left(\bigcup_{d=\mathrm{INDEX}_n}^{\mathrm{INDEX}_n+\mathrm{DEM}_n-1} use_d \cup \bigcup_{d=\mathrm{INDEX}_m}^{\mathrm{INDEX}_m+\mathrm{DEM}_m-1} use_d\right)$$

$$\forall n \in \mathrm{NODES}, \forall c < d \in \mathrm{INDEX}_n, ..., \mathrm{INDEX}_n + \mathrm{DEM}_n - 1 : |use_c - use_d| \geqslant 2$$

The objective function is to minimize to the number of frequencies used. We formulate this by minimizing the largest frequency number that occurs for the use_d variables:

$$\text{minimize } maximum_{d \in \mathrm{DEMANDS}}(use_d)$$

3.4.2 Implementation

The edges forming the telecommunications network are modeled as a list `LINK`, where edge `l` is given as (`LINK(l,1)`,`LINK(l,2)`).

For the implementation of the constraints on the values of frequencies assigned to the same node we have two equivalent choices with Kalis, namely using `abs` or `distance` constraints.

```
model "Frequency assignment"
 uses "kalis"

 forward procedure print_solution

 declarations
  NODES = 1..10                                ! Range of nodes
  LINKS = 1..18                                ! Range of links between nodes
  DEM: array(NODES) of integer                 ! Demand of nodes
  LINK: array(LINKS,1..2) of integer           ! Neighboring nodes
  INDEX: array(NODES) of integer               ! Start index in 'use'
  NUMDEM: integer                              ! Upper bound on no. of freq.
 end-declarations

 DEM :: (1..10)[4, 5, 2, 3, 2, 4, 3, 4, 3, 2]
 LINK:: (1..18,1..2)[1, 3, 1, 4, 1, 6,
                     2, 4, 2, 7,
                     3, 4, 3, 6, 3, 8, 3, 9,
                     4, 7, 4, 9, 4,10,
                     5, 7, 5, 8, 5, 9,
                     6, 9, 7, 8, 8,10]
 NUMDEM:= sum(n in NODES) DEM(n)

! Correspondence of nodes and demand indices:
! use(d) d = 1, ..., DEM(1) correspond to the demands of node 1
!            d = DEM(1)+1, ..., DEM(1)+DEM(2))     - " -     node 2  etc.
 INDEX(1):= 1
 forall(n in NODES | n > 1) INDEX(n) := INDEX(n-1) + DEM(n-1)

 declarations
  DEMANDS = 1..NUMDEM                          ! Range of frequency demands
```

```
  use: array(DEMANDS) of cpvar                ! Frequency used for a demand
  numfreq: cpvar                              ! Number of frequencies used
  Strategy: array(range) of cpbranching
 end-declarations

! Setting the domain of the decision variables
 forall(d in DEMANDS) setdomain(use(d), 1, NUMDEM)

! All frequencies attached to a node must be different by at least 2
 forall(n in NODES, c,d in INDEX(n)..INDEX(n)+DEM(n)-1 | c<d)
  distance(use(c), use(d)) >= 2
!  abs(use(c) - use(d)) >= 2

! Neighboring nodes take all-different frequencies
 forall(l in LINKS)
  all_different(
   union(d in INDEX(LINK(l,1))..INDEX(LINK(l,1))+DEM(LINK(l,1))-1) {use(d)} +
   union(d in INDEX(LINK(l,2))..INDEX(LINK(l,2))+DEM(LINK(l,2))-1) {use(d)},
   KALIS_GEN_ARC_CONSISTENCY)

! Objective function: minimize the number of frequencies used, that is,
! minimize the largest value assigned to 'use'
 setname(numfreq, "NumFreq")
 numfreq = maximum(use)

! Search strategy
 Strategy(1):=assign_var(KALIS_SMALLEST_DOMAIN, KALIS_MIN_TO_MAX, use)
 Strategy(2):=assign_var(KALIS_MAX_DEGREE, KALIS_MIN_TO_MAX, use)
 cp_set_branching(Strategy(1))
 setparam("KALIS_MAX_COMPUTATION_TIME", 1)
 cp_set_solution_callback(->print_solution)

! Try to find solution(s) with strategy 1
 if cp_minimize(numfreq) then
  cp_show_stats
  sol:=getsol(numfreq)
 end-if

! Restart search with strategy 2
 cp_reset_search
 if sol>0 then                               ! If a solution was found:
  numfreq <= sol-1                           ! Add upper bound on objective
 end-if
 cp_set_branching(Strategy(2))
 setparam("KALIS_MAX_COMPUTATION_TIME", 1000)

 if cp_minimize(numfreq) then
  cp_show_stats
 elif sol>0 then
  writeln("Optimality proven")
 else
  writeln("Problem has no solution")
 end-if

!*******************************************************************
! **** Solution printout ****
 procedure print_solution
  writeln("Number of frequencies: ", getsol(numfreq))
  writeln("Frequency assignment: ")
  forall(n in NODES) do
   write("Node ", n, ": ")
   forall(d in INDEX(n)..INDEX(n)+DEM(n)-1) write(getsol(use(d)), " ")
   writeln
  end-do
 end-procedure

end-model
```

With just the default search strategy this model finds a solution of value 11 but it runs for a long time

without being able to prove optimality. When experimenting with different search strategies we have found that the strategy obtained by changing the variable selection criterion to `KALIS_MAX_DEGREE` is able to prove optimality easily once a good solution is known. This problem is therefore solved in two steps: First, we use the default strategy for finding a good solution. This search is stopped after one second by setting a *time limit*. The search is then *restarted* (previously, we needed to reset the search tree in the solver with `cp_reset_search`) with a second strategy and the bound on the objective value from the previous run.

To ease the experiments with different search strategies we have defined an array `Strategy` of type `cpbranching` that stores the different search strategy definitions.

Another new feature demonstrated by this implementation is the use of a *callback*, more precisely the *solution callback* of *kalis*. The solution callback is defined with a user subroutine that will be called by the solver whenever the search has found a solution. Its typical uses are logging or storing of intermediate solutions or performing some statistics. Our procedure `print_solution` simply prints out the solution that has been found.

Improving the problem formulation: we may observe that in our problem formulation all demand variables within a node and the constraints on these variables are entirely *symmetric*. In the absence of other constraints, we may reduce these symmetries by imposing an order on the use variables,

$$\mathrm{use}_{\mathrm{d}} + 1 \leqslant \mathrm{use}_{\mathrm{d}+1}$$

for demands d and $\mathrm{d}+1$ belonging to the same cell. Doing so, the problem is solved to optimality within less than 40 nodes using just the default strategy. We may take this a step further by writing

$$\mathrm{use}_{\mathrm{d}} + 2 \leqslant \mathrm{use}_{\mathrm{d}+1}$$

The addition of these constraints shortens the search by yet a few more nodes. They can even be used simply in replacement of the `abs` or `distance` constraints.

3.4.3 Results

An optimal solution to this problem uses 11 different frequencies. The model shown in the program listing prints out the following assignment of frequencies to nodes:

```
Node 1: 1 3 5 7
Node 2: 1 3 5 7 10
Node 3: 2 8
Node 4: 4 6 9
Node 5: 4 6
Node 6: 4 6 9 11
Node 7: 2 8 11
Node 8: 1 3 5 7
Node 9: 1 3 5
Node 10: 2 8
```

3.5 `element`: Sequencing jobs on a single machine

The problem described in this section is taken from Section 7.4 'Sequencing jobs on a bottleneck machine' of the book 'Applications of optimization with Xpress-MP'

The aim of this problem is to provide a model that may be used with different objective functions for scheduling operations on a single (bottleneck) machine. We shall see here how to minimize the total processing time, the average processing time, and the total tardiness.

A set of tasks (or jobs) is to be processed on a single machine. The execution of tasks is non-preemptive (that is, an operation may not be interrupted before its completion). For every task i its release date, duration, and due date are given in Table 3.4.

Table 3.4: Task time windows and durations

Job	1	2	3	4	5	6	7
Release date	2	5	4	0	0	8	9
Duration	5	6	8	4	2	4	2
Due date	10	21	15	10	5	15	22

What is the optimal value for each of the objectives: minimizing the total duration of the schedule (*makespan*), the mean processing time or the total tardiness (that is, the amount of time by which the completion of jobs exceeds their respective due dates)?

3.5.1 Model formulation 1

We are going to present two alternative model formulations. The first is closer to the Mathematical Programming formulation in 'Applications of optimization with Xpress-MP'. The second uses disjunctive constraints and branching on these. In both model formulations we are going deal with the different objective functions in sequence, but the body of the models will remain the same.

To represent the sequence of jobs we introduce variables rank_k ($k \in \mathrm{JOBS} = \{1, ..., \mathrm{NJ}\}$) that take as value the number of the job in position (rank) k. Every job j takes a single position. This constraint can be represented by an *all-different* on the rank_k variables:

$$\text{all-different}(\mathrm{rank}_1, ..., \mathrm{rank}_{\mathrm{NJ}})$$

The processing time dur_k for the job in position k is given by $\mathrm{DUR}_{\mathrm{rank}_k}$ (where DUR_j denotes the duration given in the table in the previous section). Similarly, the release time rel_k is given by $\mathrm{REL}_{\mathrm{rank}_k}$ (where REL_j denotes the given release date):

$$\forall k \in \mathrm{JOBS} : \mathrm{dur}_k = \mathrm{DUR}_{\mathrm{rank}_k}$$
$$\forall k \in \mathrm{JOBS} : \mathrm{rel}_k = \mathrm{REL}_{\mathrm{rank}_k}$$

If start_k is the start time of the job at position k, this value must be at least as great as the release date of the job assigned to this position. The completion time comp_k of this job is the sum of its start time plus its duration:

$$\forall k \in \mathrm{JOBS} : \mathrm{start}_k \geqslant \mathrm{rel}_k$$
$$\forall k \in \mathrm{JOBS} : \mathrm{comp}_k = \mathrm{start}_k + \mathrm{dur}_k$$

Another constraint is needed to specify that two jobs cannot be processed simultaneously. The job in position $k + 1$ must start after the job in position k has finished, hence the following constraints:

$$\forall k \in \{1, ..., \mathrm{NJ} - 1\} : \mathrm{start}_{k+1} \geqslant \mathrm{start}_k + \mathrm{dur}_k$$

Objective 1: The first objective is to minimize the makespan (completion time of the schedule), or, equivalently, to minimize the completion time of the last job (job with rank NJ). The complete model is then given by the following (where $\mathrm{MAXTIME}$ is a sufficiently large value, such as the sum of all release

dates and all durations):

$$
\begin{aligned}
&\text{minimize } comp_{NJ}\\
&\forall k \in JOBS : rank_k \in JOBS\\
&\forall k \in JOBS : start_k, comp_k \in \{0, ..., MAXTIME\}\\
&\forall k \in JOBS : dur_k \in \{\min_{j\in JOBS} DUR_j, ..., \max_{j\in JOBS} DUR_j\}\\
&\forall k \in JOBS : rel_k \in \{\min_{j\in JOBS} REL_j, ..., \max_{j\in JOBS} REL_j\}\\
&all\text{-}different(rank_1, ..., rank_{NJ})\\
&\forall k \in JOBS : dur_k = DUR_{rank_k}\\
&\forall k \in JOBS : rel_k = REL_{rank_k}\\
&\forall k \in JOBS : start_k \geqslant rel_k\\
&\forall k \in JOBS : comp_k = start_k + dur_k\\
&\forall k \in \{1, ..., NJ - 1\} : start_{k+1} \geqslant start_k + dur_k
\end{aligned}
$$

Objective 2: For minimizing the average processing time, we introduce an additional variable $totComp$ representing the sum of the completion times of all jobs. We add the following constraint to the problem to calculate $totComp$:

$$totComp = \sum_{k\in JOBS} comp_k$$

The new objective consists of minimizing the average processing time, or equivalently, minimizing the sum of the job completion times:

$$\text{minimize } totComp$$

Objective 3: If we now aim to minimize the total tardiness, we again introduce new variables—this time to measure the amount of time that jobs finish after their due date. We write $late_k$ for the variable that corresponds to the tardiness of the job with rank k. Its value is the difference between the completion time of a job j and its due date DUE_j. If the job finishes before its due date, the value must be zero. We thus obtain the following constraints:

$$
\begin{aligned}
&\forall k \in JOBS : due_k = DUE_{rank_k}\\
&\forall k \in JOBS : late_k \geqslant comp_k - due_k
\end{aligned}
$$

For the formulation of the new objective function we introduce the variable $totLate$ representing the total tardiness of all jobs. The objective now is to minimize the value of this variable:

$$
\begin{aligned}
&\text{minimize } totLate\\
&totLate = \sum_{k\in JOBS} late_k
\end{aligned}
$$

3.5.2 Implementation of model 1

The Mosel implementation below solves the same problem three times, each time with a different objective, and prints the resulting solutions by calling the procedures `print_sol` and `print_sol3`.

```
model "B-4 Sequencing (CP)"
 uses "kalis"

 forward procedure print_sol
 forward procedure print_sol3
```

```
declarations
 NJ = 7                          ! Number of jobs
 JOBS=1..NJ

 REL: array(JOBS) of integer     ! Release dates of jobs
 DUR: array(JOBS) of integer     ! Durations of jobs
 DUE: array(JOBS) of integer     ! Due dates of jobs

 rank: array(JOBS) of cpvar      ! Number of job at position k
 start: array(JOBS) of cpvar     ! Start time of job at position k
 dur: array(JOBS) of cpvar       ! Duration of job at position k
 comp: array(JOBS) of cpvar      ! Completion time of job at position k
 rel: array(JOBS) of cpvar       ! Release date of job at position k
end-declarations

initializations from 'Data/b4seq.dat'
 DUR REL DUE
end-initializations

MAXTIME:= max(j in JOBS) REL(j) + sum(j in JOBS) DUR(j)
MINDUR:= min(j in JOBS) DUR(j); MAXDUR:= max(j in JOBS) DUR(j)
MINREL:= min(j in JOBS) REL(j); MAXREL:= max(j in JOBS) REL(j)

forall(j in JOBS) do
 1 <= rank(j); rank(j) <= NJ
 0 <= start(j); start(j) <= MAXTIME
 MINDUR <= dur(j); dur(j) <= MAXDUR
 0 <= comp(j); comp(j) <= MAXTIME
 MINREL <= rel(j); rel(j) <= MAXREL
end-do

! One posistion per job
all_different(rank)

! Duration of job at position k
forall(k in JOBS) dur(k) = element(DUR, rank(k))

! Release date of job at position k
forall(k in JOBS) rel(k) = element(REL, rank(k))

! Sequence of jobs
forall(k in 1..NJ-1) start(k+1) >= start(k) + dur(k)

! Start times
forall(k in JOBS) start(k) >= rel(k)

! Completion times
forall(k in JOBS) comp(k) = start(k) + dur(k)

! Set the branching strategy
cp_set_branching(split_domain(KALIS_SMALLEST_DOMAIN, KALIS_MIN_TO_MAX))

!**** Objective function 1: minimize latest completion time ****
if cp_minimize(comp(NJ)) then
 print_sol
end-if


!**** Objective function 2: minimize average completion time ****
declarations
 totComp: cpvar
end-declarations

totComp = sum(k in JOBS) comp(k)

if cp_minimize(totComp) then
 print_sol
end-if
```

```
!**** Objective function 3: minimize total tardiness ****
 declarations
  late: array(JOBS) of cpvar      ! Lateness of job at position k
  due: array(JOBS) of cpvar       ! Due date of job at position k
  totLate: cpvar
 end-declarations

 MINDUE:= min(k in JOBS) DUE(k); MAXDUE:= max(k in JOBS) DUE(k)

 forall(k in JOBS) do
  MINDUE <= due(k); due(k) <= MAXDUE
  0 <= late(k); late(k) <= MAXTIME
 end-do

! Due date of job at position k
 forall(k in JOBS) due(k) = element(DUE, rank(k))

! Late jobs: completion time exceeds the due date
 forall(k in JOBS) late(k) >= comp(k) - due(k)

 totLate = sum(k in JOBS) late(k)

 if cp_minimize(totLate) then
  writeln("Tardiness: ", getsol(totLate))
  print_sol
  print_sol3
 end-if

!-------------------------------------------------------------------

! Solution printing
 procedure print_sol
  writeln("Completion time: ", getsol(comp(NJ)) ,
          "  average: ", getsol(sum(k in JOBS) comp(k)))
  write("\t")
  forall(k in JOBS) write(strfmt(getsol(rank(k)),4))
  write("\nRel\t")
  forall(k in JOBS) write(strfmt(getsol(rel(k)),4))
  write("\nDur\t")
  forall(k in JOBS) write(strfmt(getsol(dur(k)),4))
  write("\nStart\t")
  forall(k in JOBS) write(strfmt(getsol(start(k)),4))
  write("\nEnd\t")
  forall(k in JOBS) write(strfmt(getsol(comp(k)),4))
  writeln
 end-procedure

 procedure print_sol3
  write("Due\t")
  forall(k in JOBS) write(strfmt(getsol(due(k)),4))
  write("\nLate\t")
  forall(k in JOBS) write(strfmt(getsol(late(k)),4))
  writeln
 end-procedure

end-model
```

NB: The reader may have been wondering why we did not use the more obvious pair `start` – `end` for naming the variables in this example: `end` is a keyword of the Mosel language (see the list of reserved words in the Mosel Language Reference Manual), which means that neither `end` nor `END` may be redefined by a Mosel program. It is possible though, to use versions combining lower and upper case letters, like `End`, but to prevent any possible confusion we do not recommend their use.

3.5.3 Results

The minimum makespan of the schedule is 31, the minimum sum of completion times is 103 (which gives an average of $103/7 = 14.71$). A schedule with this objective value is $5 \to 4 \to 1 \to 7 \to 6 \to 2 \to 3$. If we compare the completion times with the due dates we see that jobs 1, 2, 3, and 6 finish late (with a total tardiness of 21). The minimum tardiness is 18. A schedule with this tardiness is $5 \to 1 \to 4 \to 6 \to 2 \to 7 \to 3$ where jobs 4 and 7 finish one time unit late and job 3 is late by 16 time units, and it terminates at time 31 instead of being ready at its due date, time 15. This schedule has an average completion time of 15.71.

3.5.4 Alternative formulation using disjunctions

Our second model formulation is possibly a more straightforward way of representing the problem. It introduces disjunctive constraints and branching on these.

Every job is now represented by its starting time, variables start_j ($j \in \mathrm{JOBS} = \{1, ..., \mathrm{NJ}\}$) that take their values in $\{\mathrm{REL}_j, ..., \mathrm{MAXTIME}\}$ (where $\mathrm{MAXTIME}$ is a sufficiently large value, such as the sum of all release dates and all durations, and REL_j the release date of job j). We state the disjunctions as a single *disjunctive* relation on the start times and durations of all jobs.

$$\mathrm{disjunctive}([\mathrm{start}_1, ..., \mathrm{start}_{\mathrm{NJ}}], [\mathrm{DUR}_1, ..., \mathrm{DUR}_{\mathrm{NJ}}])$$

This constraint replaces the pair-wise disjunctions

$$\mathrm{start}_i + \mathrm{DUR}_i \leqslant \mathrm{start}_j \vee \mathrm{start}_j + \mathrm{DUR}_j \leqslant \mathrm{start}_i$$

for all pairs of jobs $i < j \in \mathrm{JOBS}$.

The processing time dur_k for the job in position k is given by $\mathrm{DUR}_{\mathrm{rank}_k}$ (where DUR_j denotes the duration given in the table in the previous section). Similarly, its release time rel_k is given by $\mathrm{REL}_{\mathrm{rank}_k}$ (where REL_j denotes the given release date).

$$\forall k \in \mathrm{JOBS} : \mathrm{dur}_k = \mathrm{DUR}_{\mathrm{rank}_k}$$
$$\forall k \in \mathrm{JOBS} : \mathrm{rel}_k = \mathrm{REL}_{\mathrm{rank}_k}$$

The completion time comp_j of a job j is the sum of its start time plus its duration DUR_j.

$$\forall j \in \mathrm{JOBS} : \mathrm{comp}_j = \mathrm{start}_j + \mathrm{DUR}_j$$

Objective 1: The first objective is to minimize the makespan (completion time of the schedule) or, equivalently, to minimize the completion time finish of the last job. The complete model is then given by the following (where $\mathrm{MAXTIME}$ is a sufficiently large value, such as the sum of all release dates and all durations):

$$\begin{aligned}
&\text{minimize } \mathrm{finish} \\
&\mathrm{finish} = \mathrm{maximum}_{j \in \mathrm{JOBS}}(\mathrm{comp}_j) \\
&\forall j \in \mathrm{JOBS} : \mathrm{comp}_j \in \{0, ..., \mathrm{MAXTIME}\} \\
&\forall j \in \mathrm{JOBS} : \mathrm{start}_j \in \{\mathrm{REL}_j, ..., \mathrm{MAXTIME}\} \\
&\mathrm{disjunctive}([\mathrm{start}_1, ..., \mathrm{start}_{\mathrm{NJ}}], [\mathrm{DUR}_1, ..., \mathrm{DUR}_{\mathrm{NJ}}]) \\
&\forall j \in \mathrm{JOBS} : \mathrm{comp}_j = \mathrm{start}_j + \mathrm{DUR}_j
\end{aligned}$$

Objective 2: The formulation of the second objective (minimizing the average processing time or, equivalently, minimizing the sum of the job completion times) remains unchanged from the first

model—we introduce an additional variable $\mathrm{totComp}$ representing the sum of the completion times of all jobs.

$$\begin{aligned}&\text{minimize } \mathrm{totComp}\\&\mathrm{totComp} = \sum_{k \in \mathrm{JOBS}} \mathrm{comp}_k\end{aligned}$$

Objective 3: To formulate the objective of minimizing the total tardiness, we introduce new variables late_j to measure the amount of time that a job finishes after its due date. The value of these variables corresponds to the difference between the completion time of a job j and its due date DUE_j. If the job finishes before its due date, the value must be zero. The objective now is to minimize the sum of these tardiness variables:

$$\begin{aligned}&\text{minimize } \mathrm{totLate}\\&\mathrm{totLate} = \sum_{j \in \mathrm{JOBS}} \mathrm{late}_j\\&\forall j \in \mathrm{JOBS} : \mathrm{late}_j \in \{0, ..., \mathrm{MAXTIME}\}\\&\forall j \in \mathrm{JOBS} : \mathrm{late}_j \geqslant \mathrm{comp}_j - \mathrm{DUE}_j\end{aligned}$$

3.5.5 Implementation of model 2

As with model 1, the Mosel implementation below solves the same problem three times, each time with a different objective, and prints the resulting solutions by calling the procedures `print_sol` and `print_sol3`.

```
model "B-4 Sequencing (CP)"
 uses "kalis"

 forward procedure print_sol
 forward procedure print_sol3

 declarations
  NJ = 7                          ! Number of jobs
  JOBS=1..NJ

  REL: array(JOBS) of integer     ! Release dates of jobs
  DUR: array(JOBS) of integer     ! Durations of jobs
  DUE: array(JOBS) of integer     ! Due dates of jobs
  DURS: array(set of cpvar) of integer  ! Dur.s indexed by start variables

  start: array(JOBS) of cpvar     ! Start time of jobs
  comp: array(JOBS) of cpvar      ! Completion time of jobs
  finish: cpvar                   ! Completion time of the entire schedule
  Disj: set of cpctr              ! Disjunction constraints
  Strategy: array(range) of cpbranching  ! Branching strategy
 end-declarations

 initializations from 'Data/b4seq.dat'
  DUR REL DUE
 end-initializations

 MAXTIME:= max(j in JOBS) REL(j) + sum(j in JOBS) DUR(j)

 forall(j in JOBS) do
  0 <= start(j); start(j) <= MAXTIME
  0 <= comp(j); comp(j) <= MAXTIME
 end-do

! Disjunctions between jobs
 forall(j in JOBS) DURS(start(j)):= DUR(j)
 disjunctive(union(j in JOBS) {start(j)}, DURS, Disj, 1)
```

```
! Start times
 forall(j in JOBS) start(j) >= REL(j)

! Completion times
 forall(j in JOBS) comp(j) = start(j) + DUR(j)

!**** Objective function 1: minimize latest completion time ****
 finish = maximum(comp)

 Strategy(1):= settle_disjunction(Disj)
 Strategy(2):= split_domain(KALIS_LARGEST_MAX, KALIS_MIN_TO_MAX)
 cp_set_branching(Strategy)

 if cp_minimize(finish) then
  print_sol
 end-if


!**** Objective function 2: minimize average completion time ****
 declarations
  totComp: cpvar
 end-declarations

 totComp = sum(k in JOBS) comp(k)

 if cp_minimize(totComp) then
  print_sol
 end-if


!**** Objective function 3: minimize total tardiness ****
 declarations
  late: array(JOBS) of cpvar      ! Lateness of jobs
  totLate: cpvar
 end-declarations

 forall(k in JOBS) do
  0 <= late(k); late(k) <= MAXTIME
 end-do

! Late jobs: completion time exceeds the due date
 forall(j in JOBS) late(j) >= comp(j) - DUE(j)

 totLate = sum(k in JOBS) late(k)
 if cp_minimize(totLate) then
  writeln("Tardiness: ", getsol(totLate))
  print_sol
  print_sol3
 end-if

!-------------------------------------------------------------------

! Solution printing
 procedure print_sol
  writeln("Completion time: ", getsol(finish) ,
          "  average: ", getsol(sum(j in JOBS) comp(j)))
  write("Rel\t")
  forall(j in JOBS) write(strfmt(REL(j),4))
  write("\nDur\t")
  forall(j in JOBS) write(strfmt(DUR(j),4))
  write("\nStart\t")
  forall(j in JOBS) write(strfmt(getsol(start(j)),4))
  write("\nEnd\t")
  forall(j in JOBS) write(strfmt(getsol(comp(j)),4))
  writeln
 end-procedure

 procedure print_sol3
  write("Due\t")
```

```
  forall(j in JOBS) write(strfmt(DUE(j),4))
  write("\nLate\t")
  forall(j in JOBS) write(strfmt(getsol(late(j)),4))
  writeln
 end-procedure

end-model
```

This implementation introduces a new branching scheme, namely `settle_disjunction`. As opposed to the branching strategies we have seen so far this scheme defines a branching strategy over *constraints*, and not over variables. With this scheme a node is created by choosing a constraint from the given set and the branches from the node are obtained by adding one of the mutually exclusive constraints forming this disjunctive constraint to the constraint system.

NB: the `disjunctive` constraint of Xpress Kalis establishes pair-wise inequalities between the processing times of tasks. However, the definition of disjunctive constraints is not restricted to this case: a disjunction may have more than two components, and involve constraints of any type, including other logic relations obtained by combining constraints with `and` or `or`.

3.6 `occurrence`: Sugar production

The problem description in this section is taken from Section 6.4 'Cane sugar production' of the book 'Applications of optimization with Xpress-MP'

The harvest of cane sugar in Australia is highly mechanized. The sugar cane is immediately transported to a sugarhouse in wagons that run on a network of small rail tracks. The sugar content of a wagonload depends on the field it has been harvested from and on the maturity of the sugar cane. Once harvested, the sugar content decreases rapidly through fermentation and the wagonload will entirely lose its value after a certain time. At this moment, eleven wagons loaded with the same quantity have arrived at the sugarhouse. They have been examined to find out the hourly loss and the remaining life span (in hours) of every wagon, these data are summarized in the following table.

Table 3.5: Properties of the lots of cane sugar

Lot	1	2	3	4	5	6	7	8	9	10	11
Loss (kg/h)	43	26	37	28	13	54	62	49	19	28	30
Life span (h)	8	8	2	8	4	8	8	8	8	8	8

Every lot may be processed by any of the three, fully equivalent production lines of the sugarhouse. The processing of a lot takes two hours. It must be finished at the latest at the end of the life span of the wagonload. The manager of the sugarhouse wishes to determine a production schedule for the currently available lots that minimizes the total loss of sugar.

3.6.1 Model formulation

Let $WAGONS = \{1, ..., NW\}$ be the set of wagons, NL the number of production lines and DUR the duration of the production process for every lot. The hourly loss for every wagon w is given by $LOSS_w$ and its life span by $LIFE_w$. We observe that, in an optimal solution, the production lines need to work without any break—otherwise we could reduce the loss in sugar by advancing the start of the lot that follows the break. This means that the completion time of every lot is of the form $s \cdot DUR$, with $s > 0$ and is an integer. The maximum value of s is the number of time slots (of length DUR) that the sugarhouse will work, namely $NS = ceil(NW/NL)$, where $ceil$ stands for 'rounded to the next largest integer'. If NW/NL is an integer, every line will process exactly NS lots. Otherwise, some lines will process $NS - 1$ lots, but at least one line processes NS lots. In all cases, the length of the optimal schedule is $NS \cdot DUR$ hours. We call $SLOTS = \{1, ..., NS\}$ the set of time slots.

Every lot needs to be assigned to a time slot. We define variables $\mathrm{process}_w$ for the time slot assigned to wagon w and variables loss_w for the loss incurred by this wagonload. Every time slot may take up to NL lots because there are NL parallel lines; therefore, we limit the number of *occurrences* of time slot values among the $\mathrm{process}_w$ variables (this constraint relation is often called *cardinality constraint*):

$$\mathrm{s} \in \mathrm{SLOTS} : |\mathrm{process}_w = \mathrm{s}|_{w \in \mathrm{WAGONS}} \leqslant \mathrm{NL}$$

The loss of sugar per wagonload w and time slot s is $\mathrm{COST}_{ws} = \mathrm{s} \cdot \mathrm{DUR} \cdot \mathrm{LOSS}_w$. Let variables loss_w denote the loss incurred by wagon load w:

$$\forall w \in \mathrm{WAGONS} : \mathrm{loss}_w = \mathrm{COST}_{w,\mathrm{process}_w}$$

The objective function (total loss of sugar) is then given as the sum of all losses:

$$\text{minimize} \sum_{w \in \mathrm{WAGONS}} \mathrm{loss}_w$$

3.6.2 Implementation

The following model is the Mosel implementation of this problem. It uses the function `ceil` to calculate the maximum number of time slots.

The constraints on the processing variables are expressed by `occurrence` relations and the losses are obtained via `element` constraints. The branching strategy uses the variable selection criterion `KALIS_SMALLEST_MAX`, that is, choosing the variable with the smallest upper bound.

```
model "A-4 Cane sugar production (CP)"
 uses "kalis", "mmsystem"

 declarations
  NW = 11                            ! Number of wagon loads of sugar
  NL = 3                             ! Number of production lines
  WAGONS = 1..NW
  NS = ceil(NW/NL)
  SLOTS = 1..NS                      ! Time slots for production

  LOSS: array(WAGONS) of integer     ! Loss in kg/hour
  LIFE: array(WAGONS) of integer     ! Remaining time per lot (in hours)
  DUR: integer                       ! Duration of the production (in hours)
  COST: array(SLOTS) of integer      ! Cost per wagon

  loss: array(WAGONS) of cpvar       ! Loss per wagon
  process: array(WAGONS) of cpvar    ! Time slots for wagon loads

  totalLoss: cpvar                   ! Objective variable
 end-declarations

 initializations from 'Data/a4sugar.dat'
  LOSS LIFE DUR
 end-initializations

 forall(w in WAGONS) setdomain(process(w), 1, NS)

! Wagon loads per time slot
 forall(s in SLOTS) occurrence(s, process) <= NL

! Limit on raw product life
 forall(w in WAGONS) process(w) <= floor(LIFE(w)/DUR)

! Objective function: total loss
 forall(w in WAGONS) do
  forall(s in SLOTS) COST(s):= s*DUR*LOSS(w)
  loss(w) = element(COST, process(w))
```

```
 end-do
 totalLoss = sum(w in WAGONS) loss(w)

 cp_set_branching(assign_var(KALIS_SMALLEST_MAX, KALIS_MIN_TO_MAX, process))

! Solve the problem
 if not cp_minimize(totalLoss) then
  writeln("No solution found")
  exit(0)
 end-if

! Solution printing
 writeln("Total loss: ", getsol(totalLoss))
 forall(s in SLOTS) do
  write("Slot ", s, ": ")
  forall(w in WAGONS | getsol(process(w))=s)
   write(formattext("wagon %2d (%3g)  ", w, s*DUR*LOSS(w)))
  writeln
 end-do

end-model
```

An alternative formulation of the constraints on the processing variables is to replace them by a single `distribute` relation, indicating for every time slot the minimum and maximum number ($\mathrm{MINUSE}_s = 0$ and $\mathrm{MAXUSE}_s = \mathrm{NL}$) of production lines that may be used.

```
 forall(s in SLOTS) MAXUSE(s):= NL
 distribute(process, SLOTS, MINUSE, MAXUSE)
```

Yet another formulation of this problem is possible with Xpress Kalis, namely interpreting it as a cumulative scheduling problem (see Section 5.4), where the wagon loads are represented by tasks of unit duration, scheduled on a discrete resource with a capacity corresponding to the number of production lines.

3.6.3 Results

We obtain a total loss of 1620 kg of sugar. The corresponding schedule of lots is shown in the following Table 3.6 (there are several equivalent solutions).

Table 3.6: Optimal schedule for the cane sugar lots

Slot 1	Slot 2	Slot 3	Slot 4
lot 3 (74 kg)	lot 1 (172 kg)	lot 4 (168 kg)	lot 2 (208 kg)
lot 6 (108 kg)	lot 5 (52 kg)	lot 9 (114 kg)	lot 10 (224 kg)
lot 7 (124 kg)	lot 8 (196 kg)	lot 11 (180 kg)	

3.7 `distribute`: Personnel planning

The director of a movie theater wishes to establish a plan with the working locations for his personnel. The theater has eight employees: David, Andrew, Leslie, Jason, Oliver, Michael, Jane, and Marilyn. The ticket office needs to be staffed with three persons, theater entrances one and two require two persons each, and one person needs to be put in charge of the cloakroom. Due to different qualifications and respecting individual likes and dislikes, the following constraints need to be taken into account:

1. Leslie must be at the second entrance of the theater.
2. Michael must be at the first entrance of the theater.

3. David, Michael and Jason cannot work with each other.
4. If Oliver is selling tickets, Marylin must be with him.

3.7.1 Model formulation

Let PERS be the set of personnel and $\mathrm{LOC} = \{1, ..., 4\}$ the set of working locations where 1 stands for the ticket office, 2 and 3 for entrances 1 and 2 respectively, and 4 for the cloakroom.

We introduce decision variables place_p for the working location assigned to person p. The four individual constraints on working locations can then be stated as follows:

$$\mathrm{place}_{\mathrm{Leslie}} = 3$$
$$\mathrm{place}_{\mathrm{Michael}} = 2$$
$$\text{all-different}(\mathrm{place}_{\mathrm{David}}, \mathrm{place}_{\mathrm{Michael}}, \mathrm{place}_{\mathrm{Jason}})$$
$$\mathrm{place}_{\mathrm{Oliver}} = 1 \Rightarrow \mathrm{place}_{\mathrm{Marylin}} = 1$$

We also have to meet the staffing requirement of every working location:

$$\forall l \in \mathrm{LOC} : |\mathrm{place}_p = l|_{p \in \mathrm{PERS}} = \mathrm{REQ}_l$$

3.7.2 Implementation

For the implementation of the staffing requirements we may choose among two different constraints of Xpress Kalis, namely `occurrence` constraints (one per working location) or a `distribute` constraint (also known as *global cardinality constraint*) over all working locations. The following model shows both options. As opposed to the previous example (Section 3.6.2 we now have equality constraints. We therefore use the three-argument version of `distribute` (instead of the version with four arguments where the last two are the lower and upper bounds respectively).

For an attractive display, the model also introduces a few auxiliary data structures with the names of the working locations and the corresponding index values in the set `LOC`.

```
model "Personnel Planning (CP)"
 uses "kalis"

 forward procedure print_solution

 declarations
  PERS = {"David","Andrew","Leslie","Jason","Oliver","Michael",
          "Jane","Marilyn"}               ! Set of personnel
  LOC = 1..4                              ! Set of locations
  LOCNAMES = {"Ticketoffice", "Theater1", "Theater2",
              "Cloakroom"}                ! Names of locations
  LOCNUM: array(LOCNAMES) of integer      ! Numbers assoc. with loc.s
  REQ: array(LOC) of integer              ! No. of pers. req. per loc.

  place: array(PERS) of cpvar             ! Workplace assigned to each peson
 end-declarations

! Initialize data
 LOCNUM("Ticketoffice"):= 1; LOCNUM("Theater1"):= 2
 LOCNUM("Theater2"):= 3; LOCNUM("Cloakroom"):= 4
 REQ:: (1..4)[3, 2, 2, 1]

! Each variable has a lower bound of 1 (Ticketoffice) and an upper bound
! of 4 (Cloakroom)
 forall(p in PERS) do
  setname(place(p),"workplace["+p+"]")
  setdomain(place(p), LOC)
 end-do
```

```
! "Leslie must be at the second entrance of the theater"
 place("Leslie") = LOCNUM("Theater2")

! "Michael must be at the first entrance of the theater"
 place("Michael") = LOCNUM("Theater1")

! "David, Michael and Jason cannot work with each other"
 all_different({place("David"), place("Michael"), place("Jason")})

! "If Oliver is selling tickets, Marylin must be with him"
 implies(place("Oliver")=LOCNUM("Ticketoffice"),
         place("Marilyn")=LOCNUM("Ticketoffice"))

! Creation of a resource constraint of for every location
! forall(d in LOC) occurrence(LOCNUM(d), place) = REQ(d)

! Formulation of resource constraints using global cardinality constraint
 distribute(place, LOC, REQ)

! Setting parameters of the enumeration
 cp_set_branching(assign_var(KALIS_SMALLEST_MIN, KALIS_MAX_TO_MIN, place))

! Solve the problem
 if not cp_find_next_sol then
  writeln("Problem is infeasible")
  exit(1)
 end-if

! Solution output
 nbSolutions:= 1
 print_solution

! Search for other solutions
 while (cp_find_next_sol) do
  nbSolutions += 1
  print_solution
 end-do

! **** Solution printout ****
 procedure print_solution
  declarations
   LOCIDX: array(LOC) of string
  end-declarations
  forall(l in LOCNAMES) LOCIDX(LOCNUM(l)):=l

  writeln("\nSolution number ", nbSolutions)
  forall(p in PERS)
   writeln(" Working place of ", p, ": ", LOCIDX(getsol(place(p))))
 end-procedure

end-model
```

Since we merely wish to find a feasible assignment of working locations, this model first tests whether a feasible solution exists. If this is the case, it also enumerates all other feasible solutions. Each time a solution is found it is printed out by call to the procedure `print_solution`.

3.7.3 Results

This problem has 38 feasible solutions. A graphical representation of a feasible assignment is shown in Figure 3.2.

The following Mosel code was used for generating the graphic (see the documentation of module *mmsvg* in the Mosel Language Reference Manual for further explanation).

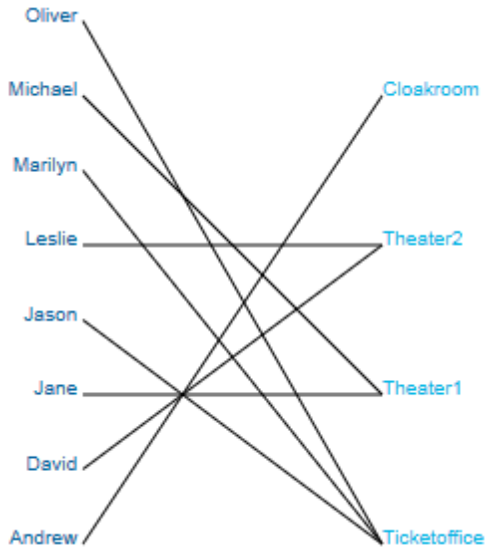

Figure 3.2: Personnel schedule representation

```
! **** Solution printout and graphical display ****
 procedure draw_solution
  svgerase
  svgaddgroup("PersGraph", "Personnel")
  svgsetstyle(SVG_TEXTANCHOR, "end")
  svgaddgroup("LocGraph", "Locations")
  svgaddgroup("AsgnGraph", "Assignments", SVG_BLACK)

  forall(d in LOCNAMES)
   svgaddtext("LocGraph", 4, LOCNUM(d), d)

  FACT:=LOCNAMES.size/PERS.size
  idx:= 1
  forall(p in PERS, idx as counter) do
   svgaddline("AsgnGraph", 2, idx*FACT, 4, getsol(place(p)))
   svgaddtext("PersGraph", 2, idx*FACT, p)
  end-do

  svgsetgraphscale(75)
  svgsetgraphviewbox(0,0,5,LOCNAMES.size+1)
  svgrefresh

  ! Uncomment to pause at every iteration:
  if nbSolutions<MAXSOL then svgpause; end-if
 end-procedure
```

3.8 `implies`: Paint production

The problem description in this section is taken from Section 7.5 'Paint production' of the book 'Applications of optimization with Xpress-MP'

As a part of its weekly production a paint company produces five batches of paints, always the same, for some big clients who have a stable demand. Every paint batch is produced in a single production process, all in the same blender that needs to be cleaned between every two batches. The durations of blending paint batches 1 to 5 are respectively 40, 35, 45, 32, and 50 minutes. The cleaning times depend on the colors and the paint types. For example, a long cleaning period is required if an oil-based paint is produced after a water-based paint, or to produce white paint after a dark color. The times are given in

minutes in the following Table 3.7 CLEAN where CLEAN_{ij} denotes the cleaning time between batch i and batch j.

Table 3.7: Matrix of cleaning times

	1	**2**	**3**	**4**	**5**
1	0	11	7	13	11
2	5	0	13	15	15
3	13	15	0	23	11
4	9	13	5	0	3
5	3	7	7	7	0

Since the company also has other activities, it wishes to deal with this weekly production in the shortest possible time (blending and cleaning). Which is the corresponding order of paint batches? The order will be applied every week, so the cleaning time between the last batch of one week and the first of the following week needs to be counted for the total duration of cleaning.

3.8.1 Formulation of model 1

As for the problem in Section 3.5 we are going to present two alternative model formulations. The first one is closer to the Mathematical Programming formulation in 'Applications of optimization with Xpress-MP', the second uses a two-dimensional element constraint.

Let $\mathrm{JOBS} = \{1, ..., \mathrm{NJ}\}$ be the set of batches to produce, DUR_j the processing time for batch j, and CLEAN_{ij} the cleaning time between the consecutive batches i and j. We introduce decision variables succ_j taking their values in JOBS, to indicate the successor of every job, and variables clean_j for the duration of the cleaning after every job. The cleaning time after every job is obtained by indexing CLEAN_{ij} with the value of succ_j. We thus have the following problem formulation.

$$\begin{aligned}
&\text{minimize} \sum_{j \in \mathrm{JOBS}} (\mathrm{DUR}_j + \mathrm{clean}_j) \\
&\forall j \in \mathrm{JOBS} : \mathrm{succ}_j \in \mathrm{JOBS} \backslash \{j\} \\
&\forall j \in \mathrm{JOBS} : \mathrm{clean}_j = \mathrm{CLEAN}_{j,\mathrm{succ}_j} \\
&\text{all-different}\left(\bigcup_{j \in \mathrm{JOBS}} \mathrm{succ}_j\right)
\end{aligned}$$

The objective function sums up the processing and cleaning times of all batches. The last (all-different) constraint guarantees that every batch occurs exactly once in the production sequence.

Unfortunately, this model does not guarantee that the solution forms a single cycle. Solving it indeed results in a total duration of 239 with an invalid solution that contains two sub-cycles $1 \to 3 \to 2 \to 1$ and $4 \to 5 \to 4$. A first possibility is to add a disjunction excluding this solution to our model and re-solve it iteratively until we reach a solution without sub-cycles.

$$\forall \mathrm{succ}_1 \neq 3 \vee \mathrm{succ}_3 \neq 2 \vee \mathrm{succ}_2 \neq 1 \vee \mathrm{succ}_1 \neq 5 \vee \mathrm{succ}_5 \neq 4$$

However, this procedure is likely to become impractical with larger data sets since it may potentially introduce an extremely large number of disjunctions. We therefore choose a different, a-priori formulation of the sub-cycle elimination constraints with a variable y_j per batch and $\mathrm{NJ} \cdot (\mathrm{NJ} - 1)$ implication constraints.

$$\begin{aligned}
&\forall j \in \mathrm{JOBS} : \mathrm{rank}_j \in \{1, ..., \mathrm{NJ}\} \\
&\forall i \in \mathrm{JOBS}, \forall j = 2, ..., \mathrm{NJ}, i \neq j : \mathrm{succ}_i = j \Rightarrow y_j = y_i + 1
\end{aligned}$$

The variables y_j correspond to the position of job j in the production cycle. With these constraints, job 1 always takes the first position.

3.8.2 Implementation of model 1

The Mosel implementation of the model formulated in the previous section is quite straightforward. The sub-cycle elimination constraints are implemented as logic relations with `implies` (a stronger formulation of these constraints is obtained by replacing the implications by equivalences, using `equiv`).

```
model "B-5 Paint production (CP)"
 uses "kalis", "mmsystem"

 declarations
  NJ = 5                              ! Number of paint batches (=jobs)
  JOBS=1..NJ

  DUR: array(JOBS) of integer         ! Durations of jobs
  CLEAN: array(JOBS,JOBS) of integer  ! Cleaning times between jobs
  CB: array(JOBS) of integer          ! Cleaning times after a batch

  succ: array(JOBS) of cpvar          ! Successor of a batch
  clean: array(JOBS) of cpvar         ! Cleaning time after batches
  y: array(JOBS) of cpvar             ! Variables for excluding subtours
  cycleTime: cpvar                    ! Objective variable
 end-declarations

 initializations from 'Data/b5paint.dat'
  DUR CLEAN
 end-initializations

 forall(j in JOBS) do
  1 <= succ(j); succ(j) <= NJ; succ(j) <> j
  1 <= y(j); y(j) <= NJ
 end-do

! Cleaning time after every batch
 forall(j in JOBS) do
  forall(i in JOBS) CB(i):= CLEAN(j,i)
  clean(j) = element(CB, succ(j))
 end-do

! Objective: minimize the duration of a production cycle
 cycleTime = sum(j in JOBS) (DUR(j)+clean(j))

! One successor and one predecessor per batch
 all_different(succ)

! Exclude subtours
 forall(i in JOBS, j in 2..NJ | i<>j)
  implies(succ(i) = j, y(j) = y(i) + 1)

! Solve the problem
 if not cp_minimize(cycleTime) then
  writeln("Problem is infeasible")
  exit(1)
 end-if
 cp_show_stats

! Solution printing
 writeln("Minimum cycle time: ", getsol(cycleTime))
 writeln("Sequence of batches:\nBatch Duration Cleaning")
 first:=1
 repeat
  writeln(formattext("  %d%8d%9d", first, DUR(first), clean(first).sol))
  first:=getsol(succ(first))
```

```
 until (first=1)

end-model
```

3.8.3 Formulation of model 2

We may choose to implement the paint production problem using rank variables similarly to the sequencing model in Section 3.5.1.

As before, let $JOBS = \{1, ..., NJ\}$ be the set of batches to produce, DUR_j the processing time for batch j, and $CLEAN_{ij}$ the cleaning time between the consecutive batches i and j. We introduce decision variables $rank_k$ taking their values in $JOBS$, for the number of the job in position k. Variables $clean_k$ ($k \in JOBS$) now denote the duration of the k^{th} cleaning time. This duration is obtained by indexing $CLEAN_{ij}$ with the values of two consecutive $rank_k$ variables. We thus have the following problem formulation.

$$\begin{aligned}
&\text{minimize} \sum_{j \in JOBS} DUR_j + \sum_{k \in JOBS} clean_k \\
&\forall k \in JOBS : rank_k \in JOBS \\
&\forall k \in \{1, ..., NJ - 1\} : clean_k = CLEAN_{rank_k, rank_{k+1}} \\
&clean_{NJ} = CLEAN_{rank_{NJ}, rank_1} \\
&\text{all-different}\left(\bigcup_{k \in JOBS} rank_k\right)
\end{aligned}$$

As in model 1, the objective function sums up the processing and cleaning times of all batches. Although not strictly necessary from the mathematical point of view, we use different sum indices for durations and cleaning times to show the difference between summing over jobs or job positions. We now have an all-different constraint over the rank variables to guarantee that every batch occurs exactly once in the production sequence.

3.8.4 Implementation of model 2

The implementation of the second model uses the 2-dimensional version of the `element` constraint in Xpress Kalis.

```
model "B-5 Paint production (CP)"
 uses "kalis", "mmsystem"

 declarations
  NJ = 5                             ! Number of paint batches (=jobs)
  JOBS=1..NJ

  DUR: array(JOBS) of integer        ! Durations of jobs
  CLEAN: array(JOBS,JOBS) of integer ! Cleaning times between jobs

  rank: array(JOBS) of cpvar         ! Number of job in position k
  clean: array(JOBS) of cpvar        ! Cleaning time after batches
  cycleTime: cpvar                   ! Objective variable
 end-declarations

 initializations from 'Data/b5paint.dat'
  DUR CLEAN
 end-initializations

 forall(k in JOBS) setdomain(rank(k), JOBS)

! Cleaning time after every batch
 forall(k in JOBS)
  if k<NJ then
```

```
   element(CLEAN, rank(k), rank(k+1)) = clean(k)
  else
   element(CLEAN, rank(k), rank(1)) = clean(k)
  end-if

! Objective: minimize the duration of a production cycle
 cycleTime = sum(j in JOBS) DUR(j) + sum(k in JOBS) clean(k)

! One position for every job
 all_different(rank)

! Solve the problem
 if not cp_minimize(cycleTime) then
  writeln("Problem is infeasible")
  exit(1)
 end-if
 cp_show_stats

! Solution printing
 writeln("Minimum cycle time: ", getsol(cycleTime))
 writeln("Sequence of batches:\nBatch Duration Cleaning")
 forall(k in JOBS)
  writeln(formattext("  %d%8d%9d", getsol(rank(k)), DUR(getsol(rank(k))),
          getsol(clean(k))))

end-model
```

3.8.5 Results

The minimum cycle time for this problem is 243 minutes which is achieved with the following sequence of batches: $1 \rightarrow 4 \rightarrow 3 \rightarrow 5 \rightarrow 2 \rightarrow 1$. This time includes 202 minutes of (incompressible) processing time and 41 minutes of cleaning.

When comparing the problem statistics produced by Xpress Kalis for this problem we see that the second model is a weaker formulation resulting in a considerably longer enumeration (using the default strategies).

3.9 equiv: Location of income tax offices

The example description in the following sections is taken from Section 15.5 'Location of income tax offices' of the book 'Applications of optimization with Xpress-MP'.

The income tax administration is planning to restructure the network of income tax offices in a region. The number of inhabitants of every city and the distances between each pair of cities are known (Table 3.8). The income tax administration has determined that offices should be established in three cities to provide sufficient coverage. Where should these offices be located to minimize the average distance per inhabitant to the closest income tax office?

3.9.1 Model formulation

Let CITIES be the set of cities. For the formulation of the problem, two groups of decision variables are necessary: a variable build_c that is one if and only if a tax office is established in city c, and a variable depend_c that takes the number of the office on which city c depends. For the formulation of the constraints, we further introduce two sets of auxiliary variables: $\mathrm{depdist}_c$, the distance from city c to the office indicated by depend_c, and numdep_c, the number of cities depending on an office location.

The following relations are required to link the build_c with the depend_c variables:
(1) numdep_c counts the number of occurrences of office location c among the variables depend_c.
(2) $\mathrm{numdep}_c \geqslant 1$ if and only if the office in c is built (as a consequence, if the office in c is not built, then

Table 3.8: Distance matrix and population of cities

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	15	37	55	24	60	18	33	48	40	58	67
2	15	0	22	40	38	52	33	48	42	55	61	61
3	37	22	0	18	16	30	43	28	20	58	39	39
4	55	40	18	0	34	12	61	46	24	62	43	34
5	24	38	16	34	0	36	27	12	24	49	37	43
6	60	52	30	12	36	0	57	42	12	50	31	22
7	18	33	43	61	27	57	0	15	45	22	40	61
8	33	48	28	46	12	42	15	0	30	37	25	46
9	48	42	20	24	24	12	45	30	0	38	19	19
10	40	55	58	62	49	50	22	37	38	0	19	40
11	58	61	39	43	37	31	40	25	19	19	0	21
12	67	61	39	34	43	22	61	46	19	40	21	0
Pop. (in 1000)	15	10	12	18	5	24	11	16	13	22	19	20

we must have $\mathrm{numdep}_c = 0$).

$$\forall c \in \mathrm{CITIES} : \mathrm{numdep}_c = |\mathrm{depend}_d = c|_{d \in \mathrm{CITIES}}$$
$$\forall c \in \mathrm{CITIES} : \mathrm{numdep}_c \geqslant 1 \Leftrightarrow \mathrm{build}_c = 1$$

Since the number of offices built is limited by the given bound NUMLOC

$$\sum_{c \in \mathrm{CITIES}} \mathrm{build}_c \leqslant \mathrm{NUMLOC}$$

it would actually be sufficient to formulate the second relation between the build_c and depend_c variables as the implication 'If $\mathrm{numdep}_c \geqslant 1$ then the office in c must be built, and inversely, if the office in c is not built, then we must have $\mathrm{numdep}_c = 0$'.

The objective function to be minimized is the total distance weighted by the number of inhabitants of the cities. We need to divide the resulting value by the total population of the region to obtain the average distance per inhabitant to the closest income tax office. The distance $\mathrm{depdist}_c$ from city c to the closest tax office location is obtained by a discrete function, namely the row c of the distance matrix DIST_{cd} indexed by the value of depend_c:

$$\mathrm{depdist}_c = \mathrm{DIST}_{c,\mathrm{depend}_c}$$

We now obtain the following CP model:

$$\text{minimize} \sum_{c \in \mathrm{CITIES}} \mathrm{POP}_c \cdot \mathrm{dist}_c$$
$$\forall c \in \mathrm{CITIES} : \mathrm{build}_c \in \{0,1\},\ \mathrm{depend}_c \in \mathrm{CITIES},$$
$$\mathrm{numdep}_c \in \mathrm{CITIES} \cup \{0\},\ \mathrm{depdist}_c \in \{\min_{d \in \mathrm{CITIES}} \mathrm{DIST}_{c,d}, \ldots, \max_{d \in \mathrm{CITIES}} \mathrm{DIST}_{c,d}\}$$
$$\forall c \in \mathrm{CITIES} : \mathrm{depdist}_c = \mathrm{DIST}_{c,\mathrm{depend}_c}$$
$$\sum_{c \in \mathrm{CITIES}} \mathrm{build}_c \leqslant \mathrm{NUMLOC}$$
$$\forall c \in \mathrm{CITIES} : \mathrm{numdep}_c = |\mathrm{depend}_d = c|_{d \in \mathrm{CITIES}}$$
$$\forall c \in \mathrm{CITIES} : \mathrm{numdep}_c \geqslant 1 \Leftrightarrow \mathrm{build}_c = 1$$

3.9.2 Implementation

To solve this problem, we define a branching strategy with two parts, one for the build_c variables and a second strategy for the $\mathrm{depdist}_c$ variables. The latter are enumerated using the `split_domain`

branching scheme that divides the domain of the branching variable into several disjoint subsets (instead of assigning a value to the variable). We now pass an array of type `cpbranching` as the argument to procedure `cp_set_branching`. The different strategies will be applied in their order in this array. Since our enumeration strategy does not explicitly include all decision variables of the problem, Xpress Kalis will enumerate these using the default strategy if any unassigned variables remain after the application of our search strategy.

The objective function—a scalar product of an array of numerical coefficients and an array of decision variables—can be stated as a linear expression, but it will be computationally more efficient to employ the dedicated `dot` constraint to formulate this sum expression as shown in the example implementation.

```
model "J-5 Tax office location (CP)"
 uses "kalis"

 forward procedure calculate_dist

 setparam("KALIS_DEFAULT_LB", 0)

 declarations
  NC = 12
  CITIES = 1..NC                        ! Set of cities

  DIST: array(CITIES,CITIES) of integer ! Distance matrix
  POP: array(CITIES) of integer         ! Population of cities
  LEN: dynamic array(CITIES,CITIES) of integer ! Road lengths
  NUMLOC: integer                       ! Desired number of tax offices
  D: array(CITIES) of integer           ! Auxiliary array used in constr. def.

  build: array(CITIES) of cpvar         ! 1 if office in city, 0 otherwise
  depend: array(CITIES) of cpvar        ! Office on which city depends
  depdist: array(CITIES) of cpvar       ! Distance to tax office
  numdep: array(CITIES) of cpvar        ! Number of depending cities per off.
  totDist: cpvar                        ! Objective function variable
  Strategy: array(1..2) of cpbranching  ! Branching strategy
 end-declarations

 initializations from 'Data/j5tax.dat'
  LEN POP NUMLOC
 end-initializations

! Calculate the distance matrix
 calculate_dist

 forall(c in CITIES) do
  build(c) <= 1
  1 <= depend(c); depend(c) <= NC
  min(d in CITIES) DIST(c,d) <= depdist(c)
  depdist(c) <= max(d in CITIES) DIST(c,d)
  numdep(c) <= NC
 end-do

! Distance from cities to tax offices
 forall(c in CITIES) do
  forall(d in CITIES) D(d):=DIST(c,d)
  element(D, depend(c)) = depdist(c)
 end-do

! Number of cities depending on every office
 forall(c in CITIES) occurrence(c, depend) = numdep(c)

! Relations between dependencies and offices built
 forall(c in CITIES) equiv( build(c) = 1, numdep(c) >= 1 )

! Limit total number of offices
 sum(c in CITIES) build(c) <= NUMLOC

! Branching strategy
```

```
 Strategy(1):= assign_and_forbid(KALIS_MAX_DEGREE, KALIS_MAX_TO_MIN, build)
 Strategy(2):= split_domain(KALIS_SMALLEST_DOMAIN, KALIS_MIN_TO_MAX,
                            depdist, true, 5)
 cp_set_branching(Strategy)

! Objective: weighted total distance
! totDist = sum(c in CITIES) POP(c)*depdist(c)
! Equivalent formulation:
 totDist = dot(POP,depdist)

! Solve the problem
 if not cp_minimize(totDist) then
  writeln("Problem is infeasible")
  exit(1)
 end-if

! Solution printing
 writeln("Total weighted distance: ", getsol(totDist),
         " (average per inhabitant: ",
    getsol(totDist)/sum(c in CITIES) POP(c), ")")
 forall(c in CITIES | getsol(build(c))>0) do
  write("Office in ", c, ": ")
  forall(d in CITIES | getsol(depend(d))=c) write(" ",d)
  writeln
 end-do

!-------------------------------------------------------------------

! Calculate the distance matrix using Floyd-Warshall algorithm
 procedure calculate_dist
 ! Initialize all distance labels with a sufficiently large value
  BIGM:=sum(c,d in CITIES | exists(LEN(c,d))) LEN(c,d)
  forall(c,d in CITIES) DIST(c,d):=BIGM

  forall(c in CITIES) DIST(c,c):=0    ! Set values on the diagonal to 0

! Length of existing road connections
  forall(c,d in CITIES | exists(LEN(c,d))) do
   DIST(c,d):=LEN(c,d)
   DIST(d,c):=LEN(c,d)
  end-do

! Update shortest distance for every node triple
  forall(b,c,d in CITIES | c<d )
   if DIST(c,d) > DIST(c,b)+DIST(b,d) then
    DIST(c,d):= DIST(c,b)+DIST(b,d)
    DIST(d,c):= DIST(c,b)+DIST(b,d)
   end-if
 end-procedure

end-model
```

This implementation contains another example of the use of a subroutine in Mosel: the calculation of the distance data is carried out in the procedure `calculate_dist`. We thus use a subroutine to structure our model, removing secondary tasks from the main model formulation.

3.9.3 Results

The optimal solution to this problem has a total weighted distance of 2438. Since the region has a total of 185,000 inhabitants, the average distance per inhabitant is 2438/185 $\approx$ 13.178 km. The three offices are established at nodes 1, 6, and 11. The first serves cities 1, 2, 5, 7, the office in node 6 cities 3, 4, 6, 9, and the office in node 11 cities 8, 10, 11, 12.

3.10 `cycle`: Paint production

In this section we work once more with the paint production problem from Section 3.8. The objective of this problem is to determine a production cycle of minimal length for a given set of jobs with sequence-dependent cleaning times between every pair of jobs.

3.10.1 Model formulation

The problem formulation in Section 3.8 uses 'all-different' constraints to ensure that every job occurs once only, calculates the duration of cleaning times with 'element' constraints, and introduces auxiliary variables and constraints to prevent subcycles in the production sequence. All these constraints can be expressed by a single constraint relation, the 'cycle' constraint.

Let $\mathrm{JOBS} = \{1, ..., \mathrm{NJ}\}$ be the set of batches to produce, DUR_j the processing time for batch j, and CLEAN_{ij} the cleaning time between the consecutive batches i and j. As before we define decision variables succ_j taking their values in JOBS, to indicate the successor of every job. The complete model formulation is the following,

$$
\begin{aligned}
&\text{minimize} \sum_{j \in \mathrm{JOBS}} \mathrm{DUR}_j + cleantime \\
&\forall j \in \mathrm{JOBS} : \mathrm{succ}_j \in \mathrm{JOBS} \backslash \{j\} \\
&cleantime = \mathrm{cycle}((\mathrm{succ}_j)_{j \in \mathrm{JOBS}}, (\mathrm{CLEAN}_{ij})_{i,j \in \mathrm{JOBS}})
\end{aligned}
$$

where 'cycle' stands for the relation *'sequence into a single cycle without subcycles or repetitions'*. The variable $cleantime$ equals the total duration of the cycle.

3.10.2 Implementation

The Mosel model using the `cycle` constraint looks as follows.

```
model "B-5 Paint production (CP)"
 uses "kalis", "mmsystem"

 setparam("KALIS_DEFAULT_LB", 0)

 declarations
  NJ = 5                              ! Number of paint batches (=jobs)
  JOBS=0..NJ-1

  DUR: array(JOBS) of integer         ! Durations of jobs
  CLEAN: array(JOBS,JOBS) of integer  ! Cleaning times between jobs

  succ: array(JOBS) of cpvar          ! Successor of a batch
  cleanTime,cycleTime: cpvar          ! Durations of cleaning / complete cycle
 end-declarations

 initializations from 'Data/b5paint.dat'
  DUR CLEAN
 end-initializations

 forall(j in JOBS) do
  0 <= succ(j); succ(j) <= NJ-1; succ(j) <> j
 end-do

! Assign values to 'succ' variables as to obtain a single cycle
! 'cleanTime' is the sum of the cleaning times
 cycle(succ, cleanTime, CLEAN)
```

```
! Objective: minimize the duration of a production cycle
 cycleTime = cleanTime +sum(j in JOBS) DUR(j)

! Solve the problem
 if not cp_minimize(cycleTime) then
  writeln("Problem is infeasible")
  exit(1)
 end-if
 cp_show_stats

! Solution printing
 writeln("Minimum cycle time: ", getsol(cycleTime))
 writeln("Sequence of batches:\nBatch Duration Cleaning")
 first:=1
 repeat
  writeln(formattext("  %d%8d%9d", first, DUR(first),
          CLEAN(first,succ(first).sol)) )
  first:=getsol(succ(first))
 until (first=1)

end-model
```

Notice that we have renumbered the tasks, starting the index range with 0, to conform with the input format expected by the `cycle` constraint.

3.10.3 Results

The optimal solution to this problem has a minimum cycle time of 243 minutes, resulting from 202 minutes of (incompressible) processing time and 41 minutes of cleaning.

The problem statistics produced by Xpress Kalis for a model run reveal that the 'cycle' version of this model is the most efficient way of representing and solving the problem: it takes fewer nodes and a shorter execution time than the two versions of Section 3.8.

3.11 Generic binary constraints: Euler knight tour

Our task is to find a tour on a chessboard for a knight figure such that the knight moves through every cell exactly once and at the end of the tour returns to its starting point. The path of the knight must follow the standard chess rules: a knight moves either one cell vertically and two cells horizontally, or two cells in the vertical and one cell in the horizontal direction, as shown in the following graphic (Figure 3.3):

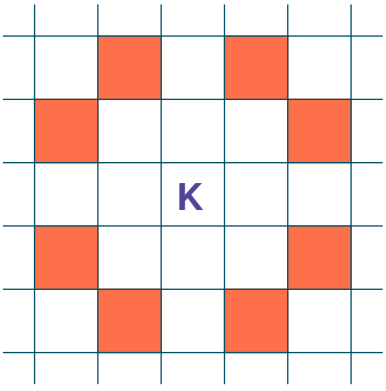

Figure 3.3: Permissible moves for a knight

3.11.1 Model formulation

To represent the chessboard we number the cells from 0 to N-1, where N is the number of cells of the board. The path of the knight is defined by N variables pos_i that indicate the i^{th} position of the knight on its tour.

The first variable is fixed to zero as the start of the tour. Another obvious constraint we need to state is that all variables pos_i take different values (every cell of the chessboard is visited exactly once):

$$\text{all-different}(pos_1, ..., pos_N)$$

We are now left with the necessity to establish a constraint relation that checks whether consecutive positions define a valid knight move. To this aim we define a *new binary constraint* 'valid_knight_move' that checks whether a given pair of values defines a permissible move according to the chess rules for knight moves. Vertically and horizontally, the two values must be no more than two cells apart and the sum of the vertical and horizontal difference must be equal to three. The complete model then looks as follows.

$$\forall i \in PATH = \{1, ..., N\} : pos_i \in \{0, ..., N-1\}$$
$$pos_1 = 0$$
$$\text{all-different}(pos_1, ..., pos_N)$$
$$\forall i \in POS = \{1, ..., N-1\} : valid\_knight\_move(pos_i, pos_{i+1})$$
$$valid\_knight\_move(pos_N, pos_1)$$

3.11.2 Implementation

Testing whether moving from position a to position b is a valid move for a knight figure can be done with the following function *valid_knight_move* where 'div' means integer division without rest and 'mod' is the rest of the integer division:

function valid_knight_move(a,b)
  xa := a div E
  ya := a mod E
  xb := b div E
  yb := b mod E
  $delta_x$:= $|xa - xb|$
  $delta_y$:= $|ya - yb|$
  return $((delta_x \leqslant 2)$ **and** $(delta_y \leqslant 2)$ **and** $(delta_x + delta_y = 3))$
end-function

The following Mosel model defines the user constraint function `valid_knight_move` as the implementation of the new binary constraints on pairs of $move_p$ variables (the constraints are established with `generic_binary_constraint`).

```
model "Euler Knight Moves"
 uses "kalis"

 parameters
  S = 8                                  ! Number of rows/columns
  NBSOL = 1                              ! Number of solutions sought
 end-parameters

 forward procedure print_solution(sol: integer)
 forward public function valid_knight_move(a:integer, b:integer): boolean

 N:= S * S                               ! Total number of cells
 setparam("KALIS_DEFAULT_LB", 0)
```

```
 setparam("KALIS_DEFAULT_UB", N-1)

 declarations
  PATH = 1..N                          ! Cells on the chessboard
  pos: array(PATH) of cpvar            ! Cell at position p in the tour
 end-declarations

! Fix the start position
 pos(1) = 0

! Each cell is visited once
 all_different(pos, KALIS_GEN_ARC_CONSISTENCY)

! The path of the knight obeys the chess rules for valid knight moves
 forall(i in 1..N-1)
  generic_binary_constraint(pos(i), pos(i+1), "valid_knight_move")
 generic_binary_constraint(pos(N), pos(1), "valid_knight_move")

! Setting enumeration parameters
 cp_set_branching(probe_assign_var(KALIS_SMALLEST_MIN, KALIS_MAX_TO_MIN,
                                   pos, 14))

! Search for up to NBSOL solutions
 solct:= 0
 while (solct<NBSOL and cp_find_next_sol) do
  solct+=1
  cp_show_stats
  print_solution(solct)
 end-do


! **** Test whether the move from position a to b is admissible ****
 public function valid_knight_move(a:integer, b:integer): boolean
  declarations
   xa,ya,xb,yb,delta_x,delta_y: integer
  end-declarations

  xa := a div S
  ya := a mod S
  xb := b div S
  yb := b mod S
  delta_x := abs(xa-xb)
  delta_y := abs(ya-yb)
  returned := (delta_x<=2) and (delta_y<=2) and (delta_x+delta_y=3)
 end-function

!*************************************************************
! Solution printing
 procedure print_solution(sol: integer)
  writeln("Solution ", sol, ":")
  forall(i in PATH)
   write(getval(pos(i)), if(i mod 10 = 0, "\n ", ""), " -> ")
  writeln("0")
 end-procedure

end-model
```

The branching scheme used in this model is the `probe_assign_var` heuristic, in combination with the variable selection `KALIS_SMALLEST_MIN` (choose variable with smallest lower bound) and the value selection criterion `KALIS_MAX_TO_MIN` (from largest to smallest domain value). Another search strategy that was found to work well (though slower than the strategy in the code listing) is

```
 cp_set_branching(assign_var(KALIS_SMALLEST_MIN, KALIS_MAX_TO_MIN, pos))
```

Our model defines two `parameters`. It is thus possible to change either the size of the chessboard (`S`) or the number of solutions sought (`NBSOL`) when executing the model without having to modify the

model source.

3.11.3 Results

The first solution printed out by our model is the following tour.

```
0 -> 17 -> 34 -> 51 -> 36 -> 30 -> 47 -> 62 -> 45 -> 39
  -> 54 -> 60 -> 43 -> 33 -> 48 -> 58 -> 52 -> 35 -> 41 -> 56
  -> 50 -> 44 -> 38 -> 55 -> 61 -> 46 -> 63 -> 53 -> 59 -> 49
  -> 32 -> 42 -> 57 -> 40 -> 25 -> 8 -> 2 -> 19 -> 4 -> 14
  -> 31 -> 37 -> 22 -> 7 -> 13 -> 28 -> 18 -> 24 -> 9 -> 3
  -> 20 -> 26 -> 16 -> 1 -> 11 -> 5 -> 15 -> 21 -> 6 -> 23
  -> 29 -> 12 -> 27 -> 10 -> 0
```

3.11.4 Alternative implementation

Whereas the aim of the model above is to give an example of the definition of user constraints, it is possible to implement this problem in a more efficient way using the model developed for the cyclic scheduling problem in Section 3.10. The set of successors (domains of variables $\mathrm{succ_p}$) can be calculated using the same algorithm that we have used above in the implementation of the user-defined binary constraints.

Without repeating the complete model definition at this place, we simply display the model formulation, including the calculation of the sets of possible successors (procedure `calculate_successors`, and the modified procedure `print_sol` for solution printout. We use a simpler version of the 'cycle' constraints than the one we have seen in Section 3.10, its only argument is the set of successor variables—there are no weights to the arcs in this problem.

```
model "Euler Knight Moves"
 uses "kalis"

 parameters
  S = 8                                  ! Number of rows/columns
  NBSOL = 1                              ! Number of solutions sought
 end-parameters

 forward procedure calculate_successors(p: integer)
 forward procedure print_solution(sol: integer)

 N:= S * S                               ! Total number of cells
 setparam("KALIS_DEFAULT_LB", 0)
 setparam("KALIS_DEFAULT_UB", N)

 declarations
  PATH = 0..N-1                          ! Cells on the chessboard
  succ: array(PATH) of cpvar             ! Successor of cell p
 end-declarations

! Calculate set of possible successors
 forall(p in PATH) calculate_successors(p)

! Each cell is visited once, no subtours
 cycle(succ)

! Search for up to NBSOL solutions
 solct:= 0
 while (solct<NBSOL and cp_find_next_sol) do
  solct+=1
  cp_show_stats
  print_solution(solct)
 end-do
```

```
! **** Calculate possible successors ****
 procedure calculate_successors(p: integer)
  declarations
   SuccSet: set of integer               ! Set of successors
  end-declarations

  xp := p div S
  yp := p mod S

  forall(q in PATH) do
   xq := q div S
   yq := q mod S
   delta_x := abs(xp-xq)
   delta_y := abs(yp-yq)
   if (delta_x<=2) and (delta_y<=2) and (delta_x+delta_y=3) then
    SuccSet +={q}
   end-if
  end-do

  setdomain(succ(p), SuccSet)
 end-procedure

!*************************************************************
! **** Solution printing ****
 procedure print_solution(sol: integer)
  writeln("Solution ", sol, ":")
  thispos:=0
  nextpos:=getval(succ(0))
  ct:=1
  while (nextpos<>0) do
   write(thispos, if(ct mod 10 = 0, "\n ", ""), " -> ")
   val:=getval(succ(thispos))
   thispos:=nextpos
   nextpos:=getval(succ(thispos))
   ct+=1
  end-do
  writeln("0")
 end-procedure

end-model
```

The calculation of the domains for the $\mathrm{succ_p}$ variables reduces these to 2-8 elements (as compared to the $\mathrm{N} = 64$ values for every $\mathrm{pos_p}$ variables), which clearly reduces the search space for this problem.

This second model finds the first solution to the problem after 131 nodes taking just a fraction of a second to execute on a standard PC whereas the first model requires several thousand nodes and considerably longer running times. It is possible to reduce the number of branch-and-bound nodes even further by using yet another version of the 'cycle' constraint that works with successor and predecessor variables. This version of 'cycle' performs a larger amount of propagation, at the expense of (slightly) slower execution times for our problem. The procedure `calculate_successors` now sets the domain of $\mathrm{pred_p}$ to the same values as $\mathrm{succ_p}$ for all cells p.

```
 declarations
  PATH = 0..N-1                          ! Cells on the chessboard
  succ: array(PATH) of cpvar             ! Successor of cell p
  pred: array(PATH) of cpvar             ! Predecessor of cell p
 end-declarations

! Calculate sets of possible successors and predecessors
 forall(p in PATH) calculate_successors(p)

! Each cell is visited once, no subtours
 cycle(succ, pred)
```

Figure 3.4 shows the graphical display of a knight's tour created with the user graph drawing functionality.

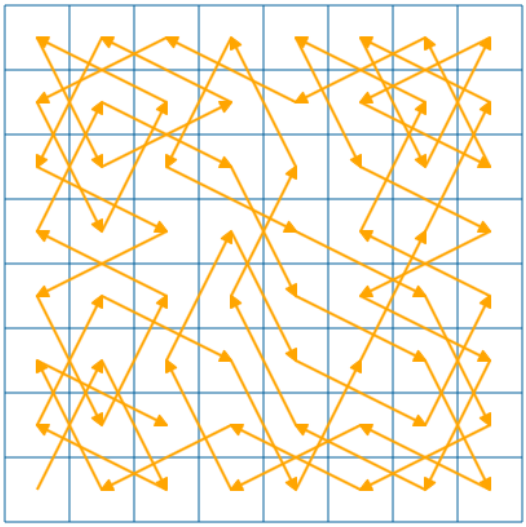

Figure 3.4: Knight's tour solution graph

3.11.5 Alternative implementation 2

Yet more efficient (a similar number of nodes with significantly shorter running times) is the following model version using the 'cycle' constraint described in Section 3.10. Since travel times from one position to the next are irrelevant in this model we simply fix all coefficients for the 'cycle' constraint to a constant and constrain the cycle length to the corresponding value.

All else, including the calculation of the sets of possible successors (procedure `calculate_successors`, and the procedure `print_sol` for solution printout is copied unchanged from the previous model.

```
model "Euler Knight Moves"
 uses "kalis"

 parameters
  S = 8                                  ! Number of rows/columns
  NBSOL = 1                              ! Number of solutions sought
 end-parameters

 forward procedure calculate_successors(p: integer)
 forward procedure print_solution(sol: integer)

 N:= S * S                               ! Total number of cells
 setparam("KALIS_DEFAULT_LB", 0)
 setparam("KALIS_DEFAULT_UB", N)

 declarations
  PATH = 0..N-1                          ! Cells on the chessboard
  succ: array(PATH) of cpvar             ! Successor of cell p
 end-declarations

! Calculate set of possible successors
 forall(p in PATH) calculate_successors(p)

! Each cell is visited once, no subtours
 cycle(succ)

! Search for up to NBSOL solutions
 solct:= 0
 while (solct<NBSOL and cp_find_next_sol) do
```

```
  solct+=1
  cp_show_stats
  print_solution(solct)
 end-do


! **** Calculate possible successors ****
 procedure calculate_successors(p: integer)
  declarations
   SuccSet: set of integer               ! Set of successors
  end-declarations

  xp := p div S
  yp := p mod S

  forall(q in PATH) do
   xq := q div S
   yq := q mod S
   delta_x := abs(xp-xq)
   delta_y := abs(yp-yq)
   if (delta_x<=2) and (delta_y<=2) and (delta_x+delta_y=3) then
    SuccSet +={q}
   end-if
  end-do

  setdomain(succ(p), SuccSet)
 end-procedure

!*************************************************************
! **** Solution printing ****
 procedure print_solution(sol: integer)
  writeln("Solution ", sol, ":")
  thispos:=0
  nextpos:=getval(succ(0))
  ct:=1
  while (nextpos<>0) do
   write(thispos, if(ct mod 10 = 0, "\n ", ""), " -> ")
   val:=getval(succ(thispos))
   thispos:=nextpos
   nextpos:=getval(succ(thispos))
   ct+=1
  end-do
  writeln("0")
 end-procedure

end-model
```

3.12 Symmetry breaking: scenes allocation problem

A movie producer has to decide when to shoot the scenes for a movie. Every scene involves a specific set of actors. A least 2 scenes and at most 5 scenes can be filmed per day. All actors of a scene must, of course, be present on the day the scene is shot. The actors have fees representing the amount they are paid per day they spend in the studio. The movie producer wishes to minimize the production costs incurred through the actors' fees.

3.12.1 Model formulation

Let SCENES denote the set of scenes, ACTORS all the actors taking part in the movie, and DAYS the set of eligible days. The decision variables $\mathrm{shoot}_{\mathrm{s}}$ indicating which day a scene is shot take their values in the set DAYS. A second set of variables are the binary indicators $\mathrm{work}_{\mathrm{ad}}$ that take the value 1 if actor a works on the day d.

Every day, the number of scenes shot needs to lie within the interval of minimum and maximum

permissible numbers (MINSCENESHOT_d, ..., MAXSCENESHOT_d). We also need to formulate the implication 'if an actor works on a day, then this day is due', or stated otherwise 'if a scene that the actor appears in is scheduled on day d, then the actor is working on this day':

$$\forall s \in \text{SCENES} : \text{shoot}_s \in \text{DAYS}$$
$$\forall a \in \text{ACTORS}, d \in \text{DAYS} : \text{work}_{ad} \in \{0, 1\}$$
$$\forall d \in \text{DAYS} : |\text{shoot}_s = d|_{s \in \text{SCENES}} \in \{\text{MINSCENESHOT}_d, ..., \text{MAXSCENESHOT}_d\}$$
$$\forall a \in \text{ACTORS}, s \in \text{APPEAR}_a, d \in \text{DAYS} : \text{shoot}_s = d \Rightarrow \text{work}_{ad} = 1$$

The objective function is to minimize the total cost incurred through actors' fees:

$$\text{minimize} \sum_{a \in \text{ACTORS}, d \in \text{DAYS}} \text{DAILYFEE}_a \cdot \text{work}_{ad}$$

Symmetry breaking constraints: it is easy to see that days are entirely interchangeable (*e.g.* all scenes from day 1 could be exchanged with those assigned to day 2 without any impact on the total cost). Indeed, any permuation of days with the same scene allocations will result in the same total cost. So it might be worthwhile to investigate how we can reduce the number of permutations in order to shorten solving times.

Consider a scene s_1: it needs to be assigned some specific day, say day 1 (all days are interchangeable at this point). Now consider a second scene s_2: it can either be assigned the same day as the first or some other (additional) day—all other days being interchangeable at this point we can assume this will be day 2. Generalizing this observation, for any scene s we can limit its value set to the days already used by scenes 1, ..., $s-1$ plus one additional day, that is:

$$\text{shoot}_1 = 1$$
$$\forall s \in \text{SCENES}, s > 1 : \text{shoot}_s \leqslant \text{maximum}(\text{shoot}_1, ..., \text{shoot}_{s-1}) + 1$$

3.12.2 Implementation

For the implementation of the lower and upper bounds on the number of scenes to be shot per day we use a global cardinality ('distribute') constraint that distributes the scenes to be shot over the available days.

For the formulation of the symmetry breaking constraints some auxiliary objects are introduced, namely the list `shootListSb` of the predecessors of scene s (that is, shoot_1, ..., shoot_{s-1}) in enumeration order of the set of scenes, and an additional decision variable maxShootSb_s defined as the maximum value among the variables in this list.

```
model "Scenes Allocation (CP)"
 uses "kalis"                    ! Gain access to the Xpress Kalis solver

! **** Data ****
 declarations
   SCENES: range                 ! Set of scene numbers
   DAYS = 1..5                   ! Day numbers when the scenes can be shot
   ACTORS: set of string         ! Set of actors
   DAILYFEE: array(ACTORS) of integer      ! Daily fees of actors
   CAST: array(SCENES) of set of string    ! Cast of actors per scene
   APPEAR: array(ACTORS) of set of integer ! Scenes where an actor appears
   MINSCENESHOT: array(DAYS) of integer    ! Minimum number of scenes per day
   MAXSCENESHOT: array(DAYS) of integer    ! Maximum number of scenes per day
 end-declarations

! Initialize actors DAILYFEE
 DAILYFEE::(["Georges Clooney","Penelope Cruz","Leonardo DiCaprio",
  "Nathalie Portman","Ryan Gosling","Monica Bellucci","Javier Bardem",
  "Keira Knightley","Vincent Cassel","Marion Cotillard","Emma Watson"])[264, 250, 303,
```

```
  40, 75, 93, 87, 57, 74, 33, 95]

! Initialize minimum and maximum numbers of scenes that have to be shot per day
 MINSCENESHOT::([1,2,3,4,5])[2,2,2,2,2]
 MAXSCENESHOT::([1,2,3,4,5])[5,5,5,5,5]

! Initialize sets of actors per scene
 CAST(1):= {"Monica Bellucci"}
 CAST(2):= {"Georges Clooney","Monica Bellucci","Ryan Gosling","Nathalie Portman"}
 CAST(3):= {"Keira Knightley","Leonardo DiCaprio","Vincent Cassel","Ryan Gosling"}
 CAST(4):= {"Penelope Cruz","Vincent Cassel"}
 CAST(5):= {"Vincent Cassel","Javier Bardem","Georges Clooney","Keira Knightley","Marion Cotillard"}
 CAST(6):= {"Emma Watson","Keira Knightley","Javier Bardem","Leonardo DiCaprio","Marion Cotillard"}
 CAST(7):= {"Penelope Cruz","Georges Clooney"}
 CAST(8):= {"Vincent Cassel","Nathalie Portman"}
 CAST(9):= {"Penelope Cruz","Keira Knightley","Vincent Cassel","Leonardo DiCaprio","Emma Watson"}
 CAST(10):= {"Penelope Cruz","Keira Knightley","Leonardo DiCaprio","Georges Clooney"}
 CAST(11):= {"Georges Clooney"}
 CAST(12):= {"Monica Bellucci","Emma Watson","Keira Knightley","Nathalie Portman","Ryan Gosling"}
 CAST(13):= {"Monica Bellucci","Nathalie Portman","Penelope Cruz","Georges Clooney"}
 CAST(14):= {"Javier Bardem","Leonardo DiCaprio"}
 CAST(15):= {"Emma Watson","Nathalie Portman","Keira Knightley","Georges Clooney"}
 CAST(16):= {"Leonardo DiCaprio","Keira Knightley","Penelope Cruz","Vincent Cassel"}
 CAST(17):= {"Leonardo DiCaprio","Georges Clooney","Ryan Gosling"}
 CAST(18):= {"Leonardo DiCaprio","Keira Knightley","Monica Bellucci","Emma Watson"}
 CAST(19):= {"Penelope Cruz"}

! Calculate appearance in scenes for each actor
 forall(a in ACTORS) APPEAR(a):= union(s in SCENES | a in CAST(s)) {s}

! **** Problem statement ****
 declarations
   shoot: array(SCENES) of cpvar ! Decision var.s: day when a scene will be shot
   work: array(ACTORS,DAYS) of cpvar
                                 ! Decision var.s: presence of actor i on day k
   totalCost: cpvar              ! Objective
   shootListSb: cpvarlist        ! List of shoot variables for symmetry breaking
   maxShootSb: array(SCENES) of cpvar   ! Max. value of shoot variables list
                                 ! per scene (formulation of symmetry breaking)
 end-declarations

 setparam("kalis_default_ub", 200000000)    ! Kalis default upper bound

! **** Domain definition ****
! Each shoot variable takes a value from the set DAYS
 forall(s in SCENES) do
   setname(shoot(s),"shoot["+s+"]")
   setdomain(shoot(s), DAYS)
 end-do

! The total cost has a lower bound of $0 and an upper bound of $200000000
 setname(totalCost, "totalCost")
 setdomain(totalCost, 0, 200000000)

! work : binary variables that indicate if actor i is present on day k
 forall(a in ACTORS, d in DAYS) do
   setname(work(a,d), "work["+a+","+d+"]")
   setdomain(work(a,d), 0, 1)
 end-do

! **** Constraints ****
! Global Cardinality Constraint to express lower and upper bounds on the
! number of scenes that have to be shot each day
 distribute(shoot,DAYS,MINSCENESHOT,MAXSCENESHOT)

! When an actor works on a day, this day is due
 forall(a in ACTORS, s in APPEAR(a), d in DAYS)
   implies(shoot(s)=d, work(a,d)=1)
```

```
! Symmetry breaking
 shoot(1) = 1                            ! All days are interchangeable at this stage
 forall(s in SCENES | s > 1) do
   shootListSb += shoot(s-1)             ! For a scene s, we need to consider
   maxShootSb(s) = maximum(shootListSb)  ! just the previously used days + 1:
   shoot(s) <= maxShootSb(s)+1           ! D(s)={1,...,max(shoot(1),...,shoot(s-1))+1}
 end-do

! Objective function
 totalCost = sum(a in ACTORS, d in DAYS) DAILYFEE(a)*work(a,d)

! Branching strategy
 cp_set_branching(assign_and_forbid(KALIS_INPUT_ORDER, KALIS_MIN_TO_MAX, shoot))

! **** Problem solving and solution reporting ****
 if cp_minimize(totalCost) then
  ! Display total cost
   writeln("Total cost: ", getsol(totalCost))

  ! Display scenes scheduled on each day
   forall(d in DAYS) do
     write("\nDay ", d, ": scenes ")
     forall(s in SCENES | getsol(shoot(s))=d) write(s, " ")
   end-do

  ! Display days worked by each actor
   forall(a in ACTORS) do
     write("\n", strfmt(a,-16), " :")
     forall(d in DAYS | getsol(work(a,d)) = 1) write("  Day ",d)
   end-do
   writeln
 else
   writeln("No solution found.")
 end-if

end-model
```

3.12.3 Results

The model shown above generates the following output:

```
Total cost: 3497

Day 1: scenes 1 8
Day 2: scenes 2 5 11 12 15
Day 3: scenes 3 7 10 13 17
Day 4: scenes 4 19
Day 5: scenes 6 9 14 16 18
Georges Clooney  :  Day 2  Day 3
Penelope Cruz    :  Day 3  Day 4  Day 5
Leonardo DiCaprio:  Day 3  Day 5
Nathalie Portman :  Day 1  Day 2  Day 3
Ryan Gosling     :  Day 2  Day 3
Monica Bellucci  :  Day 1  Day 2  Day 3  Day 5
Javier Bardem    :  Day 2  Day 5
Keira Knightley  :  Day 2  Day 3  Day 5
Vincent Cassel   :  Day 1  Day 2  Day 3  Day 4  Day 5
Marion Cotillard :  Day 2  Day 5
Emma Watson      :  Day 2  Day 5
```

Without the symmetry breaking constraints, the solving time is more than one order of magnitude longer (both for finding the optimal solution and proving its optimality) than when these constraints are included in the model.

Chapter 4

Enumeration

This chapter gives an overview on different issues related to the definition of search strategies with Xpress Kalis in Mosel, namely

- predefined search strategies,
- means of interrupting and restarting the enumeration,
- search callbacks, and
- the definition of user search strategies.

The last section discusses what may be done in the case of an infeasible constraint system:

- analyzing infeasibility and handling conflicts

4.1 Predefined search strategies

With Xpress Kalis a branching strategy is composed of three components:

- The *branching scheme* determines the shape of the search tree, this includes exhaustive schemes like `assign_var` and `split_domain` (enumeration of decision variables), `settle_disjunction` (enumeration over constraints), and `task_serialize` (enumeration of tasks in scheduling problems), or potentially incomplete searches with `probe_assign_var` or `probe_settle_disjunction`.
- The *variable selection* strategy determines the choice of the branching variables. Predefined selection criteria include

`KALIS_INPUT_ORDER`	variables in the given order,
`KALIS_LARGEST_MAX`	variable with largest upper bound,
`KALIS_LARGEST_MIN`	variable with largest lower bound,
`KALIS_MAX_DEGREE`	variable occurring in the largest number of constraints,
`KALIS_MAXREGRET_LB`	variable with largest difference between its lower bound and second-smallest domain value,
`KALIS_MAXREGRET_UB`	variable with largest difference between its upper bound and second-largest domain value,
`KALIS_RANDOM_VARIABLE`	random variable choice,
`KALIS_SDOMDEG_RATIO`	variable with smallest ratio domain size / degree,
`KALIS_SMALLEST_DOMAIN`	variable with smallest number of domain values/smallest domainintervall,

KALIS_SMALLEST_MAX	variable with smallest upper bound,
KALIS_SMALLEST_MIN	variable with smallest lower bound).
KALIS_WIDEST_DOMAIN	variable with largest number of domain values/largest domain interval,

The user may also define his own variable selection strategy (see Section 4.4.3 below).

- The *value selection* strategy determines the choice of the branching value once the variable to be branched on is known. The following predefined selection criteria are available.

KALIS_MAX_TO_MIN	enumeration of values in decreasing order,
KALIS_MIDDLE_VALUE	enumerate first values in the middle of the variable's domain,
KALIS_MIN_TO_MAX	enumeration of values in increasing order,
KALIS_NEAREST_VALUE	choose the value closest to a target value previously specified with `settarget`,
KALIS_RANDOM_VALUE	choose a random value out of the variable's domain.

The predefined criteria can be replaced by the user's own value selection strategy (see Section 4.4.3 below).

Depending on the choice of the branching scheme, it may be possible to specify additional parameters to configure the enumeration. The reader is referred to the Xpress Kalis Mosel Reference Manual for further detail. In the case of constraint branching, there is quite obviously no variable or branching value selection. The special case of *task-based branching strategies* (branching scheme `task_serialize`) is discussed in Section 5.7. Enumeration of *continuous variables* (type `cpfloatvar`) always uses the branching scheme `split_domain` with the `KALIS_SMALLEST_DOMAIN` or `KALIS_WIDEST_DOMAIN` variable selection (the former is used by the default strategy, the latter often works better in purely continuous problems) and only a subset of the value selection strategies listed above.

Branching strategies may be defined for certain specified variables (or, where applicable, constraints or tasks), or, if the corresponding argument of the branching scheme function is left out, are applied to all decision variables in a model (see, for instance, model `b4seq_ka.mos` in Section 3.5).

If the user's model does not specify any branching strategy, then the Kalis solver will apply default strategies to enumerate all variables in the model. Even if one does not wish to change the default enumeration (for discrete variables: 'smallest domain first' and 'smallest value first'), it may still be desirable to define a branching strategy to fix a certain order for the enumeration of different groups of decision variables. The previous chapter contains several examples of this: in the model `a4sugar_ka.mos` (Section 3.6) we define an enumeration over some of the variables of a model, giving them thus preference over the remainder. In model `j5tax_ka.mos` (Section 3.9) we have seen an example of a search strategy composed of different enumerations for groups of decision variables.

If a model contains both, discrete and continuous variables (`cpvar` and `cpfloatvar`) the default strategies enumerate first the discrete and then the continuous variables.

4.2 Interrupting and restarting the search

When solving large applications it is often not possible to enumerate the complete search tree within a reasonable time span. Several *stopping criteria* are therefore available in Xpress Kalis to interrupt the search. These are

KALIS_MAX_BACKTRACKS	maximum number of backtracks,
KALIS_MAX_COMPUTATION_TIME	limit on total time spent in search,

KALIS_MAX_DEPTH	maximum search tree depth,
KALIS_MAX_NODES	maximum number of nodes to explore,
KALIS_MAX_SOLUTIONS	maximum number of solutions,
KALIS_OPT_ABS_TOLERANCE	absolute difference between the objective function value in a solution and its best possible value (= current upper bound on objective function in maximization problems and lower bound with minimization),
KALIS_OPT_REL_TOLERANCE	relative difference between the objective function value in a solution and its best possible value (= current upper bound on objective function in maximization problems and lower bound with minimization).

These parameters are accessed with the Mosel functions `setparam` and `getparam` (see, for example, the output of problem statistics in model `sudoku_ka.mos` in Section 3.3, and the search time limit set in the model `freqasgn.mos` of Section 3.4).

In optimization problems, after a solution has been found the search continues from this point unless the setting of parameter `KALIS_OPTIMIZE_WITH_RESTART` is changed. The same is true if the search is interrupted by means of one of the above-named criteria and then continued, for instance with a different search strategy. To *restart the search* from the root node the procedure `cp_reset_search` needs to be called (as an example, see model `freqasgn.mos` in Section 3.4)

4.3 Callbacks

During the search the user's model may interact with the solver at certain predefined points by means of *callback functions*. This functionality is particularly useful to retrieve solution information for intermediate solutions during an optimization run as shown in the model `freqasgn.mos` (Section 3.4). Other than this *solution callback*, the user may set functions that will be called at every branch or at every node (see the Xpress Kalis Mosel Reference Manual for further detail).

4.4 User-defined enumeration strategies

The following problem description is taken from Section 14.1 of 'Applications of optimization with Xpress-MP'.

An operator needs to be assigned to each of the six machines in a workshop. Six workers have been pre-selected. Everyone has undergone a test of her productivity on every machine. Table 4.1 lists the productivities in pieces per hour. The machines run in parallel, that is, the total productivity of the workshop is the sum of the productivities of the people assigned to the machines.

Table 4.1: Productivity in pieces per hour

	Machines					
Workers	**1**	**2**	**3**	**4**	**5**	**6**
1	13	24	31	**19**	40	29
2	**18**	25	30	15	43	22
3	20	20	27	25	34	**33**
4	23	**26**	28	18	37	30
5	28	33	**34**	17	38	20
6	19	36	25	27	**45**	24

The objective is to determine an assignment of workers to machines that maximizes the total productivity. We may start by calculating a (non-optimal) heuristic solution using the following fairly natural method: choose the assignment $p \to m$ with the highest productivity, cross out the line p and the column m (since the person has been placed and the machine has an operator), and restart this process until we have assigned all persons—the resulting assignment is highlighted in bold print in the data table. However, our aim really is to solve this problem to optimality for the case of parallel machines and also for machines working in series.

4.4.1 Model formulation

This problem type is well known under the name of the *assignment problem*. Let PERS be the set of workers, MACH the set of machines (both of the same size N), and OUTP_{pm} the productivity of worker p on machine m. We define N integer variables assign_p taking values in the set of machines, where assign_p denotes the number of the machine to which the person p is assigned. The fact that a person p is assigned to a single machine m and a machine m is operated by a single person p is then expressed by the constraint that all variables assign_p take different values.

$$\forall p \in \mathrm{PERS} : \mathrm{assign}_p \in \mathrm{MACH}$$

$$\text{all-different}\left(\bigcup_{m \in \mathrm{MACH}} \mathrm{assign}_p\right)$$

Furthermore, let output_p denote the output produced by person p. The values of these variables are obtained as discrete functions in the assign_p variables:

$$\forall p \in \mathrm{PERS} : \mathrm{output}_p = \mathrm{OUTP}_{p,\mathrm{assign}_p}$$

4.4.1.1 Parallel machine assignment

The objective function to be maximized sums the output_p variables.

$$\text{maximize} \sum_{p \in \mathrm{PERS}} \mathrm{output}_p$$

Certain assignments may be infeasible. In such a case, the value of the corresponding machine needs to be removed from the domain of the variable assign_p.

4.4.1.2 Machines working in series

If the machines work in series, the least productive worker on the machine she has been assigned to determines the total productivity of the workshop. An assignment will still be described by N variables assign_p and an all-different constraint on these variables. We also have the output_p variables with the constraints linking them to the values of the assign_p variables from the previous model. To this we add a variable pmin for the minimum productivity. The objective is to maximize pmin. This type of optimization problem where one wants to maximize a minimum is called *maximin*, or *bottleneck*.

$$\text{maximize } \mathrm{pmin}$$

$$\mathrm{pmin} = \mathrm{minimum}_{p \in \mathrm{PERS}}(\mathrm{output}_p)$$

4.4.2 Implementation

The following Mosel program first implements and solves the model for the case of parallel machines. Afterwards, we define the variable pmin that is required for solving the problem for the case that the machines work in series.

```
model "I-1 Personnel assignment (CP)"
 uses "kalis"

 forward procedure parallel_heur
 forward procedure print_sol(text1,text2:string, objval:integer)

 declarations
  PERS = 1..6                        ! Personnel
  MACH = 1..6                        ! Machines
  OUTP: array(PERS,MACH) of integer  ! Productivity
 end-declarations

 initializations from 'Data/i1assign.dat'
  OUTP
 end-initializations

! **** Exact solution for parallel machines ****

 declarations
  assign: array(PERS) of cpvar       ! Machine assigned to a person
  output: array(PERS) of cpvar       ! Productivity of every person
  totalProd: cpvar                   ! Total productivity
  O: array(MACH) of integer          ! Auxiliary array for constraint def.
  Strategy: cpbranching              ! Branching strategy
 end-declarations

 forall(p in PERS) setdomain(assign(p), MACH)

! Calculate productivity per worker
 forall(p in PERS) do
  forall(m in MACH) O(m):= OUTP(p,m)
  element(O, assign(p)) = output(p)
 end-do

! Calculate total productivity
 totalProd = sum(p in PERS) output(p)

! One person per machine
 all_different(assign)

! Branching strategy
 Strategy:= assign_var(KALIS_LARGEST_MAX, KALIS_MAX_TO_MIN, output)
 cp_set_branching(Strategy)

! Solve the problem
 if cp_maximize(totalProd) then
  print_sol("Exact solution (parallel assignment)", "Total", getsol(totalProd))
 end-if

! **** Exact solution for serial machines ****

 declarations
  pmin: cpvar                        ! Minimum productivity
 end-declarations

! Calculate minimum productivity
 pmin = minimum(output)

! Branching strategy
 Strategy:= assign_var(KALIS_SMALLEST_MIN, KALIS_MAX_TO_MIN, output)
 cp_set_branching(Strategy)

! Solve the problem
 if cp_maximize(pmin) then
  print_sol("Exact solution (serial machines)", "Minimum", getsol(pmin))
 end-if
```

When the solution to the parallel assignment problem is found, we print out the solution and re-start the

search with a new branching strategy and a new objective function. Since the first search has finished completely (no interruption by a time limit, *etc.*) there is no need to reset the solver between the two runs.

The branching strategy chosen for the parallel assignment problem is inspired by the intuitive procedure described in the introduction to Section 4.4: instead of enumerating the possible assignments of workers to machines (= enumeration of the $\mathrm{assign_p}$ variables) we define an enumeration over the $\mathrm{output_p}$ variables, choosing the variable with the largest remaining value (`KALIS_LARGEST_MAX`) and branch on its values in decreasing order (`KALIS_MAX_TO_MIN`). For the second problem we need to proceed differently: to avoid being left with a small productivity value for some worker p we pick first the $\mathrm{output_p}$ variable with the smallest lower bound (`KALIS_SMALLEST_MIN`); again we enumerate the values starting with the largest one.

The following procedure `parallel_heur` may be added to the above program. It heuristically calculates a (non-optimal) solution to the parallel assignment problem using the intuitive procedure described in the introduction to Section 4.4.

```
procedure parallel_heur
 declarations
  ALLP, ALLM: set of integer        ! Copies of sets PERS and MACH
  pmax,omax,mmax: integer
 end-declarations

! Copy the sets of workers and machines
 forall(p in PERS) ALLP+={p}
 forall(m in MACH) ALLM+={m}

! Assign workers to machines as long as there are unassigned persons
 while (ALLP<>{}) do
  pmax:=0; mmax:=0; omax:=0

! Find the highest productivity among the remaining workers and machines
  forall(p in ALLP, m in ALLM)
   if OUTP(p,m) > omax then
    omax:=OUTP(p,m)
    pmax:=p; mmax:=m
   end-if

  assign(pmax) = mmax               ! Assign chosen machine to person pmax
  ALLP-={pmax}; ALLM-={mmax}        ! Remove person and machine from sets
 end-do

 writeln("Heuristic solution (parallel assignment):")
 forall(p in PERS)
  writeln("  ",p, " operates machine ", getval(assign(p)),
          " (",getval(output(p)), ")")
 writeln("  Total productivity: ", getval(totalProd))
end-procedure
```

The model is completed with a procedure for printing out the solution in a properly formatted way.

```
 writeln(text1,":")
 forall(p in PERS)
  writeln("  ",p, " operates machine ", getsol(assign(p)),
          " (",getsol(output(p)), ")")
 writeln("  ", text2, " productivity: ", objval)
end-procedure

end-model
```

4.4.3 User search

Instead of using predefined variable and value selection criteria as shown in all previous model

implementations we may choose to define our own search heuristics. We now show how to implement the search strategies from the previous model 'by hand.' In our implementation we add each time a second criterion for breaking ties in cases where the main criterion applies to several variables at a time.

Variable selection: the variable selection function has a fixed format—it receives as its argument a list of variables of type `cpvarlist` and it returns an integer, the list index of the chosen branching variable. The list of variables passed into the function may contain variables that are already instantiated. As a first step in our implementation we, therefore, determine the set `Vset` of yet uninstantiated variables—or to be more precise, the set of their indices in the list. The entries of the list of variables are accessed with the function `getvar`. Among the uninstantiated variables we calculate the set of variables `Iset` with the largest upper bound value (using function `getub`). Finally, among these variables, we choose the one with the smallest second-largest value (this corresponds to a *maximum regret* strategy). Predecessor (next-smallest) values in the domain of a decision variable are obtained with the function `getprev`. Inversely, we have function `getnext` for an enumeration of domain values in ascending order. The chosen index value is assigned to `returned` as the function's return value.

```
public function varchoice(Vars: cpvarlist): integer
 declarations
  Vset,Iset: set of integer
 end-declarations

! Set of uninstantiated variables
 forall(i in 1..getsize(Vars))
  if not is_fixed(getvar(Vars,i)) then Vset+= {i}; end-if

 if Vset={} then
  returned:= 0
 else
 ! Get the variable(s) with largest upper bound
  dmax:= max(i in Vset) getub(getvar(Vars,i))
  forall(i in Vset)
   if getub(getvar(Vars,i)) = dmax then Iset+= {i}; end-if
  dmin:= dmax

 ! Choose variable with smallest next-best value among those indexed by 'Iset'
  forall(i in Iset) do
   prev:= getprev(getvar(Vars,i),dmax)
   if prev < dmin then
    returned:= i
    dmin:= prev
   end-if
  end-do
 end-if
end-function
```

The variable selection strategy `varchoicemin` for the second optimization run (serial machines) is implemented in a similar way. We first establish the set of variables with the smallest lower bound value (using `getlb`), `Iset`; among these we choose the variable with the smallest upper bound (`getub`).

```
public function varchoicemin(Vars: cpvarlist): integer
 declarations
  Vset,Iset: set of integer
 end-declarations

! Set of uninstantiated variables
 forall(i in 1..getsize(Vars))
  if not is_fixed(getvar(Vars,i)) then Vset+= {i}; end-if

 if Vset={} then
  returned:= 0
 else
 ! Get the variable(s) with smallest lower bound
  dmin:= min(i in Vset) getlb(getvar(Vars,i))
  forall(i in Vset)
   if getlb(getvar(Vars,i)) = dmin then Iset+= {i}; end-if
```

```
 ! Choose variable with smallest upper bound among those indexed by 'Iset'
  dmax:= getparam("kalis_default_ub")
  forall(i in Iset)
   if getub(getvar(Vars,i)) < dmax then
    returned:= i
    dmax:= getub(getvar(Vars,i))
   end-if
 end-if
end-function
```

Value selection: the value selection function receives as its argument the chosen branching variable and returns a branching value for this variable. The value selection criterion we have chosen (corresponding to `KALIS_MAX_TO_MIN`) is to enumerate all values for the branching variable, starting with the largest remaining one (that is, the variable's upper bound):

```
public function valchoice(x: cpvar): integer
 returned:= getub(x)
end-function
```

Notice that with an `assign_var` or `assign_and_forbid` strategy, the user's value selection strategy should make sure to return a value that is currently in the branching variable's domain (a value chosen between the lower and upper bound is not guaranteed to lie in the domain) by using function `contains`.

Setting user search strategies: to indicate that we wish to use our own variable or value selection strategy we simply need to replace the predefined constants by the name of our Mosel functions:

```
Strategy:= assign_var("varchoice", "valchoice", output)
```

defines the strategy for the first optimization run and

```
Strategy:= assign_var("varchoicemin", "valchoice", output)
```

re-defines it for the serial machine case. Since our function `valchoice` does just the same as the `KALIS_MAX_TO_MIN` criterion, we could also combine it with our variable choice function:

```
Strategy:= assign_var("varchoicemin", KALIS_MAX_TO_MIN, output)
```

4.4.4 Results

The following table summarizes the results found with the different solution methods for the two problems of parallel and serial machines. There is a notable difference between the heuristic method and the exact solution to the problem with parallel machines.

Table 4.2: Optimal assignments for different model versions

		Person						
	Alg.	**1**	**2**	**3**	**4**	**5**	**6**	**Productivity**
Parallel Machines	**Heur.**	4 (19)	1 (18)	6 (33)	2 (26)	3 (34)	5 (45)	175
	Exact	3 (31)	5 (43)	4 (25)	6 (30)	1 (28)	2 (36)	193
Serial Machines	**Exact**	5 (40)	3 (30)	6 (33)	2 (26)	1 (28)	4 (27)	26

By adding output of solver statistics to our model (`cp_show_stats`) we find that our user search strategies result in the same search trees and program execution durations as with the predefined strategies for the parallel assignment and arrive at a slightly different solution for the serial case.

4.5 Reversible numbers

When developing your own search strategies and also in the implementation of user constraints it is sometimes helpful to be able to save some information related to a particular state of the constraint

system and have the corresponding values restored automatically whenever the system returns to an earlier state during backtrack.

The *kalis* module implements this type of backtrackable information through the types `cpreversible` and `cpreversiblearray`. A *reversible number* is an object that takes integer or real values associated with a given state of the constraint system. Earlier values are restored when the system backtracks to the corresponding state.

The values of reversible numbers are stored with the constraint system, and as such one might be tempted to compare them with decision variables, but there is a fundamental difference between the two: the principle of incrementality does not apply to reversible numbers. Whereas the bounds on decision variables can only be reduced from any node to its descendants, the value of a reversible number might be any random sequence, such as 1 at the first node, -2 at its immediate child node, and 3 at the node of the next lower level. The following small example illustrates this. At every node, the coefficients of all fixed variables are saved in the array of reversibles `ka` and the reversible number `k` is the sum of the already fixed terms. The example further uses a reversible `depth` to save the current level in the branching tree. The 'branch-up' and 'branch-down' callbacks are used for displaying the current values of all numbers.

```
model "Tracing reversible numbers"
 uses "kalis", "mmsystem"

 forward procedure save_node_state
 forward procedure branch_up
 forward procedure branch_down

 declarations
   N = 5
   R = 1..N
   C: array(R) of integer

   x: array(R) of cpvar

   k,depth: cpreversible
   ka: cpreversiblearray
 end-declarations

 C::(1..5)[-7,15,-3,19,-45]

! Initialization: all reversible numbers at 0
 set_reversible_attributes(depth, 0)
 set_reversible_attributes(k, 0)
 set_reversible_attributes(ka, 1, N, 0)

! Decision variables and constraints
 forall(i in R) setdomain(x(i), 0, 1)
 sum(i in R) x(i) >= 3

 cp_set_node_callback(->save_node_state)
 cp_set_branch_callback(->branch_down, ->branch_up)

 cp_set_branching(assign_var(KALIS_SMALLEST_DOMAIN, KALIS_MAX_TO_MIN))
 setparam("KALIS_MAX_SOLUTIONS", 3)
 while (cp_find_next_sol) do
  writeln(" "*2*round(depth.val), "Solution: ", x)
 end-do

!************ Callback functions ************
 procedure save_node_state
  setval(depth, getval(depth)+1)
  k.val:= sum(i in R) if(is_fixed(x(i)), C(i)*x(i).val, 0)
  forall(i in R) setelt(ka,i, if(is_fixed(x(i)), C(i)*x(i).val, 0))
 end-procedure

 procedure branch_up
  writeln(textfmt(" "*2*round(depth.val)+ "up  ",-20), k, "\t", ka)
```

```
 end-procedure

 procedure branch_down
  writeln(textfmt(" "*2*round(depth.val)+ "down  ",-20), k, "\t", ka)
 end-procedure

end-model
```

The output produced by the example is shown here, notice how the tree `depth` value is used to indent the lines, and observe how the values of reversibles change between nodes, in particular the value of the reversible number `k`.

```
    down            0.000000    [0 0 0 0 0 ]
      down          -7.000000   [-7 0 0 0 0 ]
        down        8.000000    [-7 15 0 0 0 ]
          down      5.000000    [-7 15 -3 0 0 ]
            down    24.000000   [-7 15 -3 19 0 ]
            Solution: [V001[1],V002[1],V003[1],V004[1],V005[1]]
          up        5.000000    [-7 15 -3 0 0 ]
            down    24.000000   [-7 15 -3 19 0 ]
            Solution: [V001[1],V002[1],V003[1],V004[1],V005[0]]
          up        5.000000    [-7 15 -3 0 0 ]
        up          8.000000    [-7 15 0 0 0 ]
          down      5.000000    [-7 15 -3 0 0 ]
            down    5.000000    [-7 15 -3 0 0 ]
            Solution: [V001[1],V002[1],V003[1],V004[0],V005[1]]
          up        5.000000    [-7 15 -3 0 0 ]
        up          8.000000    [-7 15 0 0 0 ]
      up            -7.000000   [-7 0 0 0 0 ]
    up              0.000000    [0 0 0 0 0 ]
  up                0.000000    [0 0 0 0 0 ]
```

4.6 Analyzing infeasibility and handling conflicts

In many cases the return status 'problem infeasible' or ' constraint inconsistent' is not a desirable situation since we really wish to find a solution to the problem we are solving. To support the user's analysis of the cause of the infeasibility Kalis can generate reports about sets of infeasible constraints. During the development of a model such a facility helps tracking model formulation errors and in applications inconsistencies in data are identified more easily.

We shall work once more with the Sudoku example from Section 3.3. Instead of launching the enumeration through the CP solver we now prompt the user to input a value for a randomly chosen cell. If the value set by the user leads to an inconsistent state of the constraint system we invoke the solver display of the minimal set of constraints causing the infeasibility.

4.6.1 Implementation

The problem statement and setting of start data remain exactly the same as with the standard implementation of the problem seen in Section 3.3. What is new are the prompt for user input and the subsequent handling of infeasibilities. The underlying mechanisms are markers (often referred to as *choice points*) to save the state of the constraint solver at a given moment and the possibility to return to the last such saved state. The corresponding Mosel subroutines are `cp_save_state` and `cp_restore_state`. After stating a new constraint but before propagating its effect (the automated propagation has been switched off through the setting of `AUTO_PROPAGATE` at the beginning of the model) we save the state of the constraint system. If the new constraint turns out to be infeasible the solver is set back to the state before the constraint propagation and we can then invoke the infeasibility analysis (`cp_infeas_analysis`).

The subroutine for solution printing has been replaced by a routine printing the values of all fixed variables.

```
model "Conflicts"
 uses "kalis"

 forward procedure print_values

 setparam("kalis_default_lb", 1)
 setparam("kalis_default_ub", 9)         ! Default variable bounds

 declarations
  XS = {'A','B','C','D','E','F','G','H','I'}   ! Columns
  YS = 1..9                              ! Rows
  v: array(XS,YS) of cpvar               ! Number assigned to cell (x,y)
 end-declarations

 forall(x in XS,y in YS) setname(v(x,y), "v("+x+","+y+")")

 setparam("KALIS_AUTO_PROPAGATE", 0)     ! Disable automatic propagation

! Data from "The Guardian", 29 July, 2005. http://www.guardian.co.uk/sudoku
 v('A',1)=8; v('F',1)=3
 v('B',2)=5; v('G',2)=4
 v('A',3)=2; v('E',3)=7; v('H',3)=6
 v('D',4)=1; v('I',4)=5
 v('C',5)=3; v('G',5)=9
 v('A',6)=6; v('F',6)=4
 v('B',7)=7; v('E',7)=2; v('I',7)=3
 v('C',8)=4; v('H',8)=1
 v('D',9)=9; v('I',9)=8


! All-different values in rows
 forall(y in YS) all_different(union(x in XS) {v(x,y)})

! All-different values in columns
 forall(x in XS) all_different(union(y in YS) {v(x,y)})

! All-different values in 3x3 squares
 forall(s in {{'A','B','C'},{'D','E','F'},{'G','H','I'}}, i in 0..2)
  all_different(union(x in s, y in {1+3*i,2+3*i,3+3*i}) {v(x,y)})

! Prompt user for new assignments
 declarations
  NX: array(1..9) of string
  val: integer
 end-declarations

 NX::(1..9)['A','B','C','D','E','F','G','H','I']
 cp_save_state
 res:=cp_propagate
 print_values
 setrandseed(3)

 while (res and (or(x in XS,y in YS) not is_fixed(v(x,y))) ) do
 ! Select randomly a yet unassigned cell
  rx:=round(0.5+9*random); ry:=round(0.5+9*random)
  while (is_fixed(v(NX(rx),ry))) do
   rx:=round(0.5+9*random); ry:=round(0.5+9*random)
  end-do

 ! Prompt for user input
  writeln(v(NX(rx),ry), " = ")
  readln(val)

 ! Add the new constraint
  v(NX(rx),ry)=val                    ! State a new constraint
  cp_save_state                       ! Save system state before propagation
  res:=cp_propagate                   ! Propagate the new constraint

 ! Print resulting board
```

```
  print_values
 end-do

! An infeasible state has been reached
 if not res then
  cp_restore_state                    ! Restore state before propagation
  cp_infeas_analysis                  ! Analyze infeasibility, print report
 else
  writeln("Problem solved")
 end-if

!*************************************************************
! Print fixed values
 procedure print_values
  writeln("   A B C   D E F   G H I")
  forall(y in YS) do
   write(y, ": ")
   forall(x in XS)
    write(if(is_fixed(v(x,y)), string(getval(v(x,y))), "."),
          if(x in {'C','F'}, " | ", " "))
   writeln
   if y mod 3 = 0 then
    writeln("   ---------------------")
   end-if
  end-do
 end-procedure

end-model
```

4.6.2 Results

With the following user value input sequence:

```
v(F,7)[1,5..6,8] =
1
v(I,1)[1..2,7,9] =
1
v(I,8)[2,6..7] =
2
```

we obtain an infeasibility for the third value. The values printed at this stage and the report generated by the solver show why this value leads to a conflict.

```
   A B C   D E F   G H I
1: 8 . . | . . 3 | . . 1
2: . 5 . | . . . | 4 . 7
3: 2 . 1 | . 7 . | . 6 9
   ---------------------
4: . . . | 1 . . | . . 5
5: . . 3 | . . . | 9 . .
6: 6 . . | . . 4 | . . 7
   ---------------------
7: . 7 . | . 2 1 | . . 3
8: . . 4 | . . . | . 1 2
9: . . . | 9 . . | . . 8
   ---------------------
------------------------
  Minimal Conflict Set
------------------------
  1 : AllDifferent(v(I,1),v(I,2),v(I,3),v(I,4),v(I,5),v(I,6),v(I,7),v(I,8),v(I,9
))
```

Note: in the present example we examine the effect of adding just a single (bound) constraint to our

problem. However, the infeasibility analysis functionality can be applied for any number and type of constraints.

CHAPTER 5

Scheduling

This chapter shows how to

- define and setup the modeling objects `cptask` and `cpresource`,
- formulate and solve scheduling problems using these objects,
- access information from the modeling objects.
- parameterize search strategies involving the scheduling objects,

5.1 Tasks and resources

Scheduling and planning problems are concerned with determining a plan for the execution of a given set of tasks. The objective may be to generate a *feasible* schedule that satisfies the given constraints (such as sequence of tasks or limited resource availability) or to *optimize* a given criterion such as the makespan of the schedule.

Xpress Kalis defines several aggregate modeling objects to simplify the formulation of standard scheduling problems: tasks (processing operations, activities) are represented in Mosel by the type `cptask` and resources (machines, raw material *etc.*) by the type `cpresource`. When working with these scheduling objects it is often sufficient to state the objects and their properties, such as task duration or resource use; the necessary constraint relations are set up automatically by Xpress Kalis (referred to as *implicit constraints*). In the following sections we show a number of examples using this mechanism:

- The simplest case of a scheduling problem involves only tasks and precedence constraints between tasks (project scheduling problem in Section 5.2).
- Tasks may be mutually exclusive, *e.g.* because they use the same unitary resource (disjunctive scheduling / sequencing problem in Section 5.3).
- Resources may be usable by several tasks at a time, up to a given capacity limit (cumulative resources, see Section 5.4).
- A different classification of resources is the distinction between renewable and non-renewable resources (see Section 5.5).
- Many extensions of the standard problems are possible, such as sequence-dependent setup time (see Section 5.6.1), choice of resources (Section 5.6.2), tasks with non-constant resource profiles (Section 5.6.3), or given idle times for resources (Section 5.6.4).

If the enumeration is started with the function `cp_schedule` the solver will employ specialized search strategies suitable for the corresponding (scheduling) problem type. It is possible to parameterize these

strategies or to define user search strategies for scheduling objects (see Section 5.7). Alternatively, the standard optimization functions `cp_minimize` / `cp_maximize` may be used. In this case the enumeration does not exploit the structural information provided by the scheduling objects and works simply with decision variables.

The properties of scheduling objects (such as start time or duration of tasks) can be accessed and employed, for instance, in the definition of constraints, thus giving the user the possibility to extend the predefined standard problems with other types of constraints. For even greater flexibility Xpress Kalis also enables the user to formulate his scheduling problems without the aggregate modeling objects, using dedicated global constraints on decision variables of type `cpvar`. Most examples in this chapter are therefore given with two different implementations, one using the scheduling objects and another without these objects.

5.2 Precedences

Probably the most basic type of a scheduling problem is to plan a set of tasks that are linked by precedence constraints.

The problem described in this section is taken from Section 7.1 'Construction of a stadium' of the book 'Applications of optimization with Xpress-MP'

A construction company has been awarded a contract to construct a stadium and wishes to complete it within the shortest possible time. Table 5.1 lists the major tasks and their durations in weeks. Some tasks can only start after the completion of certain other tasks, equally indicated in the table.

Table 5.1: Data for stadium construction

Task	Description	Duration	Predecessors
1	Installing the construction site	2	none
2	Terracing	16	1
3	Constructing the foundations	9	2
4	Access roads and other networks	8	2
5	Erecting the basement	10	3
6	Main floor	6	4,5
7	Dividing up the changing rooms	2	4
8	Electrifying the terraces	2	6
9	Constructing the roof	9	4,6
10	Lighting of the stadium	5	4
11	Installing the terraces	3	6
12	Sealing the roof	2	9
13	Finishing the changing rooms	1	7
14	Constructing the ticket office	7	2
15	Secondary access roads	4	4,14
16	Means of signalling	3	8,11,14
17	Lawn and sport accessories	9	12
18	Handing over the building	1	17

5.2.1 Model formulation

This problem is a classical project scheduling problem. We add a fictitious task with 0 duration that corresponds to the end of the project. We thus consider the set of tasks $TASKS = \{1, ..., N\}$ where N is the fictitious end task.

Every construction task j ($j \in TASKS$) is represented by a task object $task_j$ with variable start time $task_j.start$ and a duration fixed to the given value DUR_j. The precedences between tasks are

represented by a precedence graph with arcs (i, j) symbolizing that task i precedes task j.

The objective is to minimize the completion time of the project, that is the start time of the last, fictitious task N. We thus obtain the following model where an upper bound $HORIZON$ on the start times is given by the sum of all task durations:

$$\begin{aligned}
&\text{tasks } task_j(j \in TASKS)\\
&\text{minimize } task_N.start\\
&\forall j \in TASKS : task_j.start \in \{0, ..., HORIZON\}\\
&\forall j \in TASKS : task_j.duration = DUR_j\\
&\forall j \in TASKS : task_j.predecessors = \bigcup_{i \in TASKS, s.t. \exists ARC_{ij}} \{task_i\}
\end{aligned}$$

5.2.2 Implementation

The following Mosel model shows the implementation of this problem with Xpress Kalis. Since there are no side-constraints, the earliest possible completion time of the schedule is the earliest start of the fictitious task N. To trigger the propagation of task-related constraints we call the function `cp_propagate` followed by a call to `cp_shave` that performs some additional tests to remove infeasible values from the task variables' domains. At this point, constraining the start of the fictitious end task to its lower bound reduces all task start times to their feasible intervals through the effect of constraint propagation. The start times of tasks on the *critical path* are fixed to a single value. The subsequent call to minimization only serves for instantiating all variables with a single value so as to enable the graphical representation of the solution.

```
model "B-1 Stadium construction (CP)"
 uses "kalis"

 declarations
  N = 19                              ! Number of tasks in the project
                                      ! (last = fictitious end task)
  TASKS = 1..N
  ARC: dynamic array(range,range) of integer  ! Matrix of the adjacency graph
  DUR: array(TASKS) of integer        ! Duration of tasks
  HORIZON : integer                   ! Time horizon

  task: array(TASKS) of cptask        ! Tasks to be planned
  bestend: integer
 end-declarations

 initializations from 'Data/b1stadium.dat'
  DUR ARC
 end-initializations

 HORIZON:= sum(j in TASKS) DUR(j)

! Setting up the tasks
 forall(j in TASKS) do
  setdomain(getstart(task(j)), 0, HORIZON)   ! Time horizon for start times
  set_task_attributes(task(j), DUR(j))       ! Duration
  setsuccessors(task(j), union(i in TASKS | exists(ARC(j,i))) {task(i)})
 end-do                                      ! Precedences

 if not cp_propagate or not cp_shave then
  writeln("Problem is infeasible")
  exit(1)
 end-if

! Since there are no side-constraints, the earliest possible completion
! time is the earliest start of the fictitious task N
```

```
 bestend:= getlb(getstart(task(N)))
 getstart(task(N)) <= bestend
 writeln("Earliest possible completion time: ", bestend)

! For tasks on the critical path the start/completion times have been fixed
! by setting the bound on the last task. For all other tasks the range of
! possible start/completion times gets displayed.
 forall(j in TASKS) writeln(j, ": ", getstart(task(j)))

! Complete enumeration: schedule every task at the earliest possible date
 if cp_minimize(getstart(task(N))) then
  writeln("Solution after optimization: ")
  forall(j in TASKS) writeln(j, ": ", getsol(getstart(task(j))))
 end-if
end-model
```

Instead of indicating the predecessors of a task, we may just as well state the precedence constraints by indicating the sets of *successors* for every task:

```
setsuccessors(task(j), union(i in TASKS | exists(ARC(j,i))) {task(i)})
```

5.2.3 Results

The earliest completion time of the stadium construction is 64 weeks.

5.2.4 Alternative formulation without scheduling objects

As for the previous formulation, we work with a set of tasks $TASKS = \{1, ..., N\}$ where N is the fictitious end task. For every task j we introduce a decision variable $start_j$ to denote its start time. With DUR_j the duration of task j, the precedence relation 'task i precedes task j' is stated by the constraint

$$start_i + DUR_i \leqslant start_j$$

We therefore obtain the following model for our project scheduling problem:

$$\begin{aligned}
&\text{minimize } start_N \\
&\forall j \in TASKS : start_j \in \{0, ..., HORIZON\} \\
&\forall i, j \in TASKS, \exists ARC_{ij} : start_i + DUR_i \leqslant start_j
\end{aligned}$$

The corresponding Mosel model is printed in full below. Notice that we have used explicit posting of the precedence constraints—in the case of an infeasible data instance this may help tracing the cause of the infeasibility.

```
model "B-1 Stadium construction (CP)"
 uses "kalis"

 declarations
  N = 19                              ! Number of tasks in the project
                                      ! (last = fictitious end task)
  TASKS = 1..N
  ARC: dynamic array(range,range) of integer  ! Matrix of the adjacency graph
  DUR: array(TASKS) of integer        ! Duration of tasks
  HORIZON : integer                   ! Time horizon

  start: array(TASKS) of cpvar        ! Start dates of tasks
  bestend: integer
 end-declarations

 initializations from 'Data/b1stadium.dat'
```

```
 DUR ARC
end-initializations

HORIZON:= sum(j in TASKS) DUR(j)

forall(j in TASKS) do
 0 <= start(j); start(j) <= HORIZON
end-do

! Task i precedes task j
forall(i, j in TASKS | exists(ARC(i, j))) do
 Prec(i,j):= start(i) + DUR(i) <= start(j)
 if not cp_post(Prec(i,j)) then
  writeln("Posting precedence ", i, "-", j, " failed")
  exit(1)
 end-if
end-do

! Since there are no side-constraints, the earliest possible completion
! time is the earliest start of the fictitious task N
bestend:= getlb(start(N))
start(N) <= bestend
writeln("Earliest possible completion time: ", bestend)

! For tasks on the critical path the start/completion times have been fixed
! by setting the bound on the last task. For all other tasks the range of
! possible start/completion times gets displayed.
forall(j in TASKS) writeln(j, ": ", start(j))

end-model
```

5.3 Disjunctive scheduling: unary resources

The problem of sequencing jobs on a single machine described in Section 3.5 may be represented as a disjunctive scheduling problem using the 'task' and 'resource' modeling objects.

The reader is reminded that the problem is to schedule the processing of a set of non-preemptive tasks (or jobs) on a single machine. For every task j its release date, duration, and due date are given. The problem is to be solved with three different objectives, minimizing the makespan, the average completion time, or the total tardiness.

5.3.1 Model formulation

The major part of the model formulation consists of the definition of the scheduling objects 'tasks' and 'resources'.

Every job j ($j \in \mathrm{JOBS} = \{1, ..., \mathrm{NJ}\}$) is represented by a task object $\mathrm{task_j}$, with a start time $\mathrm{task_j}.start$ in $\{\mathrm{REL_j}, ..., \mathrm{MAXTIME}\}$ (where $\mathrm{MAXTIME}$ is a sufficiently large value, such as the sum of all release dates and all durations, and $\mathrm{REL_j}$ the release date of job j) and the task duration $\mathrm{task_j}.duration$ fixed to the given processing time $\mathrm{DUR_j}$. All jobs use the same resource res of unitary capacity. This means that at most one job may be processed at any one time, we thus implicitly state the disjunctions between the jobs.

Another implicit constraint established by the task objects is the relation between the start, duration, and completion time of a job j.

$$\forall j \in \mathrm{JOBS} : \mathrm{task_j}.end = \mathrm{task_j}.start + \mathrm{task_j}.duration$$

Objective 1: The first objective is to minimize the makespan (completion time of the schedule) or, equivalently, to minimize the completion time $finish$ of the last job. The complete model is then given by the following (where $\mathrm{MAXTIME}$ is a sufficiently large value, such as the sum of all release dates and all

durations):

$$
\begin{aligned}
&\text{resource } res \\
&\text{tasks } task_j(j \in JOBS) \\
&\text{minimize } finish \\
&finish = maximum_{j \in JOBS}(task_j.end) \\
&res.capacity = 1 \\
&\forall j \in JOBS : task_j.end \in \{0, ..., MAXTIME\} \\
&\forall j \in JOBS : task_j.start \in \{REL_j, ..., MAXTIME\} \\
&\forall j \in JOBS : task_j.duration = DUR_j \\
&\forall j \in JOBS : task_j.requirement_{res} = 1
\end{aligned}
$$

Objective 2: The formulation of the second objective (minimizing the average processing time or, equivalently, minimizing the sum of the job completion times) remains unchanged from the first model—we introduce an additional variable $totComp$ representing the sum of the completion times of all jobs.

$$
\begin{aligned}
&\text{minimize } totComp \\
&totComp = \sum_{j \in JOBS} task_j.end
\end{aligned}
$$

Objective 3: To formulate the objective of minimizing the total tardiness, we introduce new variables $late_j$ to measure the amount of time that a job finishes after its due date. The value of these variables corresponds to the difference between the completion time of a job j and its due date DUE_j. If the job finishes before its due date, the value must be zero. The objective now is to minimize the sum of these tardiness variables:

$$
\begin{aligned}
&\text{minimize } totLate \\
&totLate = \sum_{j \in JOBS} late_j \\
&\forall j \in JOBS : late_j \in \{0, ..., MAXTIME\} \\
&\forall j \in JOBS : late_j \geqslant task_j.end - DUE_j
\end{aligned}
$$

5.3.2 Implementation

The following Mosel implementation with *kalis* (file `b4seq3_ka.mos`) shows how to set up the necessary task and resource modeling objects. The resource capacity is set with procedure `set_resource_attributes` (the resource is of the type `KALIS_UNARY_RESOURCE` meaning that it processes at most one task at a time), for the tasks we use the procedure `set_task_attributes`. The latter exists in several overloaded versions for different combinations of arguments (task attributes)—the reader is referred to the Xpress Kalis Mosel Reference Manual for further detail.

For the formulation of the `maximum` constraint we use an (auxiliary) list of variables: *kalis* does not allow the user to employ the access functions to modeling objects (`getstart`, `getduration`, *etc.*) in set expressions such as `union(j in JOBS) {getend(task(j))}`.

```
model "B-4 Sequencing (CP)"
 uses "kalis"

 forward procedure print_sol
 forward procedure print_sol3
```

```
declarations
 NJ = 7                          ! Number of jobs
 JOBS=1..NJ

 REL: array(JOBS) of integer     ! Release dates of jobs
 DUR: array(JOBS) of integer     ! Durations of jobs
 DUE: array(JOBS) of integer     ! Due dates of jobs

 task: array(JOBS) of cptask     ! Tasks (jobs to be scheduled)
 res: cpresource                 ! Resource (machine)

 finish: cpvar                   ! Completion time of the entire schedule
end-declarations

initializations from 'Data/b4seq.dat'
 DUR REL DUE
end-initializations

! Setting up the resource (formulation of the disjunction of tasks)
set_resource_attributes(res, KALIS_UNARY_RESOURCE, 1)

! Setting up the tasks (durations and disjunctions)
forall(j in JOBS) set_task_attributes(task(j), DUR(j), res)

MAXTIME:= max(j in JOBS) REL(j) + sum(j in JOBS) DUR(j)

forall(j in JOBS) do
 0 <= getstart(task(j)); getstart(task(j)) <= MAXTIME
 0 <= getend(task(j)); getend(task(j)) <= MAXTIME
end-do

! Start times
forall(j in JOBS) getstart(task(j)) >= REL(j)

!**** Objective function 1: minimize latest completion time ****
declarations
 L: cpvarlist
end-declarations

forall(j in JOBS) L += getend(task(j))
finish = maximum(L)

if cp_schedule(finish) >0 then
 print_sol
end-if


!**** Objective function 2: minimize average completion time ****
declarations
 totComp: cpvar
end-declarations

totComp = sum(j in JOBS) getend(task(j))

if cp_schedule(totComp) > 0 then
 print_sol
end-if


!**** Objective function 3: minimize total tardiness ****
declarations
 late: array(JOBS) of cpvar      ! Lateness of jobs
 totLate: cpvar
end-declarations

forall(j in JOBS) do
 0 <= late(j); late(j) <= MAXTIME
end-do
```

```
! Late jobs: completion time exceeds the due date
 forall(j in JOBS) late(j) >= getend(task(j)) - DUE(j)

 totLate = sum(j in JOBS) late(j)
 if cp_schedule(totLate) > 0 then
  writeln("Tardiness: ", getsol(totLate))
  print_sol
  print_sol3
 end-if


!-------------------------------------------------------------------

! Solution printing
 procedure print_sol
  writeln("Completion time: ", getsol(finish) ,
          "  average: ", getsol(sum(j in JOBS) getend(task(j))))
  write("Rel\t")
  forall(j in JOBS) write(strfmt(REL(j),4))
  write("\nDur\t")
  forall(j in JOBS) write(strfmt(DUR(j),4))
  write("\nStart\t")
  forall(j in JOBS) write(strfmt(getsol(getstart(task(j))),4))
  write("\nEnd\t")
  forall(j in JOBS) write(strfmt(getsol(getend(task(j))),4))
  writeln
 end-procedure

 procedure print_sol3
  write("Due\t")
  forall(j in JOBS) write(strfmt(DUE(j),4))
  write("\nLate\t")
  forall(j in JOBS) write(strfmt(getsol(late(j)),4))
  writeln
 end-procedure

end-model
```

5.3.3 Results

This model produces similar results as those reported for the model versions in Section 3.5. Figure 5.1 shows a Gantt chart display of the solution. Above the Gantt chart we can see the resource usage display: the machine is used without interruption by the tasks, that is, even if we relaxed the constraints given by the release times and due dates it would not have been possible to generate a schedule terminating earlier.

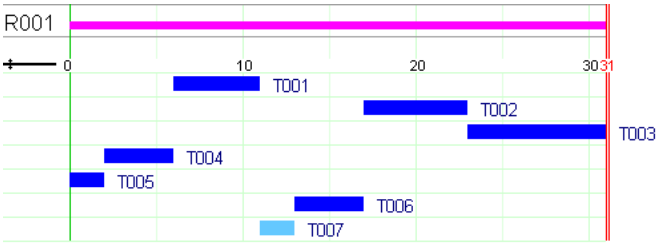

Figure 5.1: Solution Gantt chart

5.4 Cumulative scheduling: discrete resources

The problem described in this section is taken from Section 9.4 ‘Backing up files’ of the book

'Applications of optimization with Xpress-MP'

Our task is to save sixteen files of the following sizes: 46kb, 55kb, 62kb, 87kb, 108kb, 114kb, 137kb, 164kb, 253kb, 364kb, 372kb, 388kb, 406kb, 432kb, 461kb, and 851kb onto empty disks of 1.44Mb capacity. How should the files be distributed in order to minimize the number of floppy disks used?

5.4.1 Model formulation

This problem belongs to the class of *binpacking problems*. We show here how it may be formulated and solved as a cumulative scheduling problem, where the disks are the resource and the files the tasks to be scheduled.

The floppy disks may be represented as a single, discrete resource disks, where every time unit stands for one disk. The resource capacity corresponds to the disk capacity.

We represent every file f ($\mathrm{f} \in \mathrm{FILES}$) by a task object $\mathrm{file_f}$, with a fixed duration of 1 unit and a resource requirement that corresponds to the given size $\mathrm{SIZE_f}$ of the file. The 'start' field of the task then indicates the choice of the disk for saving this file.

The objective is to minimize the number of disks that are used, which corresponds to minimizing the largest value taken by the 'start' fields of the tasks (that is, the number of the disk used for saving a file). We thus have the following model.

$$
\begin{aligned}
&\text{resource } \mathrm{disks} \\
&\text{tasks } \mathrm{files_f}(\mathrm{f} \in \mathrm{FILES}) \\
&\text{minimize } \mathrm{diskuse} \\
&\mathrm{diskuse} = \text{maximum}_{\mathrm{f} \in \mathrm{FILES}}(\mathrm{file_f.start}) \\
&\mathrm{disks.capacity} = \mathrm{CAP} \\
&\forall \mathrm{f} \in \mathrm{FILES} : \mathrm{file_f.start} \geqslant 1 \\
&\forall \mathrm{f} \in \mathrm{FILES} : \mathrm{file_f.duration} = 1 \\
&\forall \mathrm{f} \in \mathrm{FILES} : \mathrm{file_f.requirement_{disks}} = \mathrm{SIZE_f}
\end{aligned}
$$

5.4.2 Implementation

The Mosel implementation with Xpress Kalis is quite straightforward. We define a resource of the type `KALIS_DISCRETE_RESOURCE`, indicating its total capacity. The definition of the tasks is similar to what we have seen in the previous example.

```
model "D-4 Bin packing (CP)"
 uses "kalis"

 declarations
  ND: integer                          ! Number of floppy disks
  FILES = 1..16                        ! Set of files
  DISKS: range                         ! Set of disks

  CAP:  integer                        ! Floppy disk size
  SIZE: array(FILES) of integer        ! Size of files to be saved

  file: array(FILES) of cptask         ! Tasks (= files to be saved)
  disks: cpresource                    ! Resource representing disks
  L: cpvarlist
  diskuse: cpvar                       ! Number of disks used
 end-declarations

 initializations from 'Data/d4backup.dat'
  CAP SIZE
 end-initializations
```

```
! Provide a sufficiently large number of disks
 ND:= ceil((sum(f in FILES) SIZE(f))/CAP)
 DISKS:= 1..ND

! Setting up the resource (capacity limit of disks)
 set_resource_attributes(disks, KALIS_DISCRETE_RESOURCE, CAP)

! Setting up the tasks
 forall(f in FILES) do
  setdomain(getstart(file(f)), DISKS)           ! Start time (= choice of disk)
  set_task_attributes(file(f), disks, SIZE(f))  ! Resource (disk space) req.
  set_task_attributes(file(f), 1)    ! Duration (= number of disks used)
 end-do

! Limit the number of disks used
 forall(f in FILES) L += getstart(file(f))
 diskuse = maximum(L)

! Minimize the total number of disks used
 if cp_schedule(diskuse) = 0 then
  writeln("Problem infeasible")
 end-if

! Solution printing
 writeln("Number of disks used: ", getsol(diskuse))
 forall(d in 1..getsol(diskuse)) do
  write(d, ":")
  forall(f in FILES | getsol(file(f).start)=d) write(" ",SIZE(f))
  writeln("  space used: ",
          sum(f in FILES | getsol(file(f).start)=d) SIZE(f))
 end-do
 cp_show_stats

end-model
```

5.4.3 Results

Running the model results in the solution shown in Table 5.2, that is, 3 disks are needed for backing up all the files.

Table 5.2: Distribution of files to disks

Disk	File sizes (in kb)	Used space (in Mb)
1	46 87 137 164 253 364 388	1.439
2	55 62 108 372 406 432	1.435
3	114 461 851	1.426

5.4.4 Alternative formulation without scheduling objects

Instead of defining task and resource objects it is also possible to formulate this problem with a 'cumulative' constraint over standard finite domain variables that represent the different attributes of tasks without being grouped into a predefined object. A single 'cumulative' constraint expresses the problem of scheduling a set of tasks on one discrete resource by establishing the following relations between its arguments (five arrays of decision variables for the properties related to tasks—start, duration, end, resource use and size— all indexed by the same set R and a constant or time-indexed resource capacity):

$$\forall j \in R : start_j + duration_j = end_j$$
$$\forall j \in R : use_j \cdot duration_j = size_j$$
$$\forall t \in TIMES : \sum_{j \in R | t \in [UB(start_j)..LB(end_j)]} use_j \leqslant CAP_t$$

where UB stands for 'upper bound' and LB for 'lower bound' of a decision variable.

Let $save_f$ denote the disk used for saving a file f and use_f the space used by the file ($f \in FILES$). As with scheduling objects, the 'start' property of a task corresponds to the disk chosen for saving the file, and the resource requirement of a task is the file size. Since we want to save every file onto a single disk, the 'duration' dur_f is fixed to 1. The remaining task properties 'end' and 'size' (e_f and s_f) that need to be provided in the formulation of 'cumulative' constraints are not really required for our problem; their values are determined by the other three properties.

5.4.5 Implementation

The following Mosel model implements the second model version using the `cumulative` constraint.

```
model "D-4 Bin packing (CP)"
 uses "kalis", "mmsystem"

 setparam("kalis_default_lb", 0)

 declarations
  ND: integer                         ! Number of floppy disks
  FILES = 1..16                       ! Set of files
  DISKS: range                        ! Set of disks

  CAP:  integer                       ! Floppy disk size
  SIZE: array(FILES) of integer       ! Size of files to be saved

  save: array(FILES) of cpvar        ! Disk a file is saved on
  use: array(FILES) of cpvar         ! Space used by file on disk
  dur,e,s: array(FILES) of cpvar     ! Auxiliary arrays for 'cumulative'
  diskuse: cpvar                     ! Number of disks used

  Strategy: array(FILES) of cpbranching ! Enumeration
  FSORT: array(FILES) of integer
 end-declarations

 initializations from 'Data/d4backup.dat'
  CAP SIZE
 end-initializations

! Provide a sufficiently large number of disks
 ND:= ceil((sum(f in FILES) SIZE(f))/CAP)
 DISKS:= 1..ND
 finalize(DISKS)

! Limit the number of disks used
 diskuse = maximum(save)

 forall(f in FILES) do
  setdomain(save(f), DISKS)          ! Every file onto a single disk
  use(f) = SIZE(f)
  dur(f) = 1
 end-do

! Capacity limit of disks
 cumulative(save, dur, e, use, s, CAP)

! Definition of search (place largest files first)
 qsort(SYS_DOWN, SIZE, FSORT)        ! Sort files in decreasing order of size
 forall(f in FILES)
  Strategy(f):= assign_var(KALIS_SMALLEST_MIN, KALIS_MIN_TO_MAX, {save(FSORT(f))})
 cp_set_branching(Strategy)

! Minimize the total number of disks used
 if not cp_minimize(diskuse) then
  writeln("Problem infeasible")
 end-if
```

```
! Solution printing
 writeln("Number of disks used: ", getsol(diskuse))
 forall(d in 1..getsol(diskuse)) do
  write(d, ":")
  forall(f in FILES | getsol(save(f))=d) write(" ",SIZE(f))
  writeln("  space used: ", sum(f in FILES | getsol(save(f))=d) SIZE(f))
 end-do

end-model
```

The solution produced by the execution of this model has the same objective function value, but the distribution of the files to the disks is not exactly the same: this problem has several different optimal solutions, in particular those that may be obtained be interchanging the order numbers of the disks. To shorten the search in such a case it may be useful to add some *symmetry breaking* constraints that reduce the size of the search space by removing a part of the feasible solutions. In the present example we may, for instance, assign the biggest file to the first disk and the second largest to one of the first two disks, and so on, until we reach a lower bound on the number of disks required (a save lower bound estimate is given by rounding up to the next larger integer the sum of files sizes divided by the disk capacity).

5.5 Renewable and non-renewable resources

Besides the distinction 'disjunctive–cumulative' or 'unary–discrete' that we have encountered in the previous sections there are other ways of describing or classifying resources. Another important property is the concept of *renewable* versus *non-renewable* resources. The previous examples have shown instances of renewable resources (machine capacity, manpower *etc.*): the amount of resource used by a task is released at its completion and becomes available for other tasks. In the case of non-renewable resources (*e.g.* money, raw material, intermediate products), the tasks using the resource consume it, that is, the available quantity of the resource is diminuished by the amount used up by processing a task.

Instead of using resources tasks may also *produce* certain quantities of resource. Again, we may have tasks that provide an amount of resource during their execution (renewable resources) or tasks that add the result of their production to a stock of resource (non-renewable resources).

Let us now see how to formulate the following problem: we wish to schedule five jobs P1 to P5 representing two stages of a production process. P1 and P2 produce an intermediate product that is needed by the jobs of the final stage (P3 to P5). For every job we are given its minimum and maximum duration, its cost or, for the jobs of the final stage, its profit contribution. There may be two cases, namely *model A*: the jobs of the first stage produce a given quantity of intermediate product (such as electricity, heat, steam) at every point of time during their execution, this intermediate product is consumed immediately by the jobs of the final stage. *Model B*: the intermediate product results as ouput from the jobs of the first stage and is required as input to start the jobs of the final stage. The intermediate product in model A is a renewable resource and in model B we have the case of a non-renewable resource.

5.5.1 Model formulation

Let $\mathrm{FIRST} = \{\mathrm{P1}, \mathrm{P2}\}$ be the set of jobs in the first stage, $\mathrm{FINAL} = \{\mathrm{P3}, \mathrm{P4}, \mathrm{P5}\}$ the jobs of the second stage, and the set JOBS the union of all jobs. For every job j we are given its minimum and maximum duration $\mathrm{MIND_j}$ and $\mathrm{MAXD_j}$ respectively. $\mathrm{RESAMT_j}$ is the amount of resource needed as input or resulting as output from a job. Furthermore we have a cost $\mathrm{COST_j}$ for jobs j of the first stage and a profit $\mathrm{PROFIT_j}$ for jobs j of the final stage.

Model A (renewable resource)

The case of a renewable resource is formulated by the following model. Notice that the resource

capacity is set to 0 indicating that the only available quantities of resource are those produced by the jobs of the first production stage.

$$
\begin{aligned}
&\textsf{resource } res \\
&\textsf{tasks } \mathrm{task_j}(\mathrm{j} \in \mathrm{JOBS}) \\
&\textsf{maximize } \sum_{\mathrm{j} \in \mathrm{JOBS}} (\mathrm{PROFIT_j} - \mathrm{COST_j}) \times \mathrm{task_j.duration} \\
&res.capacity = 0 \\
&\forall \mathrm{j} \in \mathrm{JOBS} : \mathrm{task_j.start}, \mathrm{task_j.end} \in \{0, ..., \mathrm{HORIZON}\} \\
&\forall \mathrm{j} \in \mathrm{JOBS} : \mathrm{task_j.duration} \in \{\mathrm{MIND_j}, ..., \mathrm{MAXD_j}\} \\
&\forall \mathrm{j} \in \mathrm{FIRST} : \mathrm{task_j}.provision_{res} = \mathrm{RESAMT_j} \\
&\forall \mathrm{j} \in \mathrm{FINAL} : \mathrm{task_j}.requirement_{res} = \mathrm{RESAMT_j}
\end{aligned}
$$

Model B (non-renewable resource)

In analogy to the model A we formulate the second case as follows.

$$
\begin{aligned}
&\textsf{resource } res \\
&\textsf{tasks } \mathrm{task_j}(\mathrm{j} \in \mathrm{JOBS}) \\
&\textsf{maximize } \sum_{\mathrm{j} \in \mathrm{JOBS}} (\mathrm{PROFIT_j} - \mathrm{COST_j}) \times \mathrm{task_j.duration} \\
&res.capacity = 0 \\
&\forall \mathrm{j} \in \mathrm{JOBS} : \mathrm{task_j.start}, \mathrm{task_j.end} \in \{0, ..., \mathrm{HORIZON}\} \\
&\forall \mathrm{j} \in \mathrm{JOBS} : \mathrm{task_j.duration} \in \{\mathrm{MIND_j}, ..., \mathrm{MAXD_j}\} \\
&\forall \mathrm{j} \in \mathrm{FIRST} : \mathrm{task_j}.production_{res} = \mathrm{RESAMT_j} \\
&\forall \mathrm{j} \in \mathrm{FINAL} : \mathrm{task_j}.consumption_{res} = \mathrm{RESAMT_j}
\end{aligned}
$$

However, this model does not entirely correspond to the problem description above since the production of the intermediate product occurs at the start of a task. To remedy this problem we may introduce an auxiliary task $\mathrm{End_j}$ for every job j in the first stage. The auxiliary job has duration 0, the same completion time as the original job and produces the intermediate product in the place of the original job.

$$
\begin{aligned}
&\forall \mathrm{j} \in \mathrm{FIRST} : \mathrm{task_{End_j}.end} = \mathrm{task_j.end} \\
&\forall \mathrm{j} \in \mathrm{FIRST} : \mathrm{task_{End_j}.duration} = 0 \\
&\forall \mathrm{j} \in \mathrm{FIRST} : \mathrm{task_j}.production_{res} = 0 \\
&\forall \mathrm{j} \in \mathrm{FIRST} : \mathrm{task_{End_j}}.production_{res} = \mathrm{RESAMT_j}
\end{aligned}
$$

5.5.2 Implementation

The following Mosel model implements case A. We use the default scheduling solver (function `cp_schedule`) indicating by the value `true` for the optional second argument that we wish to maximize the objective function.

```
model "Renewable resource"
 uses "kalis", "mmsystem"
```

```
 forward procedure solution_found

 declarations
  FIRST = {'P1','P2'}
  FINAL = {'P3','P4','P5'}
  JOBS = FIRST+FINAL

  MIND,MAXD: array(JOBS) of integer  ! Limits on job durations
  RESAMT: array(JOBS) of integer  ! Resource use/production
  HORIZON: integer                ! Time horizon
  PROFIT: array(FINAL) of real    ! Profit from production
  COST: array(JOBS) of real       ! Cost of production
  CAP: integer                    ! Available resource quantity

  totalProfit: cpfloatvar
  task: array(JOBS) of cptask     ! Task objects for jobs
  intermProd: cpresource          ! Non-renewable resource (intermediate prod.)
 end-declarations

 initializations from 'Data/renewa.dat'
  [MIND,MAXD] as 'DUR' RESAMT HORIZON PROFIT COST CAP
 end-initializations

! Setting up resources
 set_resource_attributes(intermProd, KALIS_DISCRETE_RESOURCE, CAP)
 setname(intermProd, "IntP")

! Setting up the tasks
 forall(j in JOBS) do
  setname(task(j), j)
  setduration(task(j), MIND(j), MAXD(j))
  setdomain(getend(task(j)), 0, HORIZON)
 end-do

! Providing tasks
 forall(j in FIRST) provides(task(j), RESAMT(j), intermProd)

! Requiring tasks
 forall(j in FINAL) requires(task(j), RESAMT(j), intermProd)

! Objective function: total profit
 totalProfit = sum(j in FINAL) PROFIT(j)*getduration(task(j)) -
               sum(j in JOBS) COST(j)*getduration(task(j))
 cp_set_solution_callback(->solution_found)
 setparam("KALIS_MAX_COMPUTATION_TIME", 30)

! Solve the problem
 starttime:= gettime
 if cp_schedule(totalProfit,true)=0 then
  exit(1)
 end-if

! Solution printing
 writeln("Total profit: ", getsol(totalProfit))
 writeln("Job\tStart\tEnd\tDuration")
 forall(j in JOBS)
  writeln(j, "\t ", getsol(getstart(task(j))), "\t ", getsol(getend(task(j))),
          "\t ", getsol(getduration(task(j))))

 procedure solution_found
  writeln(gettime-starttime , " sec. Solution found with total profit = ",
          getsol(totalProfit))
  forall(j in JOBS)
   write(j, ": ", getsol(getstart(task(j))), "-", getsol(getend(task(j))),
          "(", getsol(getduration(task(j))), ") ")
  writeln
 end-procedure

end-model
```

The model for case B adds the two auxiliary tasks (forming the set ENDFIRST) that mark the completion of the jobs in the first stage. The only other difference are the task properties produces and consumes that define the resource constraints. We only repeat the relevant part of the model:

```
 declarations
  FIRST = {'P1','P2'}
  ENDFIRST = {'EndP1', 'EndP2'}
  FINAL = {'P3','P4','P5'}
  JOBS = FIRST+ENDFIRST+FINAL

  MIND,MAXD: array(JOBS) of integer  ! Limits on job durations
  RESAMT: array(JOBS) of integer  ! Resource use/production
  HORIZON: integer                ! Time horizon
  PROFIT: array(FINAL) of real    ! Profit from production
  COST: array(JOBS) of real       ! Cost of production
  CAP: integer                    ! Available resource quantity

  totalProfit: cpfloatvar
  task: array(JOBS) of cptask     ! Task objects for jobs
  intermProd: cpresource          ! Non-renewable resource (intermediate prod.)
 end-declarations

 initializations from 'Data/renewb.dat'
  [MIND,MAXD] as 'DUR' RESAMT HORIZON PROFIT COST CAP
 end-initializations

! Setting up resources
 set_resource_attributes(intermProd, KALIS_DISCRETE_RESOURCE, CAP)
 setname(intermProd, "IntP")

! Setting up the tasks
 forall(j in JOBS) do
  setname(task(j), j)
  setduration(task(j), MIND(j), MAXD(j))
  setdomain(getend(task(j)), 0, HORIZON)
 end-do

! Production tasks
 forall(j in ENDFIRST) produces(task(j), RESAMT(j), intermProd)
 forall(j in FIRST) getend(task(j)) = getend(task("End"+j))

! Consumer tasks
 forall(j in FINAL) consumes(task(j), RESAMT(j), intermProd)
```

5.5.3 Results

The behavior of the (default) search and the results of the two models are considerably different. The optimal solution with an objective of 344.9 for case B represented in Figure 5.3 is proven within a fraction of a second. Finding a good solution for case A takes several seconds on a standard PC; finding the optimal solution (see Figure 5.2) and proving its optimality requires several minutes of running time. The main reason for this poor behavior of the search is our choice of the objective function: the cost-based objective function does not propagate well and therefore does not help with pruning the search tree. A better choice for objective functions in scheduling problems generally are criteria involving the task decision variables (start, duration, or completion time, particularly the latter).

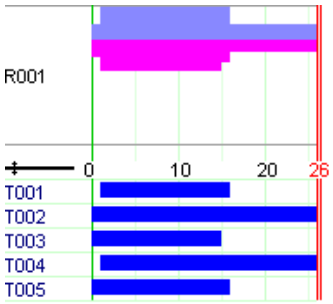

Figure 5.2: Solution for case A (resource provision and requirement)

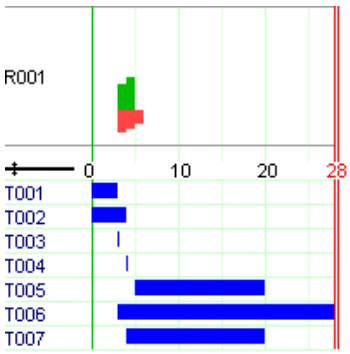

Figure 5.3: Solution for case B (resource production and consumption)

5.5.4 Alternative formulation without scheduling objects

Instead of defining task and resource objects we may equally formulate this problem with a 'producer-consumer' constraint over standard finite domain variables that represent the different attributes of tasks without being grouped into a predefined object. A single 'producer-consumer' constraint expresses the problem of scheduling a set of tasks producing or consuming a non-renewable resource by establishing the following relations between its arguments (seven arrays of decision variables for the properties related to tasks—start, duration, end, per unit and cumulated resource production, per unit and cumulated resource consumption— all indexed by the same set R):

$$\forall \mathrm{j} \in \mathrm{R} : \mathrm{start}_\mathrm{j} + \mathrm{duration}_\mathrm{j} = \mathrm{end}_\mathrm{j}$$
$$\forall \mathrm{j} \in \mathrm{R} : \mathrm{produce}_\mathrm{j} \cdot \mathrm{duration}_\mathrm{j} = \mathrm{psize}_\mathrm{j}$$
$$\forall \mathrm{j} \in \mathrm{R} : \mathrm{consume}_\mathrm{j} \cdot \mathrm{duration}_\mathrm{j} = \mathrm{csize}_\mathrm{j}$$
$$\forall \mathrm{t} \in \mathrm{TIMES} : \sum_{\mathrm{j} \in \mathrm{R} | \mathrm{t} \in [\mathrm{UB}(\mathrm{start}_\mathrm{j}) .. \mathrm{LB}(\mathrm{end}_\mathrm{j})]} (\mathrm{produce}_\mathrm{j} - \mathrm{consume}_\mathrm{j}) \geqslant 0$$

where UB stands for 'upper bound' and LB for 'lower bound' of a decision variable.

5.5.5 Implementation

The following Mosel model implements the second model version using the `producer_consumer` constraint.

```
model "Non-renewable resource"
 uses "kalis"

 setparam("KALIS_DEFAULT_LB", 0)

 declarations
  FIRST = {'P1','P2'}
  ENDFIRST = {'EndP1', 'EndP2'}
  FINAL = {'P3','P4','P5'}
  JOBS = FIRST+ENDFIRST+FINAL
  PCJOBS = ENDFIRST+FINAL

  MIND,MAXD: array(JOBS) of integer  ! Limits on job durations
  RESAMT: array(JOBS) of integer  ! Resource use/production
  HORIZON: integer                ! Time horizon
  PROFIT: array(FINAL) of real    ! Profit from production
  COST: array(JOBS) of real       ! Cost of production
  CAP: integer                    ! Available resource quantity

  totalProfit: cpfloatvar
  fstart,fdur,fcomp: array(FIRST) of cpvar! Start, duration & completion of jobs
  start,dur,comp: array(PCJOBS) of cpvar  ! Start, duration & completion of jobs
  produce,consume: array(PCJOBS) of cpvar ! Production/consumption per time unit
  psize,csize: array(PCJOBS) of cpvar     ! Cumulated production/consumption
 end-declarations

 initializations from 'Data/renewb.dat'
  [MIND,MAXD] as 'DUR' RESAMT HORIZON PROFIT COST CAP
 end-initializations

! Setting up the tasks
 forall(j in PCJOBS) do
  setname(start(j), j)
  setdomain(dur(j), MIND(j), MAXD(j))
  setdomain(comp(j), 0, HORIZON)
  start(j) + dur(j) = comp(j)
 end-do
 forall(j in FIRST) do
  setname(fstart(j), j)
  setdomain(fdur(j), MIND(j), MAXD(j))
  setdomain(fcomp(j), 0, HORIZON)
  fstart(j) + fdur(j) = fcomp(j)
 end-do

! Production tasks
 forall(j in ENDFIRST) do
  produce(j) = RESAMT(j)
  consume(j) = 0
 end-do
 forall(j in FIRST) fcomp(j) = comp("End"+j)

! Consumer tasks
 forall(j in FINAL) do
  consume(j) = RESAMT(j)
  produce(j) = 0
 end-do

! Resource constraint
 producer_consumer(start, comp, dur, produce, psize, consume, csize)

! Objective function: total profit
 totalProfit = sum(j in FINAL) PROFIT(j)*dur(j) -
               sum(j in FIRST) COST(j)*fdur(j)

 if not cp_maximize(totalProfit) then
  exit(1)
 end-if

 writeln("Total profit: ", getsol(totalProfit))
```

```
 writeln("Job\tStart\tEnd\tDuration")
 forall(j in FIRST)
  writeln(j, "\t ", getsol(fstart(j)), "\t ", getsol(fcomp(j)),
          "\t ", getsol(fdur(j)))
 forall(j in PCJOBS)
  writeln(j, "\t ", getsol(start(j)), "\t ", getsol(comp(j)),
          "\t ", getsol(dur(j)))

end-model
```

This model generates the same solution as the previous model version with a slightly longer running time (though still just a fraction of a second on a standard PC).

5.6 Extensions: setup times, resource choice, usage profiles

5.6.1 Setup times

Consider once more the problem of planning the production of paint batches introduced in Section 3.8. Between the processing of two batches the machine needs to be cleaned. The cleaning (or *setup*) times incurred are sequence-dependent and asymetric. The objective is to determine a production cycle of the shortest length.

5.6.1.1 Model formulation

For every job j ($j \in JOBS = \{1, ..., NJ\}$), represented by a task object $task_j$, we are given its processing duration DUR_j. We also have a matrix of cleaning times $CLEAN$ with entries $CLEAN_{jk}$ indicating the duration of the cleaning operation if task k succedes task j. The machine processing the jobs is modeled as a resource res of unitary capacity, thus stating the disjunctions between the jobs.

With the objective to minimize the makespan (completion time of the last batch) we obtain the following model:

$$
\begin{aligned}
&\text{resource } res \\
&\text{tasks } task_j (j \in JOBS) \\
&\text{minimize } finish \\
&finish = maximum_{j \in JOBS}(task_j.end) \\
&res.capacity = 1 \\
&\forall j \in JOBS : task_j.duration = DUR_j \\
&\forall j \in JOBS : task_j.requirement_{res} = 1 \\
&\forall j, k \in JOBS : setup(task_j, task_k) = CLEAN_{jk}
\end{aligned}
$$

The tricky bit in the formulation of the original problem is that we wish to minimize the cycle time, that is, the completion of the last job plus the setup required between the last and the first jobs in the sequence. Since our task-based model does not contain any information about the sequence or rank of the jobs we introduce auxiliary variables $firstjob$ and $lastjob$ for the index values of the jobs in the first and last positions of the production cycle, and a variable $cleanlf$ for the duration of the setup operation between the last and first tasks. The following constraints express the relations between these variables

and the task objects:

$$
\begin{aligned}
&\mathrm{firstjob}, \mathrm{lastjob} \in \mathrm{JOBS} \\
&\mathrm{firstjob} \neq \mathrm{lastjob} \\
&\forall j \in \mathrm{JOBS} : \mathrm{task}_j.\mathrm{end} = \mathrm{finish} \Leftrightarrow \mathrm{lastjob} = j \\
&\forall j \in \mathrm{JOBS} : \mathrm{task}_j.\mathrm{start} = 1 \Leftrightarrow \mathrm{firstjob} = j \\
&\mathrm{cleanlf} = \mathrm{CLEAN}_{\mathrm{lastjob,firstjob}}
\end{aligned}
$$

Minimizing the cycle time then corresponds to minimizing the sum $\mathrm{finish} + \mathrm{cleanlf}$.

5.6.1.2 Implementation

The following Mosel model implements the task-based model formulated above. The setup times between tasks are set with the procedure `setsetuptimes` indicating the two task objects and the corresponding duration value.

```
model "B-5 Paint production (CP)"
 uses "kalis", "mmsystem"

 declarations
  NJ = 5                              ! Number of paint batches (=jobs)
  JOBS=1..NJ

  DUR: array(JOBS) of integer         ! Durations of jobs
  CLEAN: array(JOBS,JOBS) of integer  ! Cleaning times between jobs

  task: array(JOBS) of cptask
  res: cpresource

  firstjob,lastjob,cleanlf,finish: cpvar
  L: cpvarlist
  cycleTime: cpvar                    ! Objective variable
 end-declarations

 initializations from 'Data/b5paint.dat'
  DUR CLEAN
 end-initializations

! Setting up the resource (formulation of the disjunction of tasks)
 set_resource_attributes(res, KALIS_UNARY_RESOURCE, 1)

! Setting up the tasks
 forall(j in JOBS) getstart(task(j)) >= 1                    ! Start times
 forall(j in JOBS) set_task_attributes(task(j), DUR(j), res)  ! Dur.s + disj.
 forall(j,k in JOBS) setsetuptime(task(j), task(k), CLEAN(j,k), CLEAN(k,j))
                                     ! Cleaning times between batches

! Cleaning time at end of cycle (between last and first jobs)
 setdomain(firstjob, JOBS); setdomain(lastjob, JOBS)
 firstjob <> lastjob
 forall(j in JOBS) equiv(getend(task(j))=getmakespan, lastjob=j)
 forall(j in JOBS) equiv(getstart(task(j))=1, firstjob=j)
 cleanlf = element(CLEAN, lastjob, firstjob)

 forall(j in JOBS) L += getend(task(j))
 finish = maximum(L)

! Objective: minimize the duration of a production cycle
 cycleTime = finish - 1 + cleanlf

! Solve the problem
 if cp_schedule(cycleTime) = 0 then
  writeln("Problem is infeasible")
  exit(1)
```

```
 end-if
 cp_show_stats

! Solution printing
 declarations
  SUCC: array(JOBS) of integer
 end-declarations

 forall(j in JOBS)
  forall(k in JOBS)
   if getsol(getstart(task(k))) = getsol(getend(task(j)))+CLEAN(j,k) then
    SUCC(j):= k
    break
   end-if
 writeln("Minimum cycle time: ", getsol(cycleTime))
 writeln("Sequence of batches:\nBatch Start Duration Cleaning")
 forall(k in JOBS)
  writeln(formattext("  %d%7d%8d%9d", k, getsol(task(k).start), DUR(k),
          if(SUCC(k)>0, CLEAN(k,SUCC(k)), cleanlf.sol)))

end-model
```

5.6.1.3 Results

The results are similar to those reported in Section 3.8. It should be noted here that this model formulation is less efficient, in terms of search nodes and running times, than the previous model versions, and in particular the 'cycle' constraint version presented in Section 3.10. However, the task-based formulation is more generic and easier to extend with additional features than the problem-specific formulations in the previous model versions.

The graphical representation of the results looks as follows (Figure 5.4).

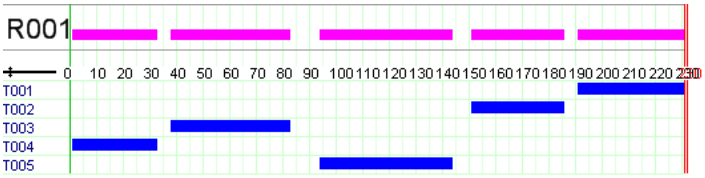

Figure 5.4: Solution graph

5.6.2 Alternative resources

Quite frequently in scheduling applications the assignment of tasks to resources is not entirely fixed from the outset. For instance, there may be a group of resources (identical or not) among which to choose for a given processing step. If the resources are identical the task will have the same properties (duration, amount of resource use/provision/consumption/production) independent of the resource used for its processing. Otherwise, if resources are non-identical there is some resource-dependent data component.

We shall show here how to formulate a tiny example of four tasks that may be processed by two non-identical machines. Each task must be assigned to a single machine. The resource usage per task depends on the machine it is assigned to.

5.6.2.1 Model formulation

For each task j in the set of tasks $TASKS$ we are given a fixed duration DUR_j and a machine-dependent amount of resource REQ_{jm} that is required per time unit for the whole duration of the task. Both machines have the same capacity CAP.

The resulting model looks as follows.

$$
\begin{aligned}
&\textbf{resource } res_m (m \in MACH)\\
&\textbf{tasks } task_j (j \in TASKS)\\
&\textbf{minimize } makespan\\
&\forall m \in MACH : res_m.capacity = CAP\\
&\forall j \in TASKS : task_j.start, task_j.end \in \{0, ..., HORIZON\}\\
&\forall j \in TASKS : task_j.duration = DUR_t\\
&\forall j \in TASKS, m \in MACH : task_j.assignment_{res_m} \in \{0, 1\}\\
&\forall j \in TASKS : \sum_{m \in MACH} task_j.assignment_{res_m} = 1\\
&\forall j \in TASKS, m \in MACH : task_j.requirement_{res_m} \in \{0, REQ_{jm}\}
\end{aligned}
$$

5.6.2.2 Implementation

This is a Mosel implementation of the scheduling problem with alternative resources.

```
 uses "kalis", "mmsystem"

 setparam("KALIS_DEFAULT_LB", 0)

 forward procedure print_solution

 declarations
  TASKS = {"a","b","c","d"}           ! Index set of tasks
  MACH = {"M1", "M2"}                 ! Index set of resources
  USE: array(TASKS,MACH) of integer   ! Machine-dependent res. requirement
  DUR: array(TASKS) of integer        ! Durations of tasks

  task: array(TASKS) of cptask        ! Tasks
  res: array(MACH) of cpresource      ! Resources
 end-declarations

 DUR::(["a","b","c","d"])[7, 9, 8, 5]
 USE::(["a","b","c","d"],["M1","M2"])[
      4, 3,
      2, 3,
      2, 1,
      4, 5]

! Define discrete resources
 forall(m in MACH)
  set_resource_attributes(res(m), KALIS_DISCRETE_RESOURCE, 5)

! Define tasks with machine-dependent resource usages
 forall(j in TASKS) do
  task(j).duration:= DUR(j)
  task(j).name:= j
  requires(task(j), union(m in MACH) {resusage(res(m), USE(j,m))})
 end-do

 cp_set_solution_callback(->print_solution)
 starttime:=timestamp

! Solve the problem
 if cp_schedule(getmakespan)=0 then
  writeln("No solution")
  exit(0)
 end-if

! Solution printing
 forall(j in TASKS)
  writeln(j, ": ", getsol(getstart(task(j))), " - ", getsol(getend(task(j))))
```

```
 forall(t in 1..getsol(getmakespan)) do
  write(strfmt(t-1,2), ": ")

  ! We cannot use 'getrequirement' here to access solution information
  ! (it returns a value corresponding to the current state, that is 0)
  forall(j in TASKS | t>getsol(task(j).start) and t<=getsol(getend(task(j))))
   write(formattext("%s:%d  ",j,
         sum(m in MACH) USE(j,m)*getsol(getassignment(task(j),res(m)))) )
  writeln
 end-do

! *************************************************************

! Print solutions during enumeration at the node where they are found
 procedure print_solution
  writeln(timestamp-starttime, "sec. Solution: ", getsol(getmakespan))

  forall(m in MACH) do
   writeln(m, ":")

   forall(t in 0..getsol(getmakespan)-1) do
    write(strfmt(t,2), ": ")
    forall(j in TASKS | getrequirement(task(j), res(m), t)>0)
     write(j, ":", getrequirement(task(j), res(m), t), "  " )
    writeln("(total ", sum(j in TASKS) getrequirement(task(j), res(m), t), ")")
   end-do

  end-do
 end-procedure

end-model
```

A new feature introduced by this example are the `resusage` objects that are employed in the definition of the resource requirement constraints for the tasks. A `resusage` consists of a resource object and a resource usage (specified by a constant, lower and upper bounds, or a complete profile — see Section 5.6.3). In this example we use the simplest version with a fixed resource usage value. Optional third and forth arguments to `requires` can be defined to specify the minimum and maximum number of resources to be used by a task. The default version (employed in this example) will choose a single resource from the set of alternative resources.

The implementation also shows how to retrieve solution information about resource assignments at a solution node (that is, during the enumeration) and after the search. At the solution node the function `getrequirement` returns the desired resource usage information; for a solution summary after terminating the search we need to work with the decision variable obtained through `getassignment`.

The formulation of `produces`, `consumes`, and `provides` constraints with `resusage` objects is analogous to what we have shown here for `requires`.

Note: when implementing alternative resources you always have to use the resource type `KALIS_DISCRETE_RESOURCES`, setting a capacity limit of 1 for modeling disjunctive resources.

5.6.2.3 Results

Figure 5.5 shows two solutions generated by the execution of our model. The first solution uses a lesser total amount resource but the second one is optimal with respect to the objective of minimizing the makespan.

Note: leaving the decision about the resource choice open for the solver considerably increases the complexity of a scheduling problem. We therefore recommend to use this feature sparingly. Optimization problems involving resource assignment and sequencing aspects are often more easily solved by a decompostion approach (see the Xpress Whitepaper *Hybrid MIP/CP solving with Xpress Optimizer and Xpress Kalis*).

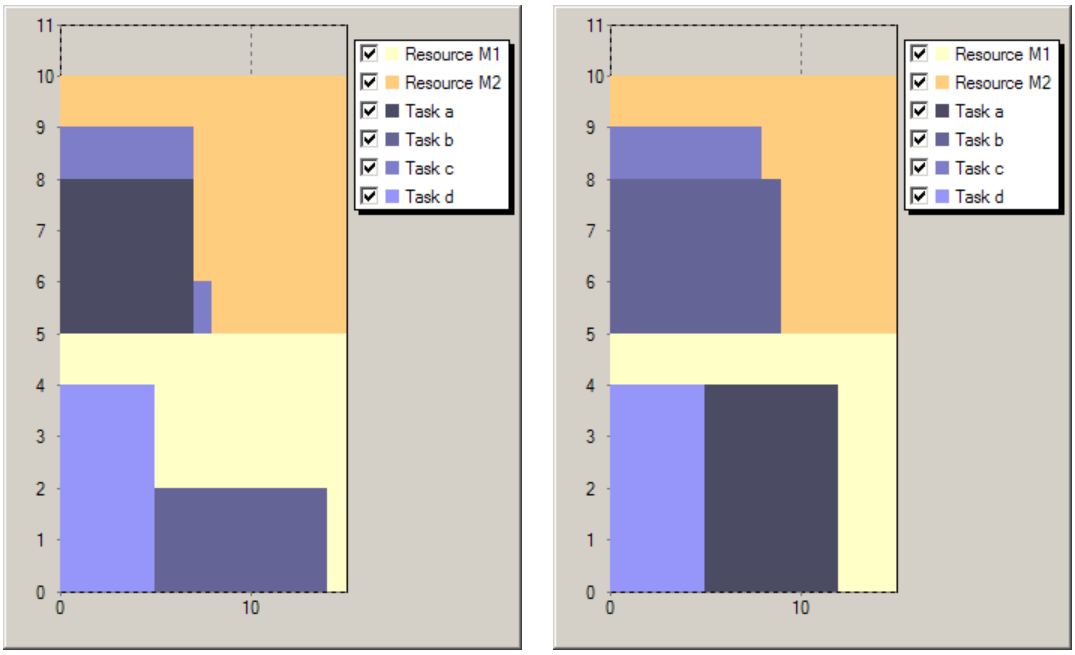

Figure 5.5: Solutions for scheduling with resource choice

5.6.3 Resource profiles

In all preceding examples we have worked with the assumption that a task is defined by a constant, though not necessarily fixed resource requirement (respectively resource provision, consumption, or production). Following this line of thought, a sequence of processesing steps with different resource usages would be represented by a corresponding number of tasks, each with a constant resource usage. However, the number of `cptask` objects required by such a formulation may be prohibitively large. Kalis therefore offers the possibility to define profiled tasks, that is, tasks with a non-constant resource usage profile. A task resource usage profile states for every time unit of the execution of the task the corresponding amount of resource. For example the four tasks in Figure 5.6 have the profiles $\mathrm{Profile}_{\mathrm{a}}$: (12, 12, 6, 2, 2, 2, 2), $\mathrm{Profile}_{\mathrm{b}}$: (12, 12, 6, 2, 2, 2, 2, 2, 2), $\mathrm{Profile}_{\mathrm{c}}$: (12, 12, 3, 3, 3, 3, 3, 3), and $\mathrm{Profile}_{\mathrm{d}}$: (6, 6, 6, 6, 6) respectively.

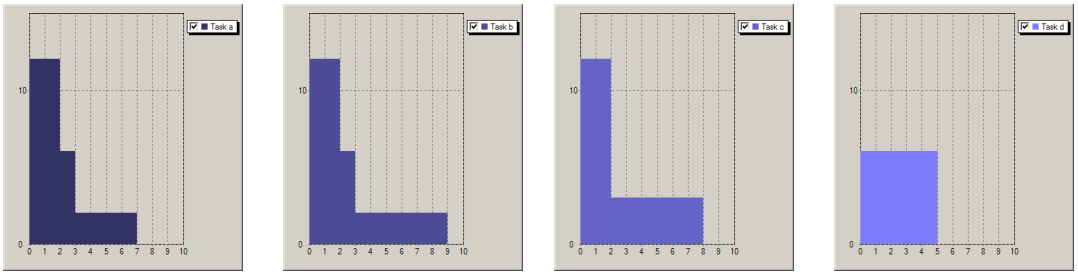

Figure 5.6: Task profiles

5.6.3.1 Model formulation

Let $\mathrm{TASKS} = \{'\mathrm{a}','\mathrm{b}','\mathrm{c}','\mathrm{d}'\}$ be the set of tasks and res a resource of capacity CAP required by all tasks for their execution. With the objective of minimizing the makespan of the schedule we may

formulate the model as follows.

$$
\begin{aligned}
&\text{resource } res \\
&\text{tasks } \mathrm{task_j} (\mathrm{j} \in \mathrm{TASKS}) \\
&\text{minimize } makespan \\
&res.\mathrm{capacity} = \mathrm{CAP} \\
&\forall \mathrm{j} \in \mathrm{TASKS} : \mathrm{task_j}.\mathrm{start}, \mathrm{task_j}.\mathrm{end} \in \{0, ..., \mathrm{HORIZON}\} \\
&\forall \mathrm{j} \in \mathrm{TASKS} : \mathrm{task_j}.\mathrm{duration} = \mathrm{DUR_j} \\
&\forall \mathrm{j} \in \mathrm{TASKS} : \mathrm{task_j}.\mathrm{requirement}_{res} = \mathrm{Profile_j}
\end{aligned}
$$

This model formulation differs from a standard cumulative scheduling model (see Section 5.4 only in a single point: the resource requirement $\mathrm{Profile_j}$ of a task j is a list of values and not just a single constant.

5.6.3.2 Implementation

Let us now take a look at the model formulation with Mosel.

```
uses "kalis"

setparam("KALIS_DEFAULT_LB", 0)

forward procedure print_solution

declarations
 TASKS = {"a","b","c","d"}           ! Index set of tasks
 Profile: array(TASKS) of list of integer   ! Task profiles
 DUR: array(TASKS) of integer        ! Durations of tasks

 task: array(TASKS) of cptask        ! Tasks
 res: cpresource                     ! Cumulative resource
end-declarations

DUR::(["a","b","c","d"])[7, 9, 8, 5]
Profile("a"):= [12, 12, 6, 2, 2, 2, 2]
Profile("b"):= [12, 12, 6, 2, 2, 2, 2, 2, 2]
Profile("c"):= [12, 12, 3, 3, 3, 3, 3, 3]
Profile("d"):= [6, 6, 6, 6, 6]

! Define a discrete resource
set_resource_attributes(res, KALIS_DISCRETE_RESOURCE, 18)
res.name:="machine"

! Define tasks with profiled resource usage
forall(t in TASKS) do
 task(t).duration:= DUR(t)
 task(t).name:= t
 requires(task(t), resusage(res, Profile(t)))
end-do

cp_set_solution_callback(->print_solution)
starttime:=timestamp

! Solve the problem
if cp_schedule(getmakespan)=0 then
 writeln("No solution")
 exit(0)
end-if

! Solution printing
writeln("Schedule with makespan ", getsol(getmakespan), ":")
forall(t in TASKS)
 writeln(t, ": ", getsol(getstart(task(t))), " - ", getsol(getend(task(t))))

! ************************************************************
```

```
! Print solutions during enumeration at the node where they are found.
! 'getrequirement' can only be used here to access solution information,
! not after the enumeration (it returns a value corresponding to the
! current state, that is, 0 after the enumeration).
 procedure print_solution
  writeln(timestamp-starttime, "sec. Solution: ", getsol(getmakespan))

  forall(i in 0..getsol(getmakespan)-1) do
   write(strfmt(i,2), ": ")
   forall(t in TASKS | getrequirement(task(t), res, i)>0)
    write(t, ":", getrequirement(task(t), res, i), "  " )
   writeln("(total ", sum(t in TASKS) getrequirement(task(t), res, i), ")")
  end-do
 end-procedure

end-model
```

The statement of the resource constraints (`requires`) uses an additional type of scheduling object for the definition of the resource profiles, namely `resusage`. We have already encountered this object in Section 5.6.2 where it was employed in the definition of sets of alternative resources. In the present case there is only a single resource that must be used by all tasks, we now employ `resusage` to define resource usage profiles.

5.6.3.3 Results

The chart in Figure 5.7 shows an optimal schedule of the four tasks on a resource with capacity 18. All tasks are completed by time 11.

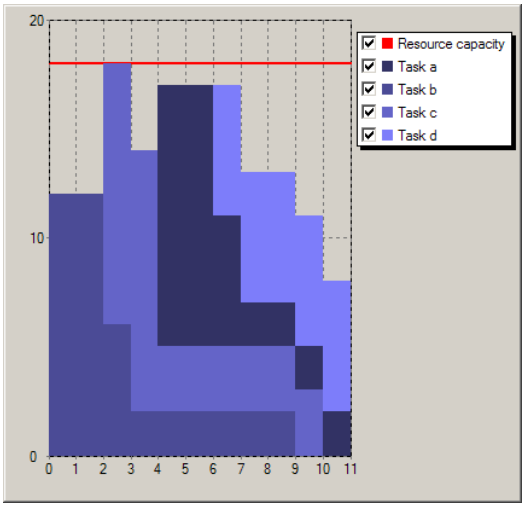

Figure 5.7: Solution for scheduling with resource profile

5.6.4 Resource idle times and preemption of tasks

In the examples of cumulative scheduling we have looked at so far resources are defined with a fixed constant capacity. In reality, it often happens that the resource availability is not constant over time: a resource 'staff', for instance, is likely to show variations corresponding to the presence or absence of personnel. Such changes to the resource level can be defined periodwise with the subroutine `setcapacity`. In the case of machine resources there may be scheduled downtimes for maintenance or other breaks, such as week-ends, relating to working hours. Any such periods that are known from the outset should be marked as 'unavailable for processing tasks'.

From the perspective of tasks there are two cases of resource unavailability: (1) any task starting before the resource unavailability must also be finished before the begin of this period (generally known as *non-preemptive* scheduling), and (2) a task started before the unavailability resumes its processing once the resource becomes available again (the case of *preemptive* scheduling).

With Kalis, the first case applies when resource capacities are set to 0 using `setcapacity` (independent of the type of the task). The second case is modeled by using profiled tasks and indicating resource unavailability with `setidletimes`.

For simplicity's sake, we merely consider two tasks, a and b, that are to be scheduled on a resource of capacity 3 that is unavailable in certain time periods. Task a has a constant resource requirement of 1 for 4 units of time, task b has the resource usage profile (1,1,1,2,2,2,1,1). Both tasks are preemptive.

5.6.4.1 Model formulation

Let $TASKS = \{'a','b'\}$ be the set of tasks and res a resource of capacity CAP required by the tasks for their execution. The set $IDLESET$ contains the time periods during which the resource is unavailable preempting the processing of tasks. With the objective of minimizing the makespan of the schedule we may formulate the model as follows.

$$
\begin{aligned}
&\text{resource } res \\
&\text{tasks } task_j (j \in TASKS) \\
&\text{minimize } makespan \\
&res.capacity = CAP \\
&res.idletimes = IDLESET \\
&\forall j \in TASKS : task_j.start, task_j.end \in \{0, ..., HORIZON\} \\
&\forall j \in TASKS : task_j.duration \geqslant DUR_j \\
&\forall j \in TASKS : task_j.requirement_{res} = Profile_j
\end{aligned}
$$

Please notice that we do not fix the durations of the tasks in this formulation: we may indicate (lower or upper) bounds to restrict the total duration of a task, but if we want to leave room for preemptions the bounds on the durations of tasks must include some slack.

5.6.4.2 Implementation

The following model shows how to implement our example problem with Mosel.

```
model "Preemption"
 uses "kalis", "mmsystem"

 setparam("KALIS_DEFAULT_LB", 0)

 forward procedure print_solution

 declarations
  aprofile,bprofile: list of integer
  a,b: cptask
  res: cpresource
 end-declarations

! Define a discrete resource
 set_resource_attributes(res, KALIS_DISCRETE_RESOURCE, 3)

! Create a 'hole' in the availability of the resource:
! Uncomment one of the following two lines to compare their effect
! setcapacity(res, 1, 2, 0);  setcapacity(res, 7, 8, 0)
 setidletimes(res, (1..2)+(7..8))

! Define two tasks (a: constant resource use, b: resource profile)
```

```
 a.duration >= 4
 a.name:= "task_a"
 aprofile:= [1,1,1,1]

 b.duration >= 8
 b.name:= "task_b"
 bprofile:= [1,1,1,2,2,2,1,1]

! Resource usage constraints
 requires(a, resusage(res,aprofile))
 requires(b, resusage(res,bprofile))

 cp_set_solution_callback(->print_solution)

! Solve the problem
 if cp_schedule(getmakespan)<>0 then
  cp_show_sol
 else
  writeln("No solution")
 end-if

! ************************************************************

 procedure print_solution
  writeln("Solution: ", getsol(getmakespan))
  writeln("Schedule: a: ", getsol(getstart(a)), "-", getsol(getend(a))-1,
          ", b: ", getsol(getstart(b)), "-", getsol(getend(b))-1)
  writeln("Resource usage:")
  forall(t in 0..getsol(getmakespan)-1)
   writeln(formattext("%5d: Cap: %d, a:%d, b:%d", t, getcapacity(res,t),
           getrequirement(a, res, t), getrequirement(b, res, t)) )
 end-procedure

end-model
```

There are several equivalent ways of stating the constant resource usage of task a. Instead of defining explicitly the complete profile we may simply write

```
 requires(a, resusage(res,1))
```

or even shorter:

```
 requires(a, 1, res)
```

5.6.4.3 Results

The model shown above produces the following solution output.

```
Solution: 12
Schedule: a: 0-5, b: 0-11
Resource usage:
    0: Cap: 3, a:1, b:1
    1: Cap: 3, a:0, b:0
    2: Cap: 3, a:0, b:0
    3: Cap: 3, a:1, b:1
    4: Cap: 3, a:1, b:1
    5: Cap: 3, a:1, b:2
    6: Cap: 3, a:0, b:2
    7: Cap: 3, a:0, b:0
    8: Cap: 3, a:0, b:0
    9: Cap: 3, a:0, b:2
   10: Cap: 3, a:0, b:1
   11: Cap: 3, a:0, b:1
```

The same model, replacing the line

```
setidletimes(res, (1..2)+(7..8))
```

by this version indicating non-preemptive resource unavailability

```
setcapacity(res, 1, 2, 0);  setcapacity(res, 7, 8, 0)
```

results in the following solution where the tasks are placed in such a way that they do not overlap with any 'hole' in resource availability.

```
Solution: 17
Schedule: a: 3-6, b: 9-16
Resource usage:
    0: Cap: 3, a:0, b:0
    1: Cap: 0, a:0, b:0
    2: Cap: 0, a:0, b:0
    3: Cap: 3, a:1, b:0
    4: Cap: 3, a:1, b:0
    5: Cap: 3, a:1, b:0
    6: Cap: 3, a:1, b:0
    7: Cap: 0, a:0, b:0
    8: Cap: 0, a:0, b:0
    9: Cap: 3, a:0, b:1
   10: Cap: 3, a:0, b:1
   11: Cap: 3, a:0, b:1
   12: Cap: 3, a:0, b:2
   13: Cap: 3, a:0, b:2
   14: Cap: 3, a:0, b:2
   15: Cap: 3, a:0, b:1
   16: Cap: 3, a:0, b:1
```

It is also possible to define both, preemptive and non-preemptive times of unavailability for the same resource. For example, if times 1-2 correspond to a weekend where products may remain on the machine `R` and times 7-8 are required for maintenance operations during which the machine must imperatively be empty we would state in our model:

```
setidletimes(res, 1, 2, 0)
setcapacity(res, (7..8))
```

In this case the execution of task `a` may start at time 0, whereas task `b` will again be processed in times 9-16.

5.7 Enumeration

In the previous scheduling examples we have used the default enumeration for scheduling problems, invoked by the optimization function `cp_schedule`. The built-in search strategies used by the solver in this case are particularly suited if we wish to minimize the completion time of a schedule. With other objectives the built-in strategies may not be a good choice, especially if the model contains decision variables that are not part of scheduling objects (the built-in strategies always enumerate first the scheduling objects) or if we wish to *maximize* an objective. *kalis* makes it therefore possible to use the standard optimization functions `cp_minimize` and `cp_maximize` in models that contain scheduling objects, the default search strategies employed by these optimization functions being different from the scheduling-specific ones. In addition, the user may also define his own problem-specific enumeration as shown in the following examples.

User-defined enumeration strategies for scheduling problems may take two different forms: *variable-based* and *task-based*. The former case is the subject of Chapter 4, and we only give a small example here (Section 5.7.1). The latter will be explained with some more detail by the means of a job-shop scheduling example.

5.7.1 Variable-based enumeration

When studying the problem solving statistics for the bin packing problem in Section 5.4 we find that the enumeration using the default scheduling search strategies requires several hundred nodes to prove optimality. With a problem-specific search strategy we may be able to do better! Indeed, we shall see that a greedy-type strategy, assigning the largest files first, to the first disk with sufficient space is clearly more efficient.

5.7.1.1 Using `cp_minimize`

The only decisions to make in this problem are the assignments of files to disks, that is, choosing a value for the start time variables of the tasks. The following lines of Mosel code order the tasks in decreasing order of file sizes and define an enumeration strategy for their start times assigning each the smallest possible disk number first. Notice that the sorting subroutine `qsort` is defined by the module *mmsystem* that needs to be loaded with a `uses` statement at the beginning of the model.

```
declarations
 Strategy: array(range) of cpbranching
 FSORT: array(FILES) of integer
end-declarations

qsort(SYS_DOWN, SIZE, FSORT)       ! Sort files in decreasing order of size
forall(f in FILES)
 Strategy(f):=assign_var(KALIS_SMALLEST_MIN, KALIS_MIN_TO_MAX,
                         {getstart(file(FSORT(f)))})
cp_set_branching(Strategy)

if not cp_minimize(diskuse) then writeln("Problem infeasible"); end-if
```

The choice of the variable selection criterion (first argument of `assign_var`) is not really important here since every strategy $Strategy_f$ only applies to a single variable and hence no selection takes place. Equivalently, we may have written

```
declarations
 Strategy: cpbranching
 FSORT: array(FILES) of integer
 LS: cpvarlist
end-declarations

qsort(SYS_DOWN, SIZE, FSORT)       ! Sort files in decreasing order of size
forall(f in FILES)
 LS+=getstart(file(FSORT(f)))
Strategy:=assign_var(KALIS_INPUT_ORDER, KALIS_MIN_TO_MAX, LS)
cp_set_branching(Strategy)

if not cp_minimize(diskuse) then writeln("Problem infeasible"); end-if
```

With this search strategy the first solution found uses 3 disks, that is, we immediately find an optimal solution. The whole search terminates after 17 nodes and takes only a fraction of the time needed by the default scheduling or minimization strategies.

5.7.1.2 Using `cp_schedule`

The scheduling search consists of a pretreatment (*shaving*) phase and two main phases (`KALIS_INITIAL_SOLUTION` and `KALIS_OPTIMAL_SOLUTION`) for which the user may specify enumeration strategies by calling `cp_set_schedule_search` with the corresponding phase selection. The 'initial solution' phase aims at providing quickly a good solution whereas the 'optimal solution' phase proves optimality. Any search limits such as maximum number of nodes apply separately to each phase, an overall time limit (parameter `KALIS_MAX_COMPUTATION_TIME`) only applies to the last phase.

The definition of variable-based branching schemes for the scheduling search is done in exactly the same way as what we have seen previously for standard search with `cp_minimize` or `cp_maximize`, replacing `cp_set_strategy` by `cp_set_schedule_strategy`:

```
cp_set_schedule_branching(KALIS_INITIAL_SOLUTION, Strategy)
if cp_schedule(diskuse)=0 then writeln("Problem infeasible"); end-if
```

With this search strategy, the optimal solution is found in the 'initial solution' phase after just 8 nodes and the enumeration stops there since the pretreatment phase has proven a lower bound of 3 which is just the value of the optimal solution.

NB: to obtain an output log from the different phases of the scheduling search set the control parameter `KALIS_VERBOSE_LEVEL` to 2, that is, add the following line to your model before the start of the solution algorithm.

```
setparam("KALIS_VERBOSE_LEVEL", 2)
```

5.7.2 Task-based enumeration

A task-based enumeration strategy consists in the definition of a selection strategy choosing the task to be enumerated, a value selection heuristic for the task durations, and a value selection heuristic for the task start times.

Consider the typical definition of a job-shop scheduling problem: we are given a set of jobs that each need to be processed in a fixed order by a given set of machines. A machine executes one job at a time. The durations of the production tasks (= processing of a job on a machine) and the sequence of machines per job for a 6×6 instance are shown in Table 5.3. The objective is to minimize the makespan (latest completion time) of the schedule.

Table 5.3: 6×6 job-shop instance from [**?**]

Job	Machines						Durations					
1	3	1	2	4	6	5	1	3	6	7	3	6
2	2	3	5	6	1	4	8	5	10	10	10	4
3	3	4	6	1	2	5	5	4	8	9	1	7
4	2	1	3	4	5	6	5	5	5	3	8	9
5	3	2	5	6	1	4	9	3	5	4	3	1
6	2	4	6	1	5	3	3	3	9	10	4	1

5.7.2.1 Model formulation

Let JOBS denote the set of jobs and MACH ($\mathrm{MACH} = \{1, ..., \mathrm{NM}\}$) the set of machines. Every job j is produced as a sequence of tasks $\mathrm{task}_{\mathrm{jm}}$ where $\mathrm{task}_{\mathrm{jm}}$ needs to be finished before $\mathrm{task}_{\mathrm{j,m+1}}$ can start. A task $\mathrm{task}_{\mathrm{jm}}$ is processed by the machine $\mathrm{RES}_{\mathrm{jm}}$ and has a fixed duration $\mathrm{DUR}_{\mathrm{jm}}$.

The following model formulates the job-shop scheduling problem.

resources $res_m(m \in MACH = \{1, ..., NM\})$
tasks $task_{jm}(j \in JOBS, m \in MACH)$
minimize $finish$
$finish = maximum_{j \in JOBS}(task_{j,NM}.end)$
$\forall m \in MACH : res_m.capacity = 1$
$\forall j \in JOBS, m \in MACH : task_{jm}.start, task_{jm}.end \in \{0, ..., MAXTIME\}$
$\forall j \in JOBS, m \in \{1, ..., NM - 1\} : task_{jm}.successors = \{task_{j,m+1}\}$
$\forall j \in JOBS, m \in MACH : task_{jm}.duration = DUR_{jm}$
$\forall j \in JOBS, m \in MACH : task_{jm}.requirement_{RES_{jm}} = 1$

5.7.2.2 Implementation

The following Mosel model implements the job-shop scheduling problem and defines a task-based branching strategy for solving it. We select the task that has the smallest remaining domain for its start variable and enumerate the possible values for this variable starting with the smallest. A task-based branching strategy is defined with the *kalis* function `task_serialize`, that takes as arguments the user task selection, value selection strategies for the duration and start variables, and the set of tasks it applies to. Such task-based branching strategies can be combined freely with any variable-based branching strategies.

```
model "Job shop (CP)"
 uses "kalis", "mmsystem"

 parameters
  DATAFILE = "jobshop.dat"
  NJ = 6                                ! Number of jobs
  NM = 6                                ! Number of resources
 end-parameters

 forward function select_task(tlist: cptasklist): integer

 declarations
  JOBS = 1..NJ                          ! Set of jobs
  MACH = 1..NM                          ! Set of resources
  RES: array(JOBS,MACH) of integer      ! Resource use of tasks
  DUR: array(JOBS,MACH) of integer      ! Durations of tasks

  res: array(MACH) of cpresource        ! Resources
  task: array(JOBS,MACH) of cptask      ! Tasks
 end-declarations

 initializations from "Data/"+DATAFILE
  RES DUR
 end-initializations

 HORIZON:= sum(j in JOBS, m in MACH) DUR(j,m)
 forall(j in JOBS) getend(task(j,NM)) <= HORIZON

! Setting up the resources (capacity 1)
 forall(m in MACH)
  set_resource_attributes(res(m), KALIS_UNARY_RESOURCE, 1)

! Setting up the tasks (durations, resource used)
 forall(j in JOBS, m in MACH)
  set_task_attributes(task(j,m), DUR(j,m), res(RES(j,m)))

! Precedence constraints between the tasks of every job
 forall (j in JOBS, m in 1..NM-1)
!  getstart(task(j,m)) + DUR(j,m) <= getstart(task(j,m+1))
```

```
  setsuccessors(task(j,m), {task(j,m+1)})

! Branching strategy
 Strategy:=task_serialize(->select_task, KALIS_MIN_TO_MAX,
            KALIS_MIN_TO_MAX,
            union(j in JOBS, m in MACH | exists(task(j,m))) {task(j,m)})
 cp_set_branching(Strategy)

! Solve the problem
 starttime:= gettime

 if not cp_minimize(getmakespan) then
  writeln("Problem is infeasible")
  exit(1)
 end-if

! Solution printing
 cp_show_stats
 write(gettime-starttime, "sec ")
 writeln("Total completion time: ", getsol(getmakespan))
 forall(j in JOBS) do
  write("Job ", strfmt(j,-2))
  forall(m in MACH | exists(task(j,m)))
   write(formattext("%3d:%3d-%2d", RES(j,m), getsol(getstart(task(j,m))),
         getsol(getend(task(j,m)))))
  writeln
 end-do

!*********************************************************************
! Task selection for branching
 function select_task(tlist: cptasklist): integer
  declarations
   Tset: set of integer
  end-declarations

  ! Get the number of elements of "tlist"
  listsize:= getsize(tlist)

  ! Set of uninstantiated tasks
  forall(i in 1..listsize)
   if not is_fixed(getstart(gettask(tlist,i))) then
    Tset+= {i}
   end-if

  returned:= 0

  ! Get a task with smallest start time domain
  smin:= min(j in Tset) getsize(getstart(gettask(tlist,j)))
  forall(j in Tset)
   if getsize(getstart(gettask(tlist,j))) = smin then
    returned:=j; break
   end-if

 end-function

end-model
```

5.7.2.3 Results

An optimal solution to this problem has a makespan of 55. In comparison with the default *scheduling* strategy, our branching strategy reduces the number of nodes that are enumerated from over 300 to just 105 nodes with a comparable model execution time (however, for larger instances the default scheduling strategy is likely to outperform our branching strategy). The default *minimization* strategy does not find any solution for this problem within several minutes running time.

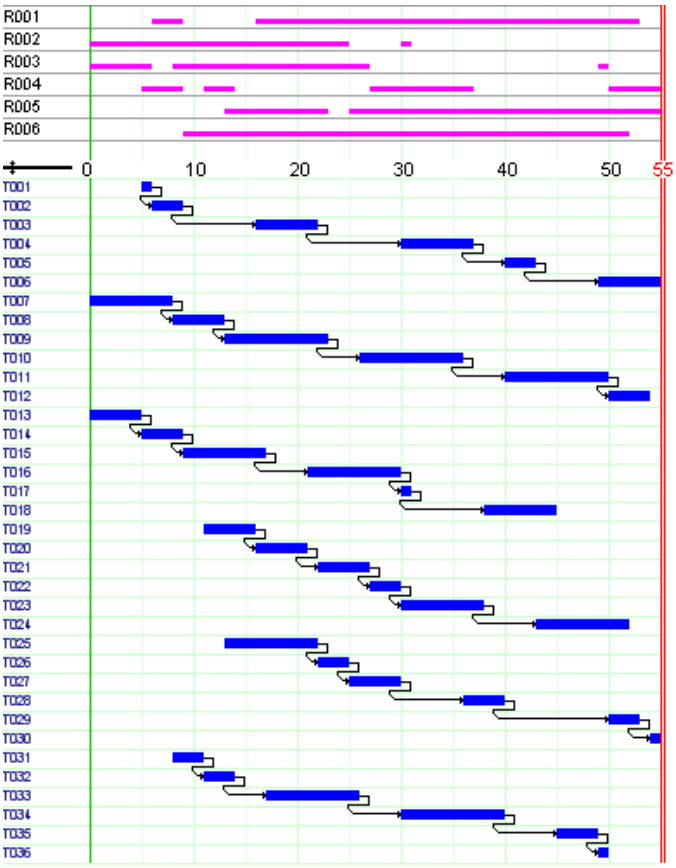

Figure 5.8: Solution graph

5.7.2.4 Alternative search strategies

Similarly to what we have seen above, we may define a user task selection strategy for the scheduling search. The only modifications required in our model are to replace `cp_set_branching` by `cp_set_schedule_strategy` and `cp_minimize` by `cp_schedule`. The definition of the user task choice function `select_task` remains unchanged.

```
! Branching strategy
 Strategy:=task_serialize(->select_task, KALIS_MIN_TO_MAX,
            KALIS_MIN_TO_MAX,
            union(j in JOBS, m in MACH | exists(task(j,m))) {task(j,m)})
 cp_set_schedule_strategy(KALIS_INITIAL_SOLUTION, Strategy)

! Solve the problem
 if cp_schedule(getmakespan)=0 then
  writeln("Problem is infeasible")
  exit(1)
 end-if
```

This strategy takes even fewer nodes for completing the enumeration than the standard search with user task selection.

Instead of the user-defined task selection function `select_task` it is equally possible to use one of the predefined task selection criteria:

`KALIS_SMALLEST_EST / KALIS_LARGEST_EST` choose task with the smallest/largest lower

bound on its start time (*'earliest start time'*),

`KALIS_SMALLEST_LST / KALIS_LARGEST_LST` choose task with the smallest/largest upper bound on its start time (*'latest start time'*),

`KALIS_SMALLEST_ECT / KALIS_LARGEST_ECT` choose task with the smallest/largest lower bound on its completion time (*'earliest completion time'*),

`KALIS_SMALLEST_LCT / KALIS_LARGEST_LCT` choose task with the smallest/largest upper bound on its completion time (*'latest completion time'*).

For the present example, the best choice proves to be `KALIS_SMALLEST_LCT` (terminating the search after approximately 60 nodes with both `cp_schedule` and `cp_minimize`):

```
Strategy:=task_serialize(KALIS_SMALLEST_LCT, KALIS_MIN_TO_MAX,
           KALIS_MIN_TO_MAX,
           union(j in JOBS, m in MACH | exists(task(j,m))) {task(j,m)})
```

Even more specialized task selection strategies can be defined with the help of `group_serialize`. In this case you may decide freely which are the variables to enumerate for a task and in which order they are to be considered. The scope of `group_serialize` is not limited to tasks, you may use this subroutine to define search strategies involving other types of (user) objects.

5.7.3 Choice of the propagation algorithm

The performance of search algorithms for scheduling problems relies not alone on the definition of the enumeration strategy; the choice of the propagation algorithm for resource constraints may also have a noticeable impact. The propagation type is set by an optional last argument of the procedure `set_resources_attributes`, such as

```
set_resource_attributes(res, KALIS_DISCRETE_RESOURCE, CAP, ALG)
```

where `ALG` is the propagation algorithm choice.

For *unary resources* we have a choice between the algorithms `KALIS_TASK_INTERVALS` and `KALIS_DISJUNCTIONS`. The former achieves stronger pruning at the cost of a larger computational overhead making the choice of `KALIS_DISJUNCTIONS` more competitive for small to medium sized problems. Taking the example of the jobshop problem from the previous section, when using `KALIS_DISJUNCTIONS` the default scheduling search is about 4 times faster than with `KALIS_TASK_INTERVALS` although the number of nodes explored is slightly larger than with the latter.

The propagation algorithm options for *cumulative resources* are `KALIS_TASK_INTERVALS` and `KALIS_TIMETABLING`. The filtering algorithm `KALIS_TASK_INTERVALS` is stronger and relatively slow making it the preferred choice for hard, medium-sized problems whereas `KALIS_TIMETABLING` should be given preference for very large problems where the computational overhead of the former may be prohibitive. In terms of an example, the binpacking problem we have worked with in Sections 5.4 and 5.7.1 solves about three times faster with `KALIS_TASK_INTERVALS` than with `KALIS_TIMETABLING` (using the default scheduling search).

CHAPTER 6

Hybridization of CP and MP

This chapter introduces the concept of LP/MIP relaxations for problems formulated as CP models with Xpress Kalis. By means of examples we show how to

- use automated linear programming relaxations,
- work with linear programming relaxations,
- configure CP search to be guided by information from LP/MIP relaxations.

Note: The functionality described in this chapter requires Xpress Optimizer to be installed and licensed in addition to Xpress Kalis.

6.1 Linear relaxations

Constraint Programming and Mathematical Programming (here used for Linear and Mixed Integer Programming, LP/MIP) are two different approaches to solving optimization problems that have a number of complementary features. Some problems, or parts of problems, are better solved by CP, others by LP/MIP techniques. Based on this observation it is possible to devise various approaches for hybrid CP-MP problem solving. The FICO Xpress Optimization whitepaper *'Hybrid MIP/CP solving with Xpress Optimizer and Xpress Kalis'* describes several decomposition schemes where coordination between MIP and CP models is done on the Mosel level. Xpress Kalis now provides a built-in linear programming relaxation functionality where communication and cooperation take place directly on the solver level, with optional configuration and user interaction from the Mosel CP model.

In the context of Xpress Kalis we use the term *linear relaxation* for sets of linear (in)equality constraints that are solved by an external LP/MIP solver. In the case of an automatic relaxation this will be a reformulation of (a subset of) the constraints in the CP problem. A user-defined 'linear relaxation', however, may also include additional information, in which case this name choice might be somewhat misleading.

Xpress Kalis can build automatically several linear (or mixed integer linear) relaxations of a CP problem and use Xpress Optimizer to solve them. Xpress Kalis uses a double modeling approach (as in [**?**]) by exchanging automatically and in a bidirectional way, information such as objective bounds, infeasibility, optimal relaxed solutions and reduced costs between the CP solver and the linear relaxations solver(s) during the search for a solution.

1. Linear relaxations can be generated fully automatically.
2. Alternatively, relaxations can be configured by selecting sets of constraints to form subproblems and by defining their solving frequency.
3. Relaxation problems can also be defined freely by the user; they may even include constraints that do not occur in the original CP problem.

These three ways of defining linear relaxations are presented in the following sections.

6.2 Automatic relaxation

When developing a CP model we recommend that you always try the effect of using the automatic relaxation. Indeed, in many of the examples presented in the previous chapters (such as examples `a4sugar_ka.mos` from Section 3.6 or `j5tax_ka.mos` from Section 3.9) we can simplify the model formulation by removing the definition of the search strategy and enabling the automated relaxation by adding the following line before the start of the optimization:

```
setparam("kalis_auto_relax",true)
```

whereby in many cases you will observe that the time to the first or optimal solution is shortened, or that optimality is proven faster than with CP-only search.

6.2.1 Integer knapsack problem with side constraint

The following integer knapsack problem with an additional 'all_different' constraint on its variables will serve us in this and the following sections to show how to work with linear relaxations:

$$
\begin{aligned}
\text{maximize} \quad & 5 \cdot x_1 + 8 \cdot x_2 + 4 \cdot x_3 + x_4 \\
\text{s.t.} \quad & x_j \in \{1, 3, 8, 9, 12\} \text{ for } j = 1, 2, 3, 4 \\
& 3 \cdot x_1 + 5 \cdot x_2 + 2 \cdot x_3 \leqslant 50 \\
& 2 \cdot x_1 + x_3 + 5 \cdot x_4 \leqslant 75 \\
& \text{all-different}(x_1, x_2, x_3, x_4)
\end{aligned}
$$

6.2.2 Implementation

The model implementation shown below is fairly straightforward, with the exception of the setting of the control parameter `KALIS_AUTO_RELAX`.

```
model "Knapsack with side constraints"
 uses "kalis"

 declarations
  VALUES = {1,3,8,9,12}
  R = 1..4
  x: array(R) of cpvar              ! Decision variables
  benefit: cpvar                    ! The objective to minimize
 end-declarations

! Enable output printing
 setparam("kalis_verbose_level", 1)

! Setting name of variables for pretty printing
 forall(i in R) setname(x(i),"x"+i)
 setname(benefit,"benefit")

! Set initial domains for variables
 forall(i in R) setdomain(x(i), VALUES)

! Knapsack constraints
 3*x(1) + 5*x(2) + 2*x(3) <= 50
 2*x(1) + x(3) + 5*x(4) <= 75

! Additional global constraint
 all_different(union(i in R) {x(i)})

! Objective function
 benefit = 5*x(1) + 8*x(2) + 4*x(3) + x(4)
```

```
! Initial propagation
 if not cp_propagate: exit(1)

! Display bounds on objective after constraint propagation
 writeln("Constraint propagation objective ", benefit)


! **** Linear relaxation ****

! Enable automatic linear relaxations
 setparam("kalis_auto_relax", true)

! Solve the problem
 if cp_maximize(benefit):
  cp_show_sol                      ! Output optimal solution to screen

end-model
```

6.2.3 Results

Enabling the automatic relaxation for this problem reduces the number of nodes spent by the search from 17 to 4. However, more time is spent in every node due to the overhead incurred by the LP solving. As solving times in this small problem are negligible in all formulations time measurements are not really meaningful; in larger problems the effect of using linear relaxations needs to be tested carefully.

6.3 Configuring automatic relaxations

This section and the remainder of the chapter are addressed at more expert readers who, besides experience with CP, also have some knowledge of LP/MIP modeling and solving techniques.

6.3.1 Configuration choices

The formulation of a linear relaxation (sub)problem with Xpress Kalis has several components, namely the problem definition (=statement of variables and constraints), the configuration of a linear relaxation solver, and its insertion into the search strategy of the original CP problem. Here is a summary of the different steps and configuration choices.

1. **Choosing the constraint type**
   The following constraints can be relaxed automatically with Xpress Kalis:

   - linear constraints
   - all-different
   - occurrence
   - distribute
   - min/max
   - absolute value
   - distance
   - element
   - cycle
   - logical (implies, or, and, equiv)

   To obtain an automatic relaxation of the whole problem, Xpress Kalis forms the intersection of the relaxations of the constraints of the CP model. This relaxation can be retrieved by a call to the

function `cp_get_linrelax`. An optional argument to this function lets you specify the (set of) constraint types to relax. The relaxation for a specific constraint—as opposed to constraint type—is obtained with `get_linrelax`.

2. **Choosing the relaxation type**
   Xpress Kalis generally provides several distinct relaxations for a given constraint. Methods `cp_get_linrelax` and `get_linrelax` take an integer parameter to choose the type of the relaxation (0 for an LP oriented relaxation and 1 for a MIP oriented relaxation). Besides the automatic relaxations, the user can also freely define his own relaxations of these or other constraints of his CP models (see Section 6.4 below).

3. **Defining relaxation solvers**
   To solve a linear relaxation, Xpress Kalis uses *relaxation solvers* (model objects of the type `cplinrelaxsolver`) that define for the relaxation:

   (a) an objective variable,

   (b) an optimization sense (either minimization or maximization),

   (c) a configuration of when and what to do with the relaxation,

   (d) a flag indicating whether to solve the relaxation as a pure LP problem or as a MIP.

   The following predefined configurations are implemented within *kalis*:

   `KALIS_TOPNODE_RELAX_SOLVER` This configuration solves the relaxation only at the top node and provides bound for the objective variable.

   `KALIS_TREENODE_RELAX_SOLVER` This configuration solves the relaxation at each node of the search tree and provides bounds for the objective variable and performs reduced costs propagation.

   `KALIS_BILEVEL_RELAX_SOLVER` This configuration solves the relaxation whenever all the variables (by default all the discrete variables such as `cpvar`) of a user defined set are instantiated. After the resolution of the relaxation, all the other variables are instantiated to the optimal value of the relaxation. The principal use of the bilevel configuration is to decompose simply and automatically the CP model into a main problem and a subproblem.

   In addition to the predefined schemes, the user can take full control of the relaxation solver by means of callbacks (see the Xpress Kalis Mosel Reference Manual for details).

   Note: a model can contain several linear relaxations solvers, *e.g.*, for solving different subproblems or with different configuration options (such as one solver for the top node and a different relaxation solver for the tree nodes of the CP search).

4. **Branching with relaxation information**
   Certain predefined branching schemes and value and variable selection heuristics in *kalis* are based on the optimal solution of a relaxation. The information obtained from the relaxations can be used to explore quickly the most promising portions of the search space. For example, a MIP style branching scheme is defined by

```
cp_set_branching(assign_and_forbid(KALIS_LARGEST_REDUCED_COST(mysolver),
                                   KALIS_NEAREST_RELAXED_VALUE(mysolver)))
```

   where the `KALIS_LARGEST_REDUCED_COST` variable selection heuristic selects the variable with the largest reduced cost in the optimal solution of the relaxation in the specified relaxation solver, and the `KALIS_NEAREST_RELAXED_VALUE` value selection heuristic selects the value in the domain of the variable that is the nearest (using L1-Norm) to the value of this variable in the optimal solution of the relaxation in the relaxation solver passed in the argument.

6.3.2 Implementation

The model shown below implements a similar relaxation as the automatic version in the previous section. This model has two parts: in the beginning we define a standard CP model for our Knapsack problem and trigger constraint propagation to display the initial bounds on the objective function. The second part shows how to define and configure a linear relaxation solver. At first, the model generates the automatic relaxation (and displays it on screen). The argument value `0` in `cp_get_linrelax` indicates that we want an LP-oriented relaxation (use `1` for a MIP-oriented relaxation). The relaxation solver defined with `get_linrelax_solver` is configured to maximize the objective `benefit`, solving the problem as a MIP, and this only once, at the first node of the CP search. The linear relaxation solver is added to the search process by a call to `cp_add_linrelax_solver` (there may be several solvers in a search process). The model definition is completed by a 'MIP style' branching scheme that branches first on the variables with largest reduced cost, and tests first the values nearest to the optimal solution of the relaxation.

```
model "Knapsack with side constraints"
 uses "kalis", "mmsystem"

 declarations
  VALUES = {1,3,8,9,12}
  R = 1..4
  x: array(R) of cpvar              ! Decision variables
  benefit: cpvar                    ! The objective to minimize
 end-declarations

! Enable output printing
 setparam("kalis_verbose_level", 1)

! Setting name of variables for pretty printing
 forall(i in R) setname(x(i),"x"+i)
 setname(benefit,"benefit")

! Set initial domains for variables
 forall(i in R) setdomain(x(i), VALUES)

! Knapsack constraints
 3*x(1) + 5*x(2) + 2*x(3) <= 50
 2*x(1) + x(3) + 5*x(4) <= 75

! Additional global constraint
 all_different(union(i in R) {x(i)})

! Objective function
 benefit = 5*x(1) + 8*x(2) + 4*x(3) + x(4)

! Initial propagation
 if not cp_propagate: exit(1)

! Display bounds on objective after constraint propagation
 writeln("Constraint propagation objective ", benefit)


! **** Linear relaxation ****

 declarations
  myrelaxall: cplinrelax
 end-declarations

! Build an automatic 'LP' oriented linear relaxation
 myrelaxall:= cp_get_linrelax(0)

! Output the relaxation to the screen
 cp_show_relax(myrelaxall)

 mysolver:= get_linrelax_solver(myrelaxall, benefit, KALIS_MAXIMIZE,
```

```
                KALIS_SOLVE_AS_LP, KALIS_TREENODE_RELAX_SOLVER)

! Define the linear relaxation
 cp_add_linrelax_solver(mysolver)

! Define a 'MIP' style branching scheme using the solution of the relaxation
 cp_set_branching(assign_var(KALIS_LARGEST_REDUCED_COST(mysolver),
                             KALIS_NEAREST_RELAXED_VALUE(mysolver)))

! Solve the problem
 starttime:= gettime
 if cp_maximize(benefit) then
  write(gettime-starttime, "sec. ")
  cp_show_sol                      ! Output optimal solution to screen
 end-if

end-model
```

6.3.3 Results

The formulation of a CP model for this problem is straightforward. However, solving this type of problems with pure CP search methods tends to be very time consuming due to the poor quality of the bounds on the objective variable derived through constraint propagation. Formulating a MIP model for this problem is a perhaps somewhat more complicated task due to the need to linearize the 'all_different' relation. Here the automatic reformulation provided by Xpress Kalis is certainly helpful. The LP relaxation provides valuable bounds and insight on how to direct the search. Combining both makes the CP search go straight to the optimal solution and optimality is proven immediatly by the relaxation bounds. The configuration options allow you to tune the behavior of the combined solution algorithm.

Try to experiment with other settings:

- LP or MIP style relaxations (first parameter of `cp_get_linrelax`),
- relax only certain constraints, *e.g.*, by specifying `KALIS_LINEAR_CONSTRAINTS` in `cp_get_linrelax`:

```
cp_get_linrelax(0, {KALIS_LINEAR_CONSTRAINTS})
```

- solving frequency (tree nodes or top node): last argument of `get_linrelax_solver`,
- subproblem type (LP or MIP): 4^{th} argument of `get_linrelax_solver`.

Note: Instead of displaying the linear relaxation on screen you may also choose to export it to a file in LP format using procedure `export_prob`, specifying a linear relaxation solver and a matrix filename (the extension ".lp" will be appended to the name) as arguments.

```
export_prob(mysolver, "mymatfile")
```

6.4 User-defined relaxations

As an alternative or in addition to the built-in relaxation definitions, linear relaxations (or parts thereof) can be built up from scratch, similarly to the definition of linear constraints in a pure CP model. For the representation of linear relaxations, Xpress Kalis provides the types `cpauxvar` and `cpauxlinexp` (for variables and linear expressions that do not occur in the original CP model). The linearization of certain global constraints requires a mapping from integer-valued CP decision variables to sets of 0-1 indicator variables in the LP/MIP formulation, where each indicator variable is associated with a value in the domain of the CP variable. These indicator variables (of type `cpauxvar`) are obtained with function `get_indicator`. Besides these special variable types, linear relaxation constraints can also contain variables of type `cpvar` or `cpfloatvar`.

6.4.1 Implementation

The implementations below replace the configured definition of a relaxation by a relaxation built up 'manually' by stating the constraints one by one. Just like the previous model version, a relaxation solver is defined with `get_linrelax_solver`. This linear relaxation solver is then added to the search process by a call to `cp_add_linrelax_solver` and the model definition is completed by a 'MIP style' branching scheme that branches first on the variables with largest reduced cost, and tests first the values nearest to the optimal solution of the relaxation.

This model extract selects one by one the constraints to be added to the relaxation:

```
declarations
 myrelax: cplinrelax
end-declarations

! Build an 'LP' oriented linear relaxation
myrelax:= get_linrelax(all_different(union(i in R) {x(i)}), 0)
myrelax += get_linrelax(3*x(1) + 5*x(2) + 2*x(3) <= 50, 0)
myrelax += get_linrelax(2*x(1) + x(3) + 5*x(4) <= 75, 0)
myrelax += get_linrelax(benefit - (5*x(1) + 8*x(2) + 4*x(3) + x(4)) = 0, 0)

mysolver:= get_linrelax_solver(myrelax, benefit, KALIS_MAXIMIZE,
               KALIS_SOLVE_AS_LP, KALIS_TREENODE_RELAX_SOLVER)

! Output the relaxation to the screen (after creation of solver to see all!)
cp_show_relax(myrelax)

! Define the linear relaxation
cp_add_linrelax_solver(mysolver)

! Define a 'MIP' style branching scheme using the solution of the relaxation
cp_set_branching(assign_var(KALIS_LARGEST_REDUCED_COST(mysolver),
                            KALIS_NEAREST_RELAXED_VALUE(mysolver)))

! Solve the problem
starttime:= gettime
if cp_maximize(benefit) then
 write(gettime-starttime, "sec. ")
 cp_show_sol                      ! Output optimal solution to screen
end-if
```

When adding linear constraints to the relaxation we have the choice to add them directly, or by using the function `get_linrelax` as in the model extract above. When using this function we can specify through its second argument whether to disregard the integrality condition on any variables the linear constraint.

If we do not want to use any of the built-in relaxation functionality, we can also state the relaxations 'manually'. In the example below the linear relaxation is formulated entirely with the binary indicators for domain values, leaving out the variables x_i that are present in all the automated relaxation versions we have seen so far.

```
declarations
 myrelax: cplinrelax
end-declarations

! Build a user-defined linear relaxation

! Formulation of 'alldifferent':
forall(val in VALUES)
 myrelax += sum(i in R | contains(x(i),val)) get_indicator(x(i), val) <= 1
forall(i in R)
 myrelax += sum(val in VALUES | contains(x(i),val)) get_indicator(x(i), val) = 1

! Reformulation of linear constraints
myrelax +=
```

```
  3*sum(val in VALUES | contains(x(1),val)) val*get_indicator(x(1), val) +
  5*sum(val in VALUES | contains(x(2),val)) val*get_indicator(x(2), val) +
  2*sum(val in VALUES | contains(x(3),val)) val*get_indicator(x(3), val) <= 50
myrelax +=
   2*sum(val in VALUES | contains(x(1),val)) val*get_indicator(x(1), val) +
   sum(val in VALUES | contains(x(3),val)) val*get_indicator(x(3), val) +
   5*sum(val in VALUES | contains(x(4),val)) val*get_indicator(x(4), val) <= 75
myrelax += benefit -
  (5*sum(val in VALUES | contains(x(1),val)) val*get_indicator(x(1), val) +
   8*sum(val in VALUES | contains(x(2),val)) val*get_indicator(x(2), val) +
   4*sum(val in VALUES | contains(x(3),val)) val*get_indicator(x(3), val) +
   sum(val in VALUES | contains(x(4),val)) val*get_indicator(x(4), val)) = 0

mysolver:= get_linrelax_solver(myrelax, benefit, KALIS_MAXIMIZE,
               KALIS_SOLVE_AS_LP, KALIS_TREENODE_RELAX_SOLVER)
```

6.4.2 Results

When comparing the program output obtained from the two formulations above, it appears that the automatic formulation provides better guidance to the CP search (that is, less nodes are required to terminate the enumeration) than the 'manual' formulation that works exclusively with the binary indicator variables. There often are different possibilities for formulating linear relaxations and some experimentation is required to determine which is the best choice.

General recommendations for the formulation of linear relaxations are (a) to be careful to define a relaxation that makes sense and (b) to make sure that the relaxation is connected to the original problem (through variables shared by both formulations) so to obtain the desired benefits from information exchange between the two views of the problem.

Note: *kalis* also makes accessible additional configuration options for the underlying LP/MIP solver; the interested reader is referred to the documentation of `set_linrelax_solver_attribute` in the Xpress Kalis Mosel Reference Manual.

Appendix

Appendix A

Trouble shooting

- **No license found:** to work with Xpress Kalis for Mosel, the Xpress licensing system, and the *kalis* module must be installed. You need to copy the license file that you will receive from your software vendor into the Xpress installation directory and set the environment variable `XPAUTH_PATH` to point to this directory.
- **The Xpress Kalis module is not found:** if the file `kalis.dso` is not installed in the directory `dso` of the Mosel distribution, then the environment variable `MOSEL_DSO` must be defined with the location of this file.

Appendix B

Glossary of CP terms

Some terms commonly used in CP might require some explanation for readers with an Operations Research background. The following list is an extract from [**?**].

Finite domain constraint problem/constraint satisfaction problem (CSP): defined by a finite set of variables taking values from *finite domains* and a (conjunctive) set of constraints on these variables. The objective may be either finding one solution (any or an optimal) or all solutions (consistent assignment of values to the variables so that all the constraints are satisfied simultaneously) for the given instance. The term *constraint network* is frequently employed to denote CP problems in allusion to the graphical representation as a hyper graph (constraint graph), where nodes represent variables, and constraints are (hyper) arcs linking several nodes. There is no standard problem representation in CP.

Model: a CP model specifies all variables, their domains and their declarative meaning and *conceptual constraints* imposed on them (as opposed to *actual constraints* that are used to implement the properties of the solution and the search process). In CP in general, a model preserves much problem-specific knowledge about variables and the relations between them. This allows the development and application of more efficient specialized solution strategies.

Variable: object that has a name and a domain (also referred to as *decision variable*).

Domain: the set of values (also: labels) a variable may take. In Xpress Kalis, it may consist of discrete values, or intervals of integers. When solving CP problems active use of the domain concept is made. At any stage, the domain of a variable is the set of values that cannot be proved to be inconsistent (with the constraints on this variable) using the available consistency checking methods. Assigning or restricting domains is often interpreted as unary constraints on the corresponding variables.

Instantiation of a set of variables is an assignment of a value to each variable from its domain, also called *labeling* of each variable with a value from its domain.

Consistent instantiation of a constraint network is an instantiation of the variables such that the constraints between variables are satisfied, also called *admissible/satisfied instantiation, consistent assignment of values*, or *consistent labeling*. *Solution* is often used as a synonym for consistent instantiation, but may also denote the result after applying any (local/partial) consistency algorithm.

Constraint: a relation over a set of variables limiting the combination of values that these variables can take; constraints may also be interpreted as mappings from the domains of the variables onto the Boolean values *true* and *false*. A (conceptual) constraint can sometimes be implemented in different ways enforcing various levels of consistency (see below) with different computational overhead. So-called *global constraints* subsume a set of other constraints (for instance an 'all-different' relation on a set of variables replaces pair wise disequality constraints). Global constraints use specific propagation/consistency algorithms that render them more efficient than the set of constraints they replace.

Redundant constraints: a constraint is redundant with respect to a set of constraints, if it is satisfied when the set of constraints is satisfied. Although redundant constraints do not change the set of solutions (consistent instantiations) of a problem, in practice it may be useful to add redundant constraints to the model formulation because they can help CP solution procedures, particularly by

achieving more powerful constraint propagation.

System of constraints: a conjunctive set of constraints, usually built up *incrementally*.

Constraint solving: deciding the consistency or satisfiability of a system of constraints.

Solution methods: finite domain CP problems are usually solved by tree search methods (Branch-and-Bound for optimization, Branch-and-Prune for decision problems) that enumerate the possible values of the variables coupled with consistency algorithms. In tree search methods with consistency checking the local consistency algorithm is triggered by the propagation of the domain changes of the branching variable. For *optimization* usually a cost constraint is introduced that propagates to the variables. It is updated (in the case of minimization: bounded to be smaller than the solution value) each time a new solution is found.

Consistency techniques and constraint propagation: Consistency algorithms remove inconsistent values from the domains of variables. Informally speaking, a consistency algorithm is 'stronger' than another one if it reduces the domains further, *i.e.*, it establishes a higher level of consistency. In finite domain CP, typically local consistency algorithms are used. *Local* or *partial consistency* signifies that only subsets of the constraints of a system of constraints are simultaneously satisfied. A locally consistent (according to some notion of consistency, such as arc-consistency) constraint network can be obtained by propagating iteratively the effects of each constraint to all other constraints it is connected to through its variables until a stable state is reached. This process is referred to as *constraint propagation*. Propagation properties of constraints vary, *e.g.*, due to their implementation, or the types of variables used. Possible events triggering their evaluation may be variable instantiation, modification of domain bounds, removing of value(s) from a domain, *etc.*

Backtrack search augmented by constraint propagation:
while *not solved* **and** *not infeasible*
  check/establish (local) consistency
  if a dead end is detected
    then backtrack to the first open node
  else
    select a variable
    select a value for the variable

Search algorithms/strategies: The values for variables come out of an enumeration process. 'Intelligent' enumeration strategies adapted to special types of constraints and variables are a central issue in CP. The search is controlled by problem specific heuristics, strategies from Mathematical Programming or the expert's knowledge; fixing variables to trial values is possible. One can distinguish variable and value selection heuristics. Due to the way the backtracking mechanism works, usually depth-first search is used.

Constraint solver: (Also: *constraint engine*.) Distinction between exact and incomplete solvers. *Exact* solvers guarantee the satisfiability of the system of constraints at any stage of the computations, they usually work on rational numbers (trees of rationals and linear constraints). *Incomplete* solvers are designed for more complex domains such as integers where checking and maintaining consistency of the overall system is too expensive or not possible with presently known algorithms. These solvers work with simplified calculations establishing some sort of partial (local) consistency among constraints; usually simply stating constraints does not produce a solution, an enumeration phase (searching for solutions) is necessary.

Index