Applications of optimization with Xpress-MP

Revised translation from the French language edition of:
Programmation linéaire
by Christelle Guéret, Christian Prins, Marc Sevaux

Translated and revised by Susanne Heipcke

Published by Dash Optimization Ltd.
www.dashoptimization.com

Published by:

Dash Optimization Ltd.
Blisworth House
Blisworth
Northants
NN7 3BX
United Kingdom

Revised translation first published 2002

Translation and editing of this book has been supported by the EC Framework 5 project LISCOS (contract G1RD-1999-00034).

Cover design: James Atkins Design, www.jades.co.uk
Manufacturing coordinator: Software Logistics, www.softwarelogistics.com

ISBN: 0-9543503-0-8

Last update 10 September, 2007

About the authors

Christelle Guéret is an Associate Professor in the Automatic Control and Production Systems department at the 'Ecole des Mines de Nantes' (France). She received her Ph.D. degree in 1997 at the 'Université de Technologie de Compiègne' and defended her Habilitation in December 2004 at the University of Nantes in the IRCCyN laboratory. Her teaching courses include graph algorithms, scheduling and linear programming. Her main research areas concern optimization problems in production and logistics, and more specifically scheduling and vehicle routing problems.

Susanne Heipcke worked for BASF-AG (Germany) before joining Dash Optimization in 1998. Her Ph.D. research (awarded in 1999 by the University of Buckingham) focused on the solution of large-scale industrial problems by a combination of constraint programming and mixed integer programming. More recently she has worked on various aspects of modeling, including the development of teaching material for Mosel, and interfaces to different types of solvers and solution methods. Since 2001 she has participated in teaching the course on mathematical modeling in the OR master program at the University Aix-Marseille II.

Christian Prins received his Ph.D. and Habilitation in computer science from Pierre and Marie Curie University (Paris VI). He is now a Professor in the Industrial Systems Engineering Department at the University of Technology of Troyes (UTT, France). His research interest include combinatorial optimization, vehicle routing and transportation, scheduling and algorithmic tools for optimization (*e.g.* metaheuristics and reusable software components). He is a member of the editorial board of *Computers and Operations Research*.

Marc Sevaux is a Professor at the University of South-Britanny in Lorient (France). He received his Ph.D. degree in 1998 at the Pierre and Marie Curie University (Paris VI) and defended his Habilitation in July 2004 at the University of Valenciennes in the LAMIH laboratory. He is primarily interested in solving combinatorial optimization problems, particularly planning, scheduling and routing problems. Marc Sevaux is the coordinator of the EUropean chapter on MEtaheuristics (EU/ME). He is an associate editor of *Journal of Heuristics* and *INFOR*.

Contents

Foreword

In the past, many students have become acquainted with optimization through a course on Linear Programming. Until very recently computers were too expensive and optimization software only available to industrial groups and research centers with powerful machines. Courses on Linear Programming therefore were usually restricted to solving problems in two variables graphically and applying the famous simplex algorithm in tableau form to tiny problems. The size of the problems thus solved by hand rarely exceeded 6 to 8 variables. The practice of modeling was commonly neglected due to a lack of time and the unavailability of software to verify the models.

Nowadays Linear Programming is still frequently taught from a very academic perspective. Of course, in the majority of course programs Linear Programming appears together with other classics of Operations Research such as the PERT method for project planning. But with the exception of a few students in applied mathematics who have learned to deal with applications, the others do not retain much: they have no precise idea of the possible applications, they do not know that now there are readily available software tools and most importantly, they have not been taught how to model.

The idea of this book was born based on this observation. With PCs becoming ever more powerful and the rapid progress of the numerical methods for optimization the market of Linear/Mixed Integer Programming software is changing. Contrary to the first products (subroutine libraries reserved for specialist use), the more recent generations of software do not require any programming. These tools are easy to use thanks to their user-friendly interface and a more and more natural syntax that is close to the mathematical formulae.

In different study programs (sciences as well as business/economics) a significant increase in the students' interest and 'productivity' has been observed when such tools are used. With such an approach, even students without any mathematical background are well able to model and solve real problems of considerable size, whilst they would be lost in a purely theory-oriented course about optimization methods. These students experience the satisfaction of mastering powerful tools and learn to appreciate the field of Linear/Mixed Integer Programming. Later, at work, they may recall the potential of these tools and based on the experience they have gained, may use them where it is necessary. What else should one aim for to spread the use of Mathematical Programming?

This book has been written for students of science and business/economics and also for decision makers, professionals, and technical personnel who would like to refresh their knowledge of Linear/Mixed Integer Programming and more importantly, to apply it in their activities.

To focus on modeling and on the application examples, the theoretical aspects of Linear/Mixed Integer Programming are only briefly mentioned in this book, without giving any detailed explanations. Ample references are provided and commented on to point the interested reader to where to find more information about the different topics.

To avoid a pre-digested textbook approach, the examples chosen for this book describe real situations. Although simplified compared to the real world so as to remain accessible for beginners, most examples are sufficiently complicated not to be solvable by hand on a piece of paper and they all require a non-trivial modeling phase.

The software used for the implementation and problem solving, Xpress-MP, may be downloaded from

http://www.dashoptimization.com/applications_book.html

free of charge in a version limited in size (but sufficient to solve the problems in this book). So the reader does not have to worry about how to solve the problems and is free to concentrate his efforts on the most noble task that nevertheless requires progressive and methodical training: the modeling. The opportunity to validate a model immediately using a software tool is gratifying: results are obtained immediately, modeling errors are discovered faster, and it is possible to perform various simulation runs by interactively modifying the problem parameters.

It is not the aim of this book to turn the reader into an Xpress-MP specialist at the expense of the model-

ing work. Every problem is therefore first modeled in mathematical form without making any reference to the implementation with the modeling software. The translation of the model into a program is given afterwards, accompanied by an explanation of any specific language features used, and followed by the presentation and discussion of the results. The mathematical model and the results are obviously independent of the software. All Xpress-MP model and data files are available from the Dash Optimization website and can be used to verify the results immediately. It would of course be possible to implement the models in a different way.

To avoid establishing a heterogeneous catalogue of applications, the collection has been structured stressing the large variety of application areas of Linear/Mixed Integer Programming. The introductory chapters about modeling and the use of Xpress-MP are followed by ten chapters with applications. Every chapter is dedicated to a different application area and may be read independently of the rest. It always starts with a general description of the topic, followed by a choice of (on the average) six problems. A summary section at the end of every chapter provides additional material and references.

Chapter 6 describes blending problems or the separation of components that are usually encountered in mining and process industries. These problems only rarely use integer variables and are the simplest and best suited for beginners. The two following chapters are dedicated to two other important groups of industrial applications, namely scheduling (Chapter 7) and production planning problems (Chapter 8).

In many applications, it is required to fill a limited space (boxes, holds of ships, computer disks) with objects or, on the contrary, cut material to extract patterns whilst minimizing the trim loss. This type of problem is the subject of Chapter 9 about loading and cutting.

Flow problems in the widest sense give rise to a large number of interesting optimization problems. This problem type is discussed across three representative application areas: ground transport (Chapter 10), air transport (Chapter 11), and the burgeoning world of telecommunications (Chapter 12). Some problems may be shifted between these three chapters, but other applications are more specific, like the tours driven by a vehicle in ground transport, the problems of flight connections in air transport, and all that concerns dimensioning and reliability of networks in telecommunications.

Since this book is not aimed exclusively at scientists, industrialists, and logisticians, more recent and less well known application areas of Mathematical Programming have also been included. Chapter 13 is dedicated to problems in economics and finance. The subjects of Chapter 14 are timetabling and personnel planning problems. Mathematical Programming may equally be of use to local authorities, administrations, and public services in general: Chapter 15 describes six examples.

The classification of the problems has been a subject of discussion: theoreticians usually prefer a classification based on the theoretical model type. For instance the so-called flow models are well known classics that concern several problems in this book (water conveyance, ground transport, telecommunications). Preference has been given to a classification by application area to allow a professional or a student of a given subject to identify himself with at least one entire chapter. Nevertheless, the other classification has not been omitted: the additions at the end of every chapter indicate the applications of the same theoretical type in other chapters of the book. An overview table at the beginning of the examples part also lists all models with their theoretical types.

The bibliography entries are all grouped at the end of the book because certain references are cited in several chapters. However, the section 'References and further material' at the end of every chapter comments on the references that are useful to learn more about a certain topic. The references have been chosen carefully to provide a blend of basic articles and works, sometimes already quite old, and very recent articles that illustrate the rapid progress of the discipline.

The Xpress-MP software may be seen as a product of the progress achieved in optimization in recent years. The first software tools in the 60s and 70s required the user to enter the problem in numerical form *via* matrices that are of little intuition, difficult to re-read and to modify. Starting in the 80s, algebraic modeling languages have been made available that enable users to write linear programs in a form close to the algebraic notation. An algebraic modeling language allows the user to write generic models using indexed variables and data arrays. This means that it is possible to separate the model from the data (stored in separate files) and to make it independent of the size of the problem. Such a high level language focuses the user's attention on the modeling, reduces the possible sources of errors, accelerates model development and facilitates the understanding and modifications to the model at a later time.

The modeling and solving environment Xpress-Mosel [CH02] used in this book belongs to a new generation of optimization tools: it provides all the functionality of an algebraic modeling language and it also gives access to the problem solving. Through its programming facilities it becomes possible to implement solution heuristics and data pre- and post-treatment in the same environment as the model itself: all application examples (to a different extent) use this facility to formulate and solve problems and display the results. An interesting feature for more advanced uses is Mosel's modular design through

which, for instance, access to other solver or solution algorithm types or different data sources becomes possible. This feature is not used in this book, since we employ a single solver, the Xpress-Optimizer for Linear/Mixed Integer and Quadratic Programming.

For working with Mosel models under Microsoft Windows the graphical user interface Xpress-IVE may be used: it provides debugging support, makes accessible additional information about the problem and the solution process, and allows the user to display his results graphically. As a default, on all supported operating systems, Mosel works with a command line interpreter based on standard text files.

Only a few other books have been published with an aim similar to the present one: a book by Williams provides twenty case studies, in the third edition [Wil93] accompanied by implementations with Xpress-MP. The case studies are well suited for student project work, requiring several days of work. Unfortunately, this book only covers relatively few application areas. Another example is the book by Schrage [Sch97] about the software Lindo. This work contains more applications, but classified by theoretical models. The problems are implemented directly with the software, without using a modeler.

Compared to the previously cited works, the present book contains a larger number of applications (sixty) structured by application areas. Although all models are implemented with Xpress-MP, every problem is first modeled with a mathematical syntax which makes it possible to use the models with different implementations. Finally, many models are re-usable for related problems or large-size instances due to a generic way of modeling (maximum flow, minimum cost flow, assignment, transport, traveling salesman problem *etc.*).

Updates to this book will be made available at

http://www.dashoptimization.com/applications_book.html

and you are encouraged to look there regularly.

The partial funding of the work on this book by the EC Framework 5 project LISCOS (contract G1RD-1999-00034) is gratefully acknowledged.

We would specially like to thank Yves Colombani for conceiving, designing and implementing Mosel. His vision has made Mosel the powerful and open software that it is today.

Preliminaries

What you need to know before reading this book

Before reading this book you should be comfortable with the use of symbols such as x or y to represent unknown quantities, and the use of this sort of variable in simple linear equations and inequalities, for example:

$$x + y \leq 6$$

Experience of a basic course in Mathematical or Linear Programming is worthwhile, but is not essential. Similarly some familiarity with the use of computers would be helpful.

For all but the simplest models you should also be familiar with the idea of summing over a range of variables. For example, if $produce_j$ is used to represent the number of cars produced on production line j then the total number of cars produced on all N production lines can be written as:

$$\sum_{j=1}^{N} produce_j$$

This says 'sum the output from each production line $produce_j$ over all production lines j from $j = 1$ to $j = N$'.

If our target is to produce at least 1000 cars in total then we would write the inequality:

$$\sum_{j=1}^{N} produce_j \geq 1000$$

We often also use a set notation for the sums. Assuming that *LINES* is the set of production lines $\{1, .., N\}$, we may write equivalently:

$$\sum_{j \in LINES} produce_j \geq 1000$$

This may be read 'sum the output from each production line $produce_j$ over all production lines j in the set *LINES*'.

Other common mathematical symbols that are used in the text are $\mathbb{N}$ (the set of non-negative integer numbers $\{0, 1, 2, \ldots\}$), $\cap$ and $\cup$ (intersection and union of sets), $\wedge$ and $\vee$ (logical 'and' and 'or'), the all-quantifier $\forall$ (read 'for all'), and $\exists$ (read 'exists').

Computer based modeling languages, and in particular the language we use, Mosel, closely mimic the mathematical notation an analyst uses to describe a problem. So provided you are happy using the above mathematical notation the step to using a modeling language will be straightforward.

Symbols and conventions

We have used the following conventions within this book:

- Mathematical objects are presented in *italics*.
- Examples of commands, models and their output are printed in a `Courier font`. Filenames are given in lower case Courier.
- Decision variables have lower case names; in the application examples these usually are verbs (such as *buy*, *make*).

- Constraint names start with an upper case letter, followed by mostly lower case (*e.g. Profit*, *TotalCost*).
- Data (arrays and sets) and constants are written entirely with upper case (*e.g. DEM*, *JOBS*, *PRICE*).
- The vertical bar symbol | is found on many keyboards as a vertical line with a small gap in the middle, but often confusingly displays on-screen without the small gap. In the UNIX world it is referred to as the pipe symbol. (Note that this symbol is not the same as the character sometimes used to draw boxes on a PC screen.) In ASCII, the | symbol is 7C in hexadecimal, 124 in decimal.

I. Developing Linear and Integer Programming models

Chapter 1

What is modeling? Why use models?

Rather than starting with a theoretical overview of what modeling is, and why it is useful, we shall look at a problem facing a very small manufacturer, and how we might go about solving the problem. When we have gained some practical modeling experience, we shall come back to see what the benefits of modeling have been.

Section 1.1 gives a word description of the problem and its translation into a mathematical form. We then define in Section 1.2 the notion of a **Linear Program** and show how the example problem fits this form. In Section 1.3 the problem is implemented with Mosel and solved with the Xpress-Optimizer. The discussion of the results comprises a graphical representation of the solution information that may be obtained from the solver. Important issues in modeling and solving linear problems are infeasibility and unboundedness (Section 1.4). The chapter closes with reflections on the benefits of modeling and optimization (Section 1.5) and the importance of the data (Section 1.6).

1.1 The chess set problem: description

A small joinery makes two different sizes of boxwood chess sets. The small set requires 3 hours of machining on a lathe, and the large set requires 2 hours. There are four lathes with skilled operators who each work a 40 hour week, so we have 160 lathe-hours per week. The small chess set requires 1 kg of boxwood, and the large set requires 3 kg. Unfortunately, boxwood is scarce and only 200 kg per week can be obtained.

When sold, each of the large chess sets yields a profit of $20, and one of the small chess set has a profit of $5.

The problem is to decide how many sets of each kind should be made each week so as to maximize profit.

1.1.1 A first formulation

Within limits, the joinery can **vary** the number of large and small chess sets produced: there are thus two **decision variables** in our model, one decision variable per product. What we want to do is to find the **best** (*i.e.* optimal) values of these decision variables, where by 'best' we mean that we get the largest profit. We shall give these variables abbreviated names:

xs : the number of small chess sets to make

xl : the number of large chess sets to make

The number of large and small chess sets we should produce to achieve the maximum contribution to profit is determined by the optimization process. In other words, we look to the optimizer to tell us the best values of *xs*, and *xl*.

The values which *xs* and *xl* can take will always be **constrained** by some physical or technological limits. One of the main tasks in building a model is to write down in a formal manner the exact constraints that define how the system can behave. In our case we note that the joinery has a maximum of 160 hours of machine time available per week. Three hours are needed to produce each small chess set, and two hours are needed to produce each large set. So if in the week we are planning to make *xs* small chess sets and

xl large chess sets, then in total the number of hours of machine time we are planning to use is:

$$3 \cdot xs + 2 \cdot xl$$

where the $3 \cdot xs$ comes from the time making small sets, and the $2 \cdot xl$ from the time machining large sets.

Note that we have already made some assumptions here. Firstly we have assumed that the lathe-hours to machine *xs* small sets is exactly *xs* times the lathe-hours required to machine one small set. This probably will not be exactly true in practice — one tends to get faster at doing something the more one does it, so it will probably take a slightly smaller amount of time to machine the 2^{nd} and subsequent sets than the first set. But it is unlikely that this will be a very important effect in our small joinery.

The second assumption we have made is much more likely to be inaccurate. We have assumed that the time for making small **and** large sets is the sum of the times for the sets. We have not allowed for any changeover time: resetting the lathes, cleaning up, getting different size tools *etc.* In some situations, the time that we lose in changeovers can be very large compared with the time we actually spend doing constructive work and then we have to resort to more complex modeling. But for the moment, we shall assume that the changeover times are negligible.

Our first **constraint** is:

$$3 \cdot xs + 2 \cdot xl \leq 160 \quad \text{(lathe-hours)}$$

which says 'the amount of time we are planning to use must be less than or equal to the amount of time available', or equivalently 'we cannot plan to use more of the resource (time) than we have available'. The allowable combinations of small and large chess sets are restricted to those that do not exceed the lathe-hours available.

In addition, only 200 kg of boxwood is available each week. Since small sets use 1 kg for every set made, against 3 kg needed to make a large set, a second constraint is:

$$1 \cdot xs + 3 \cdot xl \leq 200 \quad \text{(kg of boxwood)}$$

where the left hand side of the inequality is the amount of boxwood we are planning to use and the right hand side is the amount available.

The joinery cannot produce a negative number of chess sets, so two further **non-negativity constraints** are:

$$xs \geq 0$$
$$xl \geq 0$$

In a similar way, we can write down an expression for the total profit. Recall that for each of the large chess sets we make and sell we get a profit of \$20, and one of the small chess set gives us a profit of \$5. Assuming that we can sell all the chess sets we make (and note that this may not always be a reasonable assumption) the total profit is the sum of the individual profits from making and selling the *xs* small sets and the *xl* large sets, *i.e.*

$$Profit = 5 \cdot xs + 20 \cdot xl$$

Profit is the **objective function**, a linear function which is to be optimized, that is, maximized. In this case it involves all of the decision variables but sometimes it involves just a subset of the decision variables. Note that *Profit* may be looked at as a **dependent variable**, since it is a function of the decision variables. In maximization problems the objective function usually represents profit, turnover, output, sales, market share, employment levels or other 'good things'. In minimization problems the objective function describes things like total costs, disruption to services due to breakdowns, or other less desirable process outcomes.

Consider some possible values for *xs*, and *xl* (see Table 1.1). The aim of the joinery is to maximize profit, but we cannot select any combination of *xs* and *xl* that uses more of any of the resources than we have available. If we do plan to use more of a resource than is available, we say that the plan **violates** the constraint, and the plan is **infeasible** if one or more constraints is violated. If no constraints are violated, the plan is **feasible**. The column labeled 'OK?' in the table tells us if the plan is feasible. Plans C and E are infeasible.

In terms of profit, plan H looks good. But is it the best plan? Is there a plan that we have not considered that gives us profit greater than 1320? To answer this question we must move to the notion of **optimization**.

Table 1.1: Values for *xs* and *xl*

	xs	*xl*	Lathe-hours	Boxwood	OK?	*Profit*	Notes
A	0	0	0	0	Yes	0	Unprofitable!
B	10	10	50	40	Yes	250	We won't get rich doing this.
C	-10	10	-10	20	No	150	Planning to make a negative number of small sets.
D	53	0	159	53	Yes	265	Uses all the lathe-hours. There is spare boxwood.
E	50	20	190	110	No	650	Uses too many lathe-hours.
F	25	30	135	115	Yes	725	There are spare lathe-hours and spare boxwood.
G	12	62	160	198	Yes	1300	Uses all the resources
H	0	66	130	198	Yes	1320	Looks good. There are spare resources.

1.2 Linear Programming

We have just built a model for the decision process that the joinery owner has to make. We have isolated the decisions he has to make (how many of each type of chess set to manufacture), and taken his objective of maximizing profit. The constraints acting on the decision variables have been analyzed. We have given names to his variables and then written down the constraints and the objective function in terms of these variable names.

At the same time as doing this we have made, explicitly or implicitly, various assumptions. The explicit assumptions that we noted were:

- For each size of chess set, manufacturing time was proportional to the number of sets made.
- There was no down-time because of changeovers between sizes of sets.
- We could sell all the chess sets we made.

But we made many implicit assumptions too. For instance, we assumed that no lathe will ever break or get jammed; that all the lathe operators will turn up for work every day; that we never find any flaws in the boxwood that lead to some being unusable or a chess set being unacceptable; that we never have to discount the sale price (and hence the per unit profit) to get an order. And so on. We have even avoided a discussion of what is the worth of a fraction of a chess set — is it a meaningless concept, or can we just carry the fraction that we have made over into next week's production?

All mathematical models necessarily contain some degree of simplification of the real world that we are attempting to describe. Some assumptions and simplifications seem eminently reasonable (for instance, that we can get the total profit by summing the contributions of the individual profits from the two sizes); others may in some circumstances be very hopeful (no changeover time lost when we swap between sizes); whilst others may just be cavalier (all the lathe operators will arrive for work the day after the World Cup finals).

Modeling is an art, not a precise science. Different modelers will make different assumptions, and come up with different models of more or less precision, and certainly of different sizes, having different numbers of decision variables. And at the same time as doing the modeling, the modeler has to be thinking about whether he will be able to *solve* the resulting model, that is find the maximum or minimum value of the objective function and the values to be given to the decision variables to achieve that value.

It turns out that many models can be cast in the form of **Linear Programming** models, and it is fortunate that Linear Programming (LP) models of very large size can be solved in reasonable time on relatively inexpensive computers. It is not the purpose of this book to discuss the algorithms that are used to solve LP problems in any depth, but it is safe to assume that problems with tens of thousands of variables and constraints can be solved with ease. So if you can produce a model of your real-world situation, without too many wild assumptions, in the form of an LP then you know you can get a solution.

So we next need to see what a Linear Programming problem consists of. To do so, we first introduce the notion of a **linear expression**. A linear expression is a sum of the following form

$$A_1 \cdot x_1 + A_2 \cdot x_2 + A_3 \cdot x_3 + \ldots + A_N \cdot x_N$$

which in mathematical notation is usually written as

$$\sum_{j=1}^{N} A_j \cdot x_j$$

where $A_1, \ldots, A_N$ are constants and $x_1, \ldots, x_N$ are decision variables. So for instance, if we have variables x, $make_P$ and $make_Q$

$$2 \cdot x - 3 \cdot make_P + 4 \cdot make_Q$$

is a linear expression, but

$$2 \cdot x \cdot make_P - 3 \cdot make_P + 4 \cdot make_Q$$

is not, as the first term contains the product of two variables.

Next, we introduce the notion of **linear inequalities** and **linear equations**. For any linear expression $\sum_{j=1}^{N} A_j \cdot x_j$ and any constant B, the inequalities

$$\sum_{j=1}^{N} A_j \cdot x_j \leq B \quad \text{and} \quad \sum_{j=1}^{N} A_j \cdot x_j \geq B$$

are **linear inequalities**, and the equation

$$\sum_{j=1}^{N} A_j \cdot x_j = B$$

is a **linear equation**. So for example our lathe-hours constraint

$$3 \cdot xs + 2 \cdot xl \leq 160$$

is a linear inequality, but

$$2 \cdot xs \cdot xl + 3 \cdot xs = 200$$

is not a linear equation because of the first term, which is a product of two decision variables.

Now, if we have decision variables

$$x_1, x_2, x_3, \ldots, x_N,$$

a linear expression

$$C_1 \cdot x_1 + C_2 \cdot x_2 + C_3 \cdot x_3 + \ldots + C_N \cdot x_N$$

and a number of linear inequalities and linear equations

$$\begin{aligned}
&A_{i1} \cdot x_1 + A_{i2} \cdot x_2 + A_{i3} \cdot x_3 + \ldots + A_{iN} \cdot x_N \leq B_i \quad \text{for } i = 1, \ldots, M_1 \\
&A_{i1} \cdot x_1 + A_{i2} \cdot x_2 + A_{i3} \cdot x_3 + \ldots + A_{iN} \cdot x_N = B_i \quad \text{for } i = M_1 + 1, \ldots, M_2 \\
&A_{i1} \cdot x_1 + A_{i2} \cdot x_2 + A_{i3} \cdot x_3 + \ldots + A_{iN} \cdot x_N \geq B_i \quad \text{for } i = M_2 + 1, \ldots, M_3,
\end{aligned}$$

then a **Linear Programming problem** is to

$$\begin{aligned}
\text{maximize or minimize} \quad & \sum_{i=j}^{N} C_j \cdot x_j \quad \text{(the \textbf{objective function})} \\
\text{subject to the constraints} \quad & \sum_{j=1}^{N} A_{ij} \cdot x_j \leq B_i \quad \text{for } i = 1, \ldots, M_1 \\
& \sum_{j=1}^{N} A_{ij} \cdot x_j = B_i \quad \text{for } i = M_1 + 1, \ldots, M_2 \\
& \sum_{j=1}^{N} A_{ij} \cdot x_j \geq B_i \quad \text{for } i = M_2 + 1, \ldots, M_3 \\
\text{and} \quad & x_j \geq 0 \quad \text{for each } j = 1, \ldots, N \quad \text{(\textbf{non-negativity constraints})}
\end{aligned}$$

The B_i are often called the **right hand sides** (RHS).

So, for instance, the chess set model is a linear program as it has variables xs and xl and is to

$$\begin{aligned}
\text{maximize} \quad & 5 \cdot xs + 20 \cdot xl \\
\text{subject to} \quad & 1 \cdot xs + 3 \cdot xl \leq 200 \quad \text{(kg of boxwood)} \\
& 3 \cdot xs + 2 \cdot xl \leq 160 \quad \text{(lathe-hours)} \\
& xs \geq 0 \\
& xl \geq 0
\end{aligned}$$

We have stressed the linearity condition, where the objective function and all of the constraints must be linear in the decision variables. But there are two further properties that we must have, Divisibility and Determinism.

Divisibility means that in an acceptable solution any values of the decision variables are allowed within the restrictions imposed by the linear constraints. In particular, we are not constrained to accept only whole number (integer) values for some or all of the decision variables. We have already alluded to this, when we remarked on the debate as to whether a fraction of a chess set is worth something (or, more precisely, whether a fraction *f* of a chess set is worth exactly *f* times the worth of a whole chess set).

The final requirement we have of an LP problem is that it is **deterministic** — all the coefficients in the constraints and the objective function are known exactly. Determinism is sometimes a very strong assumption, particularly if we are building planning models which extend some way into the future. Is it reasonable, for example, to assume that it will always take 7.5 days for the oil tanker to reach our refinery from the Gulf? Or that all of the lathe operators will arrive for work every day next week?

Problems where we **must** consider the variability in objective function coefficients, right hand sides or coefficients in the constraints are **Stochastic Programming** problems. We do not consider them in this book, because they are difficult to deal with, not because they are of little practical interest. In fact, it can be argued that **all** planning models have stochastic elements, and we will later demonstrate some methods for dealing with uncertainty in an LP framework.

1.3 Solving the chess set problem

1.3.1 Building the model

The Chess Set problem can be solved easily using Mosel. We will discus Mosel much more in the next chapter, and then throughout the book, but give the Mosel model here so you can see the solution to the problem we have modeled.

The first stage is to get the model we have just developed into Mosel. Here it is.

Table 1.2: Chess Set Model

```
                                                                    Notes
model Chess                                                           1
 uses "mmxprs"                  ! We use the Xpress-MP Optimizer      2
 declarations
  xs, xl: mpvar                 ! These are the decision variables    3
 end-declarations

 Profit:=  5*xs + 20*xl                                               4
 Boxwood:= 1*xs + 3*xl <=  200   ! kg of boxwood
 Lathe:=   3*xs + 2*xl <=  160   ! Lathehours

 maximize(Profit)                                                     5

 writeln("LP Solution:")                                              6
 writeln(" Objective: ", getobjval)
 writeln("Make ", getsol(xs), " small sets")
 writeln("Make ", getsol(xl), " large sets")
end-model                                                             7
```

When we were building our mathematical model we used the notation that items in italics (for example, *xs*) were the mathematical variables. The corresponding Mosel decision variables have the same name in non-italic courier font (for example, `xs`).

Once the model has been defined, the objective function can be optimized and the optimal number of chess sets to make is obtained.

Notice that the character '`*`' is used to denote multiplication of a decision variable by the constant associated with it in the constraint. Blanks are not significant and the modeling language distinguishes between upper and lower case, so `X1` would be recognized as different from `x1`.

By default, Mosel assumes that all variables are constrained to be non-negative unless it is informed otherwise. There is therefore no need to specify non-negativity constraints on variables. It is possible to tell Mosel that variables are not non-negative: we shall see how to do this later.

Here are some notes on the perhaps non-obvious parts of the model. They follow the numbering in the model listing 1.2. All of the points will be elaborated later, so do not be overly concerned if you do not understand some point now.

1 and 7. We give the model a name. The `end-model` statement terminates the model.
2. We tell Mosel we shall be using the Xpress-MP Optimizer to maximize the problem.
3. We declare the 'make' variables to be decision variables, Mosel's `mpvar` variables.
4. The objective function and constraints are given here.
5. Here we tell Mosel to maximize the objective function `Profit`. 6. This part writes out the solution values. `getobjval` returns the optimal value of the objective function, while `getsol(x)` returns the optimal value of `x`.

1.3.2 The results

We shall see later how to run the Mosel model. Here is the output.

```
LP Solution:
Objective: 1333.33
Make 0 small sets
Make 66.6667 large sets
```

Considering the original model:

$$\begin{array}{ll} \text{maximize} & 5 \cdot xs + 20 \cdot xl \\ \text{subject to} & 1 \cdot xs + 3 \cdot xl \leq 200 \quad \text{(kg of boxwood)} \\ & 3 \cdot xs + 2 \cdot xl \leq 160 \quad \text{(lathe-hours)} \\ & xs \geq 0 \\ & xl \geq 0 \end{array}$$

we can check the answer. Mosel tells us that xl is the only non-zero variable, with value 66.6667. The profit is indeed $20 \cdot 66.6667 = 1333.33$. We can easily see that the amount of boxwood we use is $3 \cdot 66.6667 = 200$, exactly what we have available; and the lathe-hours we use are $2 \cdot 66.6667 = 133.3333$, 66.6667 fewer than are available; and we satisfy the non-negativity constraints. Note that we do not use all the lathe-hours, but we do use all the boxwood.

We may represent the solution to this problem graphically as in Figure 1.1. The grey shaded area is the **feasible region**. The values of the objective function *Profit* are marked with dotted lines.

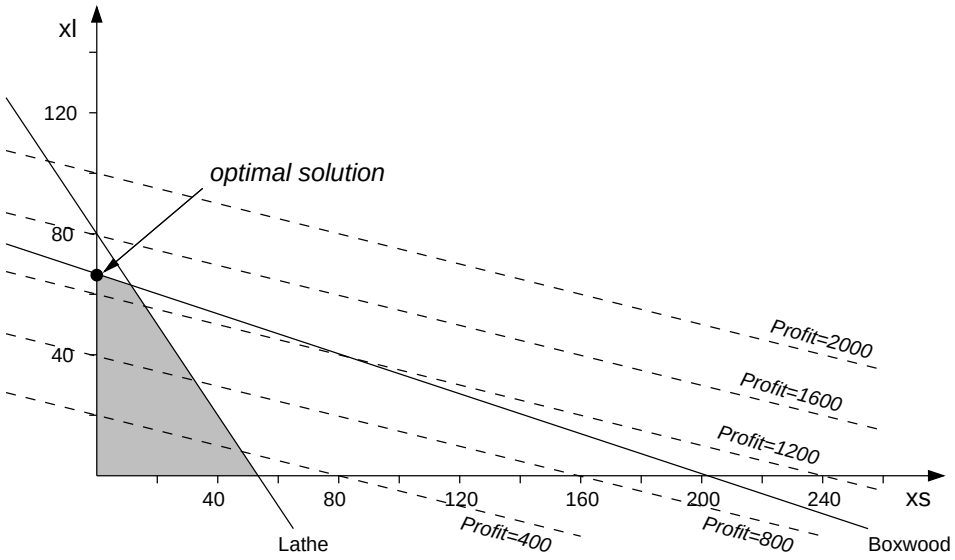

Figure 1.1: Solution of the Chess Set problem

If we want to analyze the solution to this problem further, we may wish to obtain the constraint activities and their dual values (shadow prices), and also the reduced costs of the decision variables. We might add some more writing as follows to the Mosel model:

```
 writeln("Activities/Dual Values")
 writeln(" Lathe: ", getact(Lathe)," / ", getdual(Lathe))
 writeln(" Boxwood: ", getact(Boxwood)," / ", getdual(Boxwood))

 writeln("Reduced Costs")
 writeln(" xs: ", getrcost(xs))
 writeln(" xl: ", getrcost(xl))
```

The output then becomes

```
LP Solution:
Objective: 1333.33
 Make 0 small sets
 Make 66.6667 large sets
Activities/Dual Values
 Lathe: 133.333 / 0
 Boxwood: 200 / 6.66667
Reduced Costs
 xs: -1.66667
 xl: 0
```

The **activity** of a constraint is the evaluation of its left hand side (that is, the sum of all terms involving decision variables). The optimal solution only uses 133.333 lathe-hours, but all 200 kg of boxwood that are available. As the 160 available lathe-hours come from 4 machines that operate 40 hours per week, we could reduce the weekly working time to 35 hours (resulting in a total of $4 \cdot 35 = 140$ hours) without changing anything in the solution (Figure 1.2).

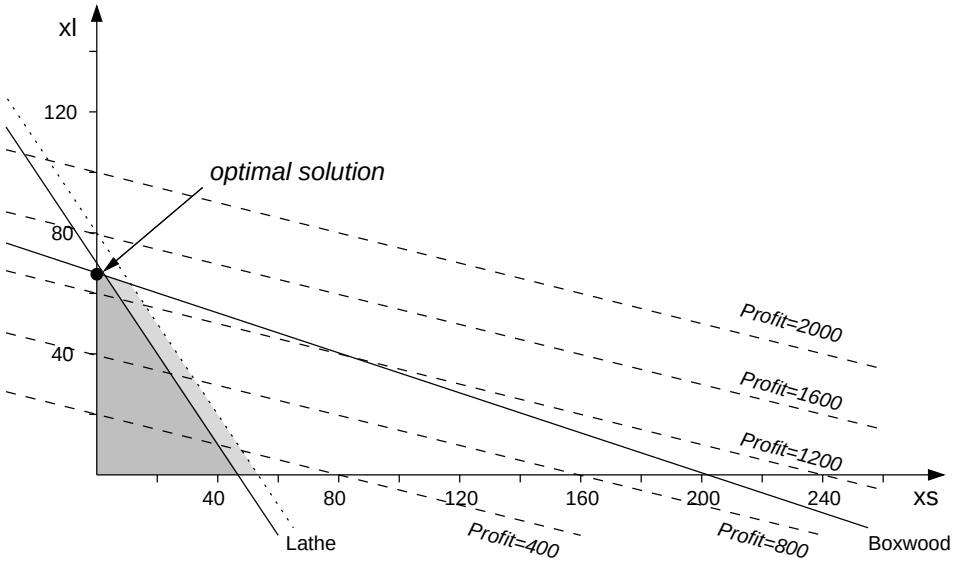

Figure 1.2: Reduction of lathe-hours

If we modify the limit on the availability of boxwood, this will have an immediate effect on the solution: the **dual value** (also called **shadow price**) of 6.66667 for the boxwood constraint tells us that one additional unit (*i.e.* 1 kg) of boxwood will lead to an increase of the objective function by 6.66667, giving a total profit of $1340. Graphically, this may be represented as follows (Figure 1.3).

The **reduced cost** values of the variables indicate the cost of producing one (additional) unit of every type of chess set: if we decided to produce one small chess set this would cost us 1.66667 more (in terms of resource usage) than what its sale contributes to the total profit. Or put otherwise, we need to increase the sales price of *xs* by 1.66667 to make small chess sets as profitable as large ones. Figure 1.4 shows the effect on the solution when we increase the price of *xs* to 6.66667 (its **break-even point**): any point on the highlighted edge of the feasible region now leads to the same objective value of $1333.33, so we might choose to produce a small number of small chess sets with the same total profit.

1.3.3 Divisibility again

In the present example we are faced with the Divisibility problem mentioned earlier. In reality accepting fractional answers may be practical since a chess set that is not completed in one week's production sched-

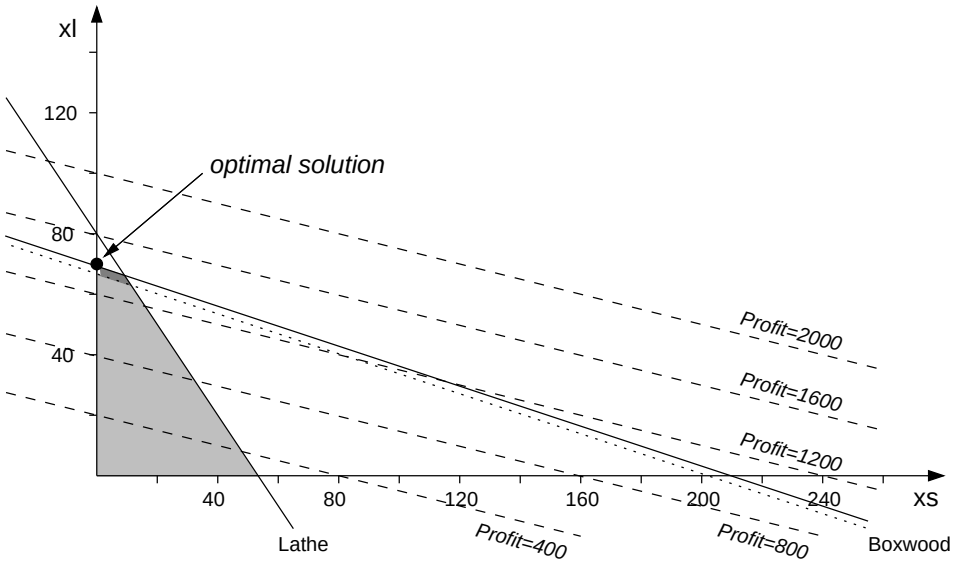

Figure 1.3: Additional unit of wood

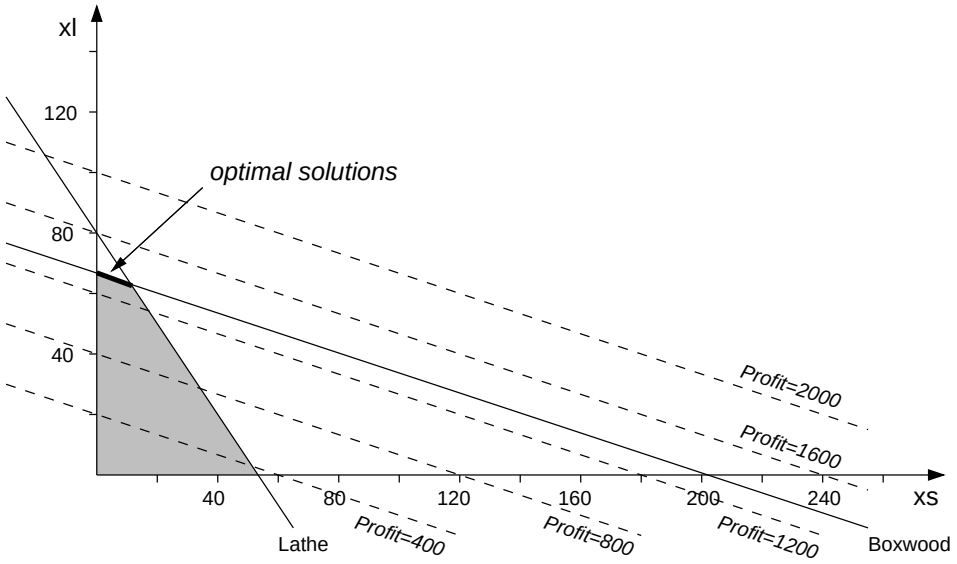

Figure 1.4: Increasing the price of *xs* to its break-even point

ule may be completed the following week. However, some optimization problems have variables that cannot be fractional in this way. Such problems require the techniques of **Mixed Integer Programming**, which we cover later (Chapter 3).

1.3.4 Unboundedness and infeasibility

Although we have found an optimal solution to the problem given, this may not always be possible. If the constraints were insufficient it might be possible to improve the objective function indefinitely. In this case, the problem is said to be **unbounded**. This usually means that a relevant constraint has been omitted from the problem or that the data are in error, rather than indicating an infinite source of profit. Conversely, if the constraints are too restrictive it may not be possible to find any values for the decision variables that satisfy all constraints. In this case, the problem is said to be **infeasible**. This would be the case in the chess set problem if an order of 100 large chess sets had to be made each week.

1.4 Diagnosing infeasibility and unboundedness

One of the most common, and unwelcome, things that happens when one is building LP models is when the solver says that the solution is **unbounded** or **infeasible**, both of which mean that we have done something wrong in our modeling or that the data we are feeding to the model are wrong in some way.

Let us deal with **unboundedness** first. Unboundedness in a maximization problem means that we can increase the profit without limit, thereby ultimately getting infinite profit. Rather than immediately retiring, buying a yacht and going off to the sunshine, we should pause for a moment and consider that infinite profits do not arise in practice — we have done something wrong. Unboundedness usually springs from having forgotten some constraint or having put in an item which has a cost associated with it with a profitable coefficient in the objective function. This mistake should usually be very easy to spot as the LP solver will tell you which decision variable can be increased indefinitely and so theoretically gives an infinite profit. All you have to do is to look at that particular variable and see what in the real world is stopping it from increasing indefinitely. You then have to model the forgotten constraint, and try solving again. In practice it is usually fairly easy to diagnose unboundedness using this technique.

Unfortunately, it is not usually as easy to diagnose **infeasibility**, the situation where there is no set of values for the decision variables that satisfy all the constraints simultaneously. Unlike unboundedness it is perfectly possible for infeasibility to arise in practice in a planning situation where one is making assumptions about resource availabilities, market demands *etc.*

If one is optimizing an existing system, which by observation we know has a feasible solution, then we have an easier job to diagnose the infeasibility. Take the values of the decision variables that we are using right now and put them into the linear programming problem. An easy way to do this is to put the variables in as fixed variables. Then see which of the constraints are being violated. These constraints are prime candidates for having been wrongly modeled in some way. For example, if we know that today we can produce certain numbers of 60 watt and 100 watt light bulbs but our model tells us that we have violated a manufacturing constraint then we have wrongly modeled that constraint and it is in that area of the model where we need to concentrate our efforts.

If we do not have a set of values that we know we can realize then we are faced with a much tougher problem. When seeing an infeasible problem, naive users often say 'which constraint is wrong?' or equivalently 'which constraint is causing the infeasibility?' There is no answer to this question other than to tell the naive user that it is the **combination** of constraints that is causing the infeasibility, or, in other words, that the combination of the constraints is contradictory. Consider the following contrived problem:

$$
\begin{aligned}
make_1 &\leq 2 \\
make_2 &\leq 3 \\
make_3 &\leq 4 \\
make_1 + make_2 + make_3 &\geq 10
\end{aligned}
$$

It is obvious that these constraints are contradictory (add up the first three of them and compare with the last) but without some knowledge of what the model refers to it is impossible to see which one (or even all four) of these constraints is 'wrong'. All we know is that these four constraints taken together are contradictory.

It turns out to be useful in practice to be able to isolate from a large model sets of constraints that together are contradictory but such that if you removed any one of them from the set the remainder

are not contradictory. These are the so called **Irreducible Infeasible Sets** (IIS) which good LP solvers can provide for infeasible problems. It is beyond the scope of this chapter to talk about how to apply IISs and we refer you to the Xpress-MP manuals for a full discussion.

It has to be acknowledged that diagnosing infeasibilities is the hardest job in modeling and even the most experienced analyst sighs when faced with the 'Problem is Infeasible' message from an optimization system. Since infeasibilities can arise from bad modeling or bad data there is no obvious first place to look. Experienced modelers of course know that the data are the most likely things to be in error but even experienced modelers do make mistakes from time to time.

There is one small crumb of comfort — infeasibility has nothing to do with the objective function and is only influenced by the constraints.

Here are a few things to look for when you are facing an infeasible problem:

- Have I got the right sense of a constraint? (For instance, I may have put in a $\geq$ constraint when it should be a $\leq$ one; or an equality when an inequality would do.)
- Have I got the right flow direction for a flow variable (see Section 2.2)? (It is easy to get the sign wrong in flow balance constraints.)
- Have I got a non zero value for a resource availability? (Often missing data shows up us a zero value and if this turns into a zero right hand side for a resource availability we frequently get an infeasibility.)

1.5 The benefits of modeling and optimization

Having just discussed rather technical points it is now time to reflect on modeling and optimization and the benefits they bring to an organization. But first we will discuss how the somewhat idealized processes we have considered so far actually occur in practice.

Building a model, solving it and then implementing the 'answers' is not generally a linear process. As we have hinted above we often make mistakes in our modeling which are usually only detected by the optimization process, where we could get answers that were patently wrong (*e.g.* unbounded or infeasible) or that do not accord with our intuition. If this happens we are forced to reflect further about the model and go into an iterative process of model refinement, re-solution and further analyses of the optimum solution. During this process it is quite likely that we will add extra constraints, perhaps remove constraints that we were mislead into adding, correct erroneous data or even be forced to collect new data that we had previously not considered necessary.

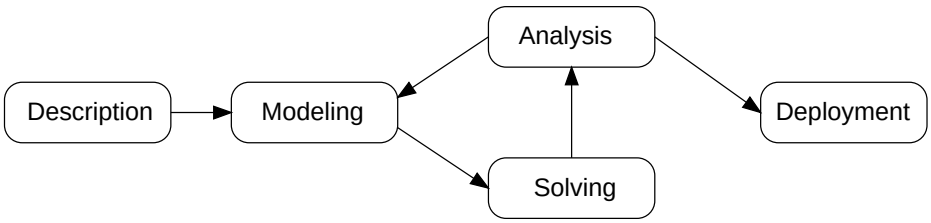

Figure 1.5: Simplified scheme of a typical optimization project

The problems of data accuracy and reconciliation are frequently the most difficult to tackle. By its very nature a model that attempts to capture the influence of constraints coming from different parts of the organization will require the use of data that similarly have come from parts of the organization that may not normally communicate with each other. These data may have been collected in different ways, to different standards and indeed assumptions. The severe process of optimization, which by its nature will seek out extreme solutions if they exist, is very testing of the accuracy of data. Though one will try to use data that already exist in the organization one is sometimes faced with the problem of collecting primary data where existing data are just not adequate.

Though we have posed the discussion above as a set of problems, a more reflective view of the process of modeling and optimization shows that the thoughtful and critical analysis it requires is in fact one of its major benefits. The integrative nature of a model requires that we have data of a consistent standard and consistent accuracy, ideals which any organization should strive for anyway.

Of course, the most obvious benefit of modeling and optimization is that we come up with a **recommended answer** which in some sense that we have defined gives us the best way of running the system

we are modeling. We may choose to reject the solution proffered and indeed there may be very good reasons for so doing. For instance a plan we are suggesting for tomorrow may differ radically in the set of tasks that are suggested but have almost the same objective function value as that we would get from doing exactly the same as we are doing today. In this case it makes perfect sense to forego a small amount of profit for the sake of continuity. The important thing, however, about a quantitative model is that we know how much profit we are foregoing and we can make a **rational and informed decision**, not just one based on hunch or intuition. We can get an idea of the sensitivity of our solution to the inevitable changes and variations in the data, which leads on to the idea of **robust solutions**, solutions which will in some sense remain 'good' as the input data change.

The final major benefit of modeling is that it enables us to **understand the organization** or part of the organization or system that we are studying in a more quantitative and demonstrably rational way, hopefully devoid of organizational politics. This understanding may lead us to consider ways of working or different processes that we have not considered before, which again might lead to considerable **increase in profitability or efficiency**. The boundaries of the model may then increase as we consider an enhanced system or start to consider the system evolving over time.

One thing that is certain about models: as they develop and are worked upon they grow in size. At the advent of the personal computer, a Linear Programming model with 1000 constraints and perhaps 2000 decision variables was considered big, and in fact would probably not be solved on a PC. Now such a model would be considered to be small and models of one hundred to a thousand times that size are regularly solved. Not all of this size increase has come from expansions of the boundary of the typical model, most of it has come from the ability to solve models that have been disaggregated by considering subsets of machinery, or products, time periods, customer zones, labor categories *etc.*

Models will continue to increase in size but it has to be admitted that the benefits of models do not grow in proportion to the model's size – smaller models that capture the essence of the decision problem may give greater insights than bloated models where every factor is disaggregated to the most minute detail.

1.6 Data in models

The primary focus of this book is on the construction of clear, accurate and maintainable models, but we should pause to discuss the data that go into the model.

One of the important characteristics of models is that they often bring together different aspects of the organization in an attempt to achieve an overall optimal solution. A consequence of this is that a model will often require data from many different sources, and there is a need for the accuracy of these data sources to be approximately comparable. There is little point in having some technological data accurate to 6 digits when some of the cost data have been derived from arbitrary accounting principles. In fact one of the major obstacles to successful modeling is the belief in many organizations that accounting data are accurate, whereas frequently much cost accounting data are derived from very questionable assumptions.

Experienced modelers tend to be very distrustful of data in the initial stages of a project. By its very nature, optimization will find extreme solutions, and it is very testing of the accuracy of data. In practice it often turns out that the original data have to be rejected and minor studies initiated with the objective of getting data of better quality.

There is however a danger of over-stressing the need for accuracy in data when we should equally be thinking about the internal consistency of the data. As a perhaps extreme example, it would not affect our optimal policy (*i.e.* the values of the optimal decision variables) if all costs in a model were consistently underestimated by 50%. The value of the objective function of course would be wrong but the actions we propose would be correct. The problem comes when some sources of data are precise whilst others are consistently biased in some way.

It is always best, if possible,

- to use data in their original format, rather than in some aggregated or disaggregated form, and
- to obtain data directly from the owner of the data, that is the person who is responsible for collecting the data and maintaining their accuracy.

Such data will almost invariably be kept in a spreadsheet or a database system and so any high quality modeling system will have links to the popular spreadsheets and databases. That way the data can be extracted from the primary source, alleviating the danger of using out of date information.

It is hard to make other than sweeping generalizations about the role of data gathering in LP modeling. Sometimes one is very fortunate and the data that the modeler requires are immediately available;

sometimes data of the required accuracy are just not present in the organization and the whole LP project has to be abandoned as the costs and time involved in gathering the primary data cannot be justified. Fortunately the latter is a rare event but the authors have found that most modelers consistently underestimate the time involved in acquiring good data.

1.7 References and further material

Linear Programming is often dealt with in the form of a few chapters in general books about Operations Research, such as the one by Winston [Win94]. Other books are entirely dedicated to this topic; let us cite here the works by Chvàtal [Chv83], Schrijver [Sch86], and Bazaraa and Jarvis [BJ90]. The topic of Linear Programming is also discussed in general books on optimization such as [Lue84] and [SN96].

The **Simplex method** that is commonly used for solving linear problems was introduced by Dantzig in the 1940s; his monograph on Linear Programming [Dan63] is re-edited regularly. A more recent method for solving linear problems is the so-called **Interior Point** or **Newton-Barrier** algorithm (original article in [Kar84], see [ARVK89] for a conceptually simpler form). The book by Arbel [Arb93] comes with a collection of software; a textbook on Linear Programming based on the Interior Point method is [Roo97]. Interior Point algorithms are available as part of commercial software packages such as Xpress-MP.

As mentioned at different places, especially in the following Chapter 2, commercial LP solvers usually do not work with a problem in the form it is given but transform it using so-called **presolving** techniques to simplify the problem and to reduce its size (see for instance [AA95]). At the end of the solution algorithm the transformation is reversed (**postsolving**) to provide the user with the solution to his original problem.

Chapter 2
Typical LP model constructs

We move now to a more disciplined approach to modeling. Though we have noted that modeling is an art, this does not mean that we should treat each modeling exercise in an *ad hoc* way. Rather, we need to learn from our previous modeling experience so as to increase our productivity and skill in the next modeling work we do.

A natural way to approach any new modeling exercise is to try to understand the constraints in terms of constraint types that we have seen before and so know how to model. If we can do a disaggregation of the overall set of constraints into individual constraints that we know how to model, then we can feel quite confident that we will be able to complete the whole model.

So we shall look at some typical constraint types. We will first give various 'word descriptions' of constraints, that is, phrases in which the constraint might be articulated by the person responsible for the system we are modeling. We will then give an algebraic formulation of the constraint, and see how it might be generalized. The presentation below is given in terms of the following constraint types: bounds (Section 2.1), flow constraints (Section 2.2), simple resource constraints (Section 2.3), material balance constraints (Section 2.4), quality requirements (Section 2.5), accounting constraints (Section 2.6), blending constraints (Section 2.7), modes (Section 2.8), soft constraints (Section 2.9), and as a special form of non-constrained 'constraints', objective functions (Section 2.10),

It should be noted that these constraint types are somewhat arbitrary — another presentation could have categorized them in different ways.

2.1 Simple upper and lower bounds

A) 'Marketing tells me that we cannot sell more than 100 units of Product 4 this period.'

B) 'I'm committed to sending at least 20 tonnes of Blend 6 to Boston tomorrow.'

C) 'I have to send exactly 30 tonnes of Blend 5 to Paris next period.'

D) 'Electricity can flow in either direction along the Channel Cable.'

In **A**, one of the decision variables will be the amount of Product 4 that we sell, and suppose this is variable $sell_4$. The words say that there is an **upper limit** or **upper bound** of 100 on the single variable $sell_4$, *i.e.*

$$sell_4 \leq 100$$

In case **B**, we have a restriction in the opposite direction. If the decision variable is, say, $send_{b6,Boston}$, then we have a **lower limit** or **lower bound** on the single variable,

$$send_{b6,Boston} \geq 20$$

Note that this is more restrictive than the normal non-negativity constraint

$$send_{b6,Boston} \geq 0$$

We can see case **C**, where the decision variable might be called $send_{b5,Paris}$, as an example of a variable being **fixed**, or equivalently that its lower and upper bounds are identical, *i.e.*

$$send_{b5,Paris} = 30$$

or

$$send_{b5,Paris} \geq 30 \text{ and } send_{b5,Paris} \leq 30$$

One might ask 'why have $send_{b5,Paris}$ as a decision variable at all, as it is fixed and there is no decision to make about its value?'. If every day we always sent exactly 30 tonnes to Paris then it might be legitimate to remove this from the decision process, remembering to account for the resources that are used in the blend, the transport capacity that is used up in moving the blend, *etc.* But of course there will inevitably come a day when Paris needs more of Blend 5, and we have to spend time recalculating the resource availabilities, transport capacity *etc.*, net of the new Parisian demand. We will get things wrong unless someone remembers to do the recalculation; and of course forgetting to redo the sums is likely when we are having to rejig our production plan because of the extra demands from other cities in France (it is Bastille Day tomorrow). What we gain by removing one variable from the model is not worth the risk arising from later inflexibility.

The last case, **D**, expresses the fact that sometimes a variable is not bound by the normal non-negativity constraint, and in this situation it is said to be **free**. Since, as we have mentioned before, all decision variables are by default assumed to be non-negative, we have to explicitly tell our modeling software that the decision variable here (which we shall call *eflow*), denoting the flow from Britain to France where a positive value means that electricity is flowing from Britain to France and a negative value means that it is flowing in the reverse direction is not non-negative

$$-\infty \leq eflow \leq +\infty$$

In Mosel, this is indicated by the notation

```
eflow is_free
```

Free variables can be replaced by normal non-negative variables, *e.g.*

$$eflow = eflow_{BF} - eflow_{FB}$$

where $eflow_{BF}$ ($eflow_{FB}$) is the non-negative flow from Britain to France (respectively, France to Britain), but since optimization software can handle free variables directly there may be no reason to do this transformation. If, however, there is some asymmetry in the flows (perhaps a flow from Britain to France is taxed in some way by the French government, whereas a flow in the other direction is not taxed), the $eflow_{BF}$ and $eflow_{FB}$ variables may be required anyway by other parts of the model, so the substitution is harmless, or even necessary.

2.2 Flow constraints

Flow constraints arise where one has some sort of divisible item like electricity, water, chemical fluids, traffic *etc.* which can be divided into several different streams, or alternatively streams can come together. Some word formulations might be

A) 'I have got a tank with 1000 liters in it and 3 customers C1, C2 and C3 to supply.'

B) 'I buy in disk drives from 3 suppliers, S1, S2 and S3. Next month I want to have at least 5000 disk drives arriving in total.'

C) 'I have 2 water supplies into my factory, S1 and S2. I get charged by the water supplier for the amount of water that enters my site. I lose 1% of the water coming in from S1 by leakage and 2% of that from S2. My factory needs 100,000 gallons of water a day.'

Case **A** is where there is an outflow from a single source, whilst case **B** is where we want to model a total inflow. Case **C** is a little more complicated, as we have to model losses.

In case **A** we have flow variables $supply_1$, $supply_2$ and $supply_3$, being the amounts we supply to the three customers. The total amount supplied is the sum of these three decision variables *i.e.* $supply_1 + supply_2 + supply_3$, and this total must be less than or equal to the 1000 liters we have available. Thus we have

$$supply_1 + supply_2 + supply_3 \leq 1000$$

(providing we have defined the units of supply to be liters).

Case **B** is very similar. The obvious decision variables are buy_1, buy_2 and buy_3, and the total amount we buy is $buy_1 + buy_2 + buy_3$. Our requirement is for at least 5000 drives in total, so the constraint is

$$buy_1 + buy_2 + buy_3 \geq 5000$$

If we wanted exactly 5000 drives, then the constraint would be

$$buy_1 + buy_2 + buy_3 = 5000$$

Case **C** is more challenging. 'I have 2 water supplies into my factory, S1 and S2. I get charged by the water supplier for the amount of water that enters my site. I lose 1% of the water coming in from S1 by leakage and 2% of that from S2. My factory needs 100,000 gallons of water a day.'

There are several different ways we might approach the modeling. One way is to define decision variables as follows: buy_1 and buy_2 are the amounts we buy from supplier 1 and 2 respectively, and get_1 and get_2 are the amounts that we actually get (after the losses). If we measure the decision variables in thousands of gallons, then we have the requirement constraint

$$get_1 + get_2 = 100$$

and we can model the losses as follows (**loss equations**):

$$get_1 = 0.99 \cdot buy_1$$
$$get_2 = 0.98 \cdot buy_2$$

since, for example, we only receive 98% of the water from supplier 2 that we have to pay for. Then the objective function would, among other entries, have terms $PRICE_1 \cdot buy_1 + PRICE_2 \cdot buy_2$, where $PRICE_1$ and $PRICE_2$ are the prices per thousand gallons of water from the two suppliers.

We have the option of using the loss equations to substitute for buy_1 and buy_2 (or, alternatively, get_1 and get_2), and doing so would reduce the number of equations and variables by 2. But as we have discussed before, it almost certainly is not worth the effort, and the LP solver should be clever enough to do the work for us automatically anyway if it thinks it will reduce the solution time.

2.3 Simple resource constraints

We have already seen examples of this sort of constraint in the chess set problem where, to remind you, we had limited amounts of lathe-hours and boxwood, and we had to create a production plan that used neither too many lathe-hours nor too much boxwood.

If we generalize what we mean by a resource, all linear programs can be formulated as maximizing some objective function subject to not using any more of any resource than we have available. But it is not particularly helpful to think of everything as a resource, so we will give more word formulations here which involve real resources.

A) 'I can only get hold of 10,000 connectors a month. Each Industry PC I make needs 8 connectors and each Home PC I make requires 5 connectors.'

B) 'Each tonne of chemical C1 uses 50 grams of a rare catalyst and 1 kg of a certain fine chemical, whereas each tonne of chemical C2 requires 130 grams of the catalyst and 1.5 kg of the fine chemical. I can only afford to buy 10 kilograms of catalyst per month and I can only acquire 200 kg of the fine chemical each month.'

C) 'If I do one unit of activity i it takes up $COST_i$ of my disposable income. My disposable income next week is $150.'

If we consider just one scarce resource then a resource constraint can be characterized by the activities that potentially use up some of that resource, the level at which we are planning to conduct those activities, the availability of the resource and, finally, the amount of the resource that is used per unit of each activity.

These latter numbers, the resources used per unit of activity, are called **technological coefficients**. because they are determined by the technology that we are using. Typically, when we have several products which are each made from several resources, we will have a **technological coefficient matrix** which gives the number of units of each resource required to make one unit of the product. It would be presented in the form of Table 2.1, where, for example, the entry 0.8528 means that to make 1 unit of Product 3 we require 0.8528 units of Resource 3. We have to be careful to use consistent and sensible units for how we measure products and resources.

Case **A** where we stated that 'I can only get hold of 10,000 connectors a month. Each Industry PC I make needs 8 connectors and each Home PC I make requires 5 connectors' has a simple technological matrix:

If we have decision variables $make_I$ and $make_H$, being the number of Industrial and Home computers we plan to make, then the number of connectors we plan to use is the number used making industrial computers plus the number used making Home computers. If we make 1 Industrial computer then we use 8 connectors, so if we make $make_I$ Industrial computers we use $8 \cdot make_I$ connectors; and if we make 1 Home computer then we use 5 connectors, so if we make $make_H$ Home computers we use $5 \cdot make_H$

Table 2.1: Technological coefficient matrix

	Resource 1	Resource 2	Resource 3	...	Resource R
Product 1	1.28235				0.24
Product 2	0.95474				1.8361
Product 3	12.33312	6.128	0.8528		
...				*etc.*	
Product P	11.2322				5.208

Table 2.2: Simple technological coefficient matrix

	Connector
Industry PC	8
Home PC	5

connectors. So in total if we make $make_I$ Industrial computers and $make_H$ Home computers we will use $8 \cdot make_I + 5 \cdot make_H$ connectors. This total must not be greater than the number available *i.e.*

$$8 \cdot make_I + 5 \cdot make_H \leq 10000$$

In case **B** we said 'Each tonne of chemical C1 uses 50 grams of a rare catalyst and 1 kg of a certain fine chemical, whereas each tonne of chemical C2 requires 130 grams of the catalyst and 1.5 kg of the fine chemical. I can only afford to buy 10 kilograms of catalyst per month and I can only acquire 200 kg of the fine chemical each month.' So we have two products and two resources.

Table 2.3: Resource use

	Catalyst	Fine Chemical
C1	50	1.0
C2	130	1.5

We measure the catalyst in grams, and the fine chemical in kg. Defining decision variable $make_1$ and $make_2$ to be the number of tonnes of the two chemicals (C1 and C2) we plan to make, and using an argument similar to the one in case A), we get two constraints, one for the catalyst and one for the fine chemical.

$$50 \cdot make_1 + 130 \cdot make_2 \leq 1000 \quad \text{(catalyst)}$$
$$1.0 \cdot make_1 + 1.5 \cdot make_2 \leq 200 \quad \text{(fine chemical)}$$

We have been careful to keep the units of measurement consistent.

In general, we have one constraint for each resource, relating the amount we are planning to use to the amount available.

Case **C** is now hardly challenging. 'If I do one unit of activity i it takes up $COST_i$ of my disposable income. My disposable income next week is \$150.' The decision variables are $doact_i$, the amount of activity i that I plan to do. Making the linearity assumption that if I do $doact_i$ units of activity i it costs me $COST_i \cdot doact_i$, my total planned expenditure is $\sum_i COST_i \cdot doact_i$ and this can be no more than the money I have (assuming credit is not an option for me!). Then the constraint is

$$\sum_i COST_i \cdot doact_i \leq 150$$

2.4 Material balance constraints

Many systems have constraints which can be expressed in words as 'what goes out must in total equal what comes in'. In other words, there can be no loss of mass when some particular process takes place. If the process is some sort of material flow and the thing that is flowing is incompressible, then sometimes the constraint is expressed in terms of conservation of volume.

Pictorially if we have a process with inflow variables $flowin_i$ where i ranges from 1 to N_{IN} and outflows $flowout_j$ where j ranges from 1 to N_{OUT}, and an inflow L from the outside world, we might have

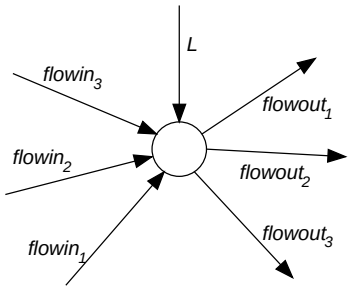

Figure 2.1: Material flow balance in a node

If the flow variables are measured in terms of mass or weight then we will have

$$L + \sum_{i=1}^{M_{IN}} flowin_i = \sum_{j=1}^{N_{OUT}} flowout_j$$

A particular form of material balance constraint occurs when we are accounting for the flow of materials between time periods in a **multi-time period model**, that is, a model where we represent time as a series of intervals, not concerning ourselves with the details of when during an interval events happen. Suppose that we have a simple factory that only makes one product and that we are trying to decide the manufacturing levels ($make_t$) for the next NT periods, *i.e.* t ranges from 1 to NT. Another set of decision variables are $sell_t$, the amount of the product that we decide to sell in time period t. If we have the possibility of storing product from one time period to the next time period we can introduce a further set of decision variables $store_t$ which are the amounts of product to have in stock (inventory) at the end of time period t.

Let us consider the material balance in time period t. In words we can say 'the stock at the end of time period t is equal to the stock at the beginning of time period t, plus what we make, minus what we sell.'

We are faced with a slight problem now — we do not know the decision variable for the stock at the beginning of time period t but we can see that, assuming there is no loss of stock, the stock at the beginning of time period t is the same at the end of time period t -1. So in different words, the previous statement can be phrased as 'the stock at the end of time period t is equal to the stock at the end of time period $t - 1$, plus what we make in period t, minus what we sell in period t'. In algebraic form this can be written

$$stock_t = stock_{t-1} + make_t - sell_t$$

This is deceptively simple but it contains one flaw: there is no time period preceding time period 1, so there is no $stock_0$ variable. We need a special constraint that relates to time period 1 only, and says in words 'the stock at the end of time period 1 is equal to the opening inventory $SINIT$, plus what we make in time period 1, minus what we sell in time period 1', or in algebra

$$stock_1 = SINIT + make_1 - sell_1$$

Other material balance constraints arise when we are talking about raw materials that we buy in from an outside supplier and use in our production process. A word description might be, 'for a particular raw material, the stock of the raw material at the end of time period t is equal to the stock at the end of time period $t - 1$, plus the amount of the raw material that we buy in time period t, minus the amount of raw material that we use in making the products during time period t'. Then considering one raw material we might have decision variables $rbuy_t$, $rstock_t$ and $ruse_t$, so the constraints for all but the first time period are

$$rstock_t = rstock_{t-1} + rbuy_t - ruse_t$$

We have to remember again that there is a special constraint for time period 1 as we already know the opening stock level in that period: it is what we have in stock right now.

It is also likely that the $ruse_t$ variables will be related to the decision variables in another part of the model which represent how much product we are going to make. If we have products indexed by $p = 1, \ldots, NP$ with technological coefficients REQ_{pr}, the number of units of raw material r required by 1 unit of product p, and decision variables $make_{pt}$ denoting the amount of product p that we make in time period t, we can immediately write down an expression for the amount of raw material we use in time period t ($ruse_t$)

as

$$ruse_t = \sum_{p=1}^{NP} REQ_{pr} \cdot make_{pt}$$

(see Section 2.3 'Simple resource constraints' above for an explanation of this).

Since we have just written down an equation for $ruse_t$ we might consider substituting for it in the constraint in which it occurs. This would generally not be a particularly good thing to do, but not particularly bad either. It would reduce the number of variables a little but we are probably going to have to use these variables somewhere else in the model anyway and so the substitution would have to take place everywhere the $ruse_t$ variables appeared in the model. The model would certainly be less comprehensible, and consequently harder to maintain.

A new form of material balance equations of the multi-period type occurs where we have **fixed demands** for our product or products in the *NT* time periods. In other words, the selling decision variables ($sell_t$ in the example above) are fixed. We have the choice of expressing this as a set of simple equality constraints with $sell_t$ variables *e.g.*

$$sell_t = MUSTSELL_t$$

or by substituting the $sell_t$ variables by $MUSTSELL_t$ everywhere they occur. This time the decision as to whether to substitute for the $sell_t$ variables is probably a matter of choice — some modelers will eliminate variables while others might put in the simple equality constraint arguing that the $sell_t$ variables will probably at some stage in the future be of interest in themselves.

2.5 Quality requirements

Here are some word statements

A) 'I cannot have more than 0.02% of sulphur in this gasoline.'

B) 'There must be at least 3% of protein in this dried apricot mixture.'

C) 'This foodstuff must have no less than 5% fat but no more than 10%.'

These sorts of quality requirements frequently occur when we are blending together various raw materials with differing properties to create a final product. Typically each of the raw materials will come with a set of properties, for instance in foodstuffs it might be the fat content or the percentage of carbohydrates; in the petrochemical industry it might be sulphur content or the octane number. For some characteristics of the final product we require that quality specifications are adhered to. For 'bad' things these are usually expressed as a maximum percentages; for desirable properties they will be expressed as a minimum percentage and other properties may have to lie between specified lower and upper bounds (perhaps so that the product has stability or other characteristics that we desire). An extreme form of the latter case is where we have to have an exact percentage of a particular characteristic in the final product.

Consider a number *NR* of raw materials and just one characteristic that we are concerned about in the final product. Suppose that one unit of raw material r contains $CONT_r$ units of the characteristic (for instance $CONT_3$ might be 0.1 if raw material number 3 is 10% fat and the characteristic we are considering is fat). As usual with LP we assume that we know the *CONT* data precisely.

In many manufacturing processes it is reasonable to assume that the characteristic in the resulting end product comes from blending the raw materials linearly. For example if we blend together one kg of a gasoline with 0.01% sulphur and 1 kg of gasoline with 0.03% sulphur, then we will get 2 kg of the mixture with 0.02% sulphur. Whether this assumption of linearity is correct very much depends upon the physics or chemistry of the blending process and in some industries it certainly does not hold. However, making the assumption of linearity, we can see that the proportion of the characteristic in the final product is given by

$$p = proportion = \frac{\text{total mass of constituents}}{\text{total mass of blend}}$$

If the decision variables are $ruse_r$ for the amount of raw material r to use, then we have

$$p = \frac{\sum_{r=1}^{NR} C_r \cdot ruse_r}{\sum_{r=1}^{NR} ruse_r}$$

and we can apply the desired inequality (or equality) to this proportion. For instance, $p \leq 0.1$ (*i.e.* 10%) becomes

$$\frac{\sum_{r=1}^{NR} C_r \cdot ruse_r}{\sum_{r=1}^{NR} ruse_r} \leq 0.1$$

At first sight this does not seem like a linear inequality and indeed it is not as it is a constraint applying to the ratio of two linear expressions. However, by cross multiplying we can get the constraint into the form

$$\sum_{r=1}^{NR} C_r \cdot ruse_r \leq 0.1 \sum_{r=1}^{NR} ruse_r$$

Note: we have to be careful when multiplying inequalities. The direction of the inequality is preserved if we multiply by a non-negative amount, but is reversed if we multiply by a negative amount. Here, we are multiplying both sides of the inequality by the sum of a set of non-negative variables, so the direction of the inequality is maintained.

If we collect together two terms for each decision variable $ruse_r$ we get the linear inequality

$$\sum_{r=1}^{NR} (C_r - 0.1) \cdot ruse_r \leq 0$$

2.6 Accounting constraints, non-constraining 'constraints'

A typical way of building up a model is to construct it in quasi-independent pieces. So, for instance, we might have a constraint expressed as 'the total cost of operating region 1 plus the cost of operating region 2,... plus the cost of operating region 16 must be less than $10M', and then later on in the model we have 16 different equations which capture the cost of operating each of the regions. If we had decision variables *totalcost* and $rcost_i$ this would be expressed as

$$totalcost = \sum_{i=1}^{16} rcost_i$$

where $rcost_i$ = $some\_expression_i$ for i = 1 to 16.

Obviously it is possible to eliminate the $rcost_i$ variables (the cost of operating region *i*) by substituting $some\_expression_i$ into the equation defining *totalcost* but it is very likely that we will want the costs of operating different regions to appear in some report, so we might as well let the LP system calculate the values of these variables for us. We call the $rcost_i$ variables **accounting variables** and the constraints that define them **accounting constraints**.

Whether you choose to use the LP system to do the accounting for you, or do them afterwards when producing reports, is a matter of taste. In the past, when LP systems were limited and one had to squeeze the problem into as few constraints as possible, it was obviously a good idea to eliminate accounting variables and then to calculate their values from the optimum values of the variables contributing to them. Now, where a few hundred extra constraints are of no consequence it is better to leave accounting variables and constraints in the model.

There is however one subtlety that we must be aware of: it is possible that an accounting variable does not necessarily have to be non-negative. Suppose we had variables $revenue_i$ and $expenditure_i$ which were the revenue and expenditure of plant *i*. If we were to define accounting variables $profit_i$ and accounting constraints

$$profit_i = revenue_i - expend_i$$

to pick up the values of the $profit_i$ variables then we must remember that it is possible for a plant to have negative profitability and so the $profit_i$ variables **must** be declared as **free variables** (unless for some reason we have to run each plant in a profitable mode).

Forgetting that decision variables are by default non-negative is a common cause of error with accounting variables. However, if one just wants the LP system to do accounting for you, it is possible to add extra unconstrained linear expressions to a model, letting the LP system work out the value of the linear expression for you. It used to be the case that adding lots of these expressions could slow down the LP solution process, but modern solvers recognize such constraints and discard them during the solution process, only reinstating them when the rest of the LP has been solved.

2.7 Blending constraints

A very common use of LP modeling is to deal with the situation where we have a set of inputs each with certain percentages of different characteristics and we want to blend these inputs together to get a final

product with certain desired characteristics. Blending typically falls into one of two major categories: either we use **fixed recipes** where the proportion of different inputs is determined in advance (rather like the recipes you see in cookery books which specify the exact weights of the constitutes required to make a certain weight of cake) or we have **variable recipes** where it is up to us to decide, perhaps within limits, the proportions of the inputs that we are going to blend together.

An example of a variable recipe might be in the manufacture of animal feed stuffs which can be made from a wide variety of different raw materials, not necessarily always mixed in the same proportions. An interesting example of recipes which pretend to be fixed but in fact are variable are those found in books that tell you how to mix cocktails. They usually pretend that the proportions of gin, tequila, vermouth etc are fixed, but of course anyone who has been in the position of having restricted stocks of any of these raw materials knows that within limits the recipes are very variable.

Suppose we have 3 raw materials and the constraint in words is 'the ratios of raw material 1, raw material 2 and raw material 3 are 6:3:1'. If we have decision variables raw_1, raw_2 and raw_3 representing the weight of the materials in the blend, then the total weight in the blend is $raw_1 + raw_2 + raw_3$ and we have the constraints

$$\frac{raw_1}{raw_1 + raw_2 + raw_3} = \frac{6}{10}$$
$$\frac{raw_2}{raw_1 + raw_2 + raw_3} = \frac{3}{10}$$
$$\frac{raw_3}{raw_1 + raw_2 + raw_3} = \frac{1}{10}$$

These are ratio constraints, not linear equations, but as before we can convert them into linear equations by cross multiplying to get

$$(1 - 0.6) \cdot raw_1 = 0.6 \cdot raw_2 + 0.6 \cdot raw_3, \quad \text{i. e.}$$
$$0.4 \cdot raw_1 = 0.6 \cdot raw_2 + 0.6 \cdot raw_3$$

The cross multiplication preserves the direction of the inequality since the denominator is always non-negative.

Similarly we get

$$0.7 \cdot raw_2 = 0.3 \cdot raw_1 + 0.3 \cdot raw_3, \quad \text{and}$$
$$0.9 \cdot raw_3 = 0.1 \cdot raw_1 + 0.1 \cdot raw_2$$

One of these equations is redundant as it is implied by the other two but it does no harm to put all three equations into the model. In fact it is probably a good idea to put all three equations down as inevitably at some time in the future we will have a fourth raw material and if we try to be too clever in eliminating redundant constraints we will forget that we have omitted a previously redundant equation, and make a mistake that will be hard to detect.

2.8 Modes

A generalization of blending constraints is where we have *M* inputs and *N* outputs which must be in strict proportions. As an example, this might be expressed as 'My 3 inputs have to be used in the ratio of 2 : 1.6 : 2.2 and they produce outputs of 1.6 of output 1 and 1.7 of output 2'. So, for example we might put in 1 kg of input 1, 0.8 kg of input 2, and 1.1 kg of input 3, and get out 0.8 kg of output 1 and 0.85 kg of output 2.

It is easy to see that if we have just one output then this is a simple fixed ratio blending example.

Such *M*-input/*N*-output constraints often arise where we have a plant that we can operate in different ways (**modes**), and the ratios differ for different modes. At any point in time, the plant can only be in one mode.

Consider a very simple example, where we have 3 inputs, 2 outputs and 3 possible operating modes. We present the operating characteristics in the tables below, where we have defined a unit of a mode as 1 hour *i.e.* we have shown the kg of each input used, and output produced, by the plant.

The decision variables are the number of hours the plant spends in each mode *m*, say $usemode_m$. Then the usage of Input 1, for instance, is $20 \cdot usemode_1 + 22 \cdot usemode_2 + 24 \cdot usemode_3$ and the production of output 2 is $17 \cdot usemode_1 + 20 \cdot usemode_2 + 24 \cdot usemode_3$.

Table 2.4: Fixed ratio blending example

	Quantities used/produced		
	Mode 1	**Mode 2**	**Mode 3**
Input 1	20	22	24
Input 2	16	14	13
Input 3	22	21	18
Output 1	16	15	14
Output 2	17	20	24

In any given planning period we will also have a constraint on the total number of hours the plant spends in all the modes. For instance, if we are doing a weekly plan (168 hours), then we will have the constraint that

$$usemode_1 + usemode_2 + usemode_3 \leq 168$$

We are assuming that we can work at most 168 hours in the week, and that we do not lose any time changing from one mode to another. The latter assumption may be totally unrealistic for some plants, and if we cannot assume that change-over times are negligible then we have to use an Integer Programming formulation.

2.9 Soft constraints and 'panic variables'

The constraint that we have just modeled, that we can run a plant for at most 168 hours, is an example of a **hard constraint**. It is impossible to get more than 168 hours into a week. Other examples of hard constraints are ones that relate to physical, chemical or engineering properties, such as a boiler's capacity, or a reaction rate. Other hard constraints come from accounting or definitional constraints (for instance, profit = revenue-cost is a hard constraint, as it is just a definition really).

But there are other, more 'economic' constraints where in reality if we have to violate a constraint we can do so at a cost. If there is a scarce resource, the economic environment invariably sets up a market in that resource, and we can go to the market if we really do need to acquire more of that resource. (This is somewhat of a generalization as a particular resource may actually be so specific to us that we are the only people who can make it to the standards or in the locality or to the timescale we require.)

Consider again the statement of case B of Section 2.3 'Simple resource constraints' above: We said 'Each tonne of chemical C1 uses 50 grams of a rare catalyst and 1 kg of a certain fine chemical, whereas each tonne of chemical C2 requires 130 grams of the catalyst and 1.5 kg of the fine chemical. I can only afford to buy 10 kilograms of catalyst per month and I can only acquire 200 kg of the fine chemical each month.' The key part of this phrasing is the last sentence, where we say we can only afford so much of the catalyst, and we can only acquire 200 kg of the fine chemical. We expressed the fine chemical constraint as

$$1.0 \cdot make_1 + 1.5 \cdot make_2 \leq 200 \quad \text{(fine chemical)}$$

i.e. as a hard constraint. But suppose that if we were willing to pay P per kg we could get more of the fine chemical. Then we could introduce an extra decision variable *buyFC* to represent the number of kg of fine chemical we bought at this price. If we were building a profit maximizing LP then we would have to add a term $-P \cdot buyFC$ to the objective function, accounting for the money we pay for the fine chemical, but now the constraint becomes

$$1.0 \cdot make_1 + 1.5 \cdot make_2 \leq 200 + buyFC \quad \text{(fine chemical, with buy-in)}$$

We now have a **soft constraint**, *i.e.* we can violate the original constraint but at a penalty to our objective function. The *buyFC* variable can be considered a **panic variable**: it is a variable which is there in case we cannot manage with our normal resources and have to resort to the market to satisfy our needs.

The way we have introduced panic variables is not the way they are most commonly used in modeling. We have mentioned before that getting an infeasible solution to an LP is very difficult to diagnose, and good modelers have learned the art of what we might call 'defensive modeling'. One of the worst things that can happen to a model when it has been deployed to end-users is that it gives an infeasible solution message to the poor end-user, who then does not know what to do. Defensive modelers try to avoid this by adding panic variables to constraints that might possibly be violated, at the same time trying to get estimates of how much it will realistically cost to violate the constraint. Often the end-user can help in this, as they will know what can be done to get extra resources — for instance, if it is a question of shutting down the plant or renting a plane to fly in the scarce resource, you may well decide to rent the plane.

Panic variables do not just apply to resource availability constraints; they can also be added to demand constraints. For instance, we might have a constraint that a particular customer C has to have a total supply from our depots of 100 tonnes, modeled as

$$\sum_{d \in DEPOTS} flow_{dC} \geq 100$$

If, however, it is just not possible to supply him then we may undersupply. There may be a contractual cost to doing this, in which case we know what the direct cost will be, or we might have some measure of the cost in terms of loss of goodwill (admittedly these costs are hard to estimate, and can be very subjective).

If a panic variable is non-zero in an optimal solution, then we can infer one of two things: either we have put in too low a penalty cost, or we are truly being forced to use the panic variable to achieve feasibility. Rerunning the model with higher penalty costs will enable us to distinguish these cases. But in either case we have an implementable solution, and we are not left with an end-user staring at a computer screen wondering what on earth to do.

2.10 Objective functions

An objective function is not a constraint, but since it consists of a linear expression usually involving lots of variables its construction has many similarities to the process of building constraints. In fact, now we have so extensively covered modeling constraints there is little new to say about modeling an objective function.

The objective is usually cost (when we minimize) or profit (when we maximize). Most LP systems minimize by default and the industry MPS standard for presenting an LP problem to an optimizer does not specify whether the objective is to be minimized or maximized, so everyone has at one time or another selected the wrong sense for optimization, and come up with a silly answer. One feels foolish, but it is usually obvious that a mistake has been made.

Problems do exist where there is no objective function, and the LP optimizer is being used just to obtain a feasible solution, or to answer the question in a design study as to whether the constraints are too restrictive. But such problems are rare.

2.10.1 Minimax objective functions

Two apparently non-linear objective functions can be converted into LP forms by tricks. The first is where we wish to:

$$\text{minimize } \max(t_1, t_2, t_3, \ldots, t_N)$$

i.e. minimize the maximum of a set of decision variables. We can model this by introducing a new decision variable, say s, and then

$$\begin{aligned} \text{minimize} \quad & s \\ \text{subject to} \quad & s \geq t_1 \\ & s \geq t_2 \\ & s \geq t_3 \\ & \ldots \\ & s \geq t_N \end{aligned}$$

You can see that s has to be no less than each of the t's, and the minimization objective will force it down to take the value of the largest t. Unfortunately, the same trick cannot be applied where we want to minimize $\min(t_1, t_2, t_3, \ldots, t_N)$ or maximize $\max(t_1, t_2, t_3, \ldots, t_N)$.

2.10.2 Ratio objective functions

The second trick enables us to deal with a **ratio objective function** of the form

$$\begin{aligned} \text{minimize} \quad & Obj = \frac{\sum_j N_j \cdot x_j}{\sum_j D_j \cdot x_j} \\ \text{subject to LP constraints} \quad & \sum_j A_{ij} \cdot x_j \sim R_i \end{aligned}$$

where $\sim$ is one of $\leq$, $\geq$ or =.

Define new variables $d = \frac{1}{\sum_j D_j \cdot x_j}$ and $y_j = x_j \cdot d$ (*i.e.* $x_j = y_j \ / \ d$). Then the objective becomes

$$\begin{aligned} Obj &= d \cdot \textstyle\sum_j N_j \cdot x_j \\ &= \textstyle\sum_j N_j \cdot d \cdot x_j \\ &= \textstyle\sum_j N_j \cdot y_j \end{aligned}$$

The definition of d can be linearized by cross multiplying:

$$1 = \sum_j D_j \cdot d \cdot x_j = \sum_j D_j \cdot y_j$$

and into the normal LP constraints we can substitute for x_j thus:

$$\sum_j A_{ij} \cdot x_j \sim R_i$$

or (multiplied by d)

$$d \cdot \sum_j A_{ij} \cdot x_j \sim d \cdot R_i$$

or

$$\sum_j A_{ij} \cdot d \cdot x_j \sim d \cdot R_i$$

i.e. or

$$\sum_j A_{ij} \cdot y_j \sim d \cdot R_i$$

The resulting LP has variables d and y_j. When it has been solved, the optimal values of the original x_j variables can be recovered from the definition $x_j = y_j \ / \ d$.

Note that we are in trouble if variable d can become 0. We also have to taken care if d is always negative, as the signs of the inequalities must be reversed when we multiply by d above.

Chapter 3
Integer Programming models

Though many systems can accurately be modeled as Linear Programs, there are situations where discontinuities are at the very core of the decision making problem. There are three major areas where non-linear facilities are required

- where entities must inherently be selected from a discrete set of possibilities;
- in modeling logical conditions; and
- in finding the global optimum over functions.

Modeling languages let you model these non-linearities and discontinuities using a range of discrete entities and then a Mixed Integer Programming (MIP) optimizer can be used to find the overall (global) optimum of the problem. Usually the underlying structure is that of a Linear Program, but optimization may be used successfully when the non-linearities are separable into functions of just a few variables.

Note that in this book we use the terms Integer Programming (IP) and Mixed Integer Programming interchangeably. Strictly speaking MIP models have both discrete entities and the continuous variables, and pure IP models only have discrete entities. But it seems pedantic to distinguish these two cases unless the distinction is important to our modeling or optimization; and it is not.

The first Section 3.1 of this chapter introduces the different types of IP modeling objects (also referred to as 'global entities'). The following Section 3.2 describes the branch and bound method for solving MIP problems. Binary variables merit special attention: they may be used to formulate do/don't do decisions, implications and other logical expressions, and even products between several binary variables (Section 3.3). As shown in Section 3.4, it would indeed be possible to reduce the whole set of IP modeling objects to binary variables — but this at the expense of efficiency. In combination with real variables, binary variables can be used, among others, to express discontinuities (Section 3.5).

3.1 IP modeling objects: 'global entities'

In principle, all you need in building MIP models are continuous variables and binary variables, where binary variables are decision variables that can take either the value 0 or the value 1. They are often called '0/1 variables' or 'do/don't do' variables. But it is convenient to extend the set of modeling entities to embrace objects that frequently occur in practice.

- **Integer variables**: an integer variable is a variable that must take only an integer (whole number) value.
- **Partial integer variables**: a partial integer variable is a variable that must take an integer value if it is less than a certain user-specified limit L, or any value above that limit.
- **Semi-continuous (SC) variables**: an SC variable is a decision variable that can take either the value 0, or a value between some user specified lower limit L and upper limit U. SCs help model situations where if a variable is to be used at all, it has to be used at some minimum level.
- **Semi-continuous integer variables (SI)**: decision variables that can take either the value 0, or an integer value between some lower limit and upper limit. SIs help model situations where if a variable is to be used at all, it has to be used at some minimum level, and has to be integer.
- **Special Ordered Sets of type 1 (SOS1 or S1)**: an ordered set of non-negative variables at most one of which can take a non-zero value.

- **Special Ordered Sets of type 2 (SOS2 or S2)**: an ordered set of non-negative variables, of which at most two can be non-zero, and if two are non-zero these must be consecutive in their ordering.

Why might some of these variables and sets, which collectively we call global entities, be used? We shall give a brief overview, and later we shall expand on these examples.

Integer variables occur naturally, being most frequently found where the underlying decision variable really has to take on a whole number value for the optimal solution to make sense. If we are modeling the number of trains that leave a railway station in a half-hour interval, then the variable representing this number must take on an integer value — it makes no sense to say that 11.23 trains are to leave.

The idea of **partial integers** arose from a study where car engines in short supply were being allocated to different regions. Some of the regions were large, and took many engines, so it was permissible to round a solution saying the region should take 1654.19 engines to the integer value 1654. But for a regions taking only a few engines, it was not acceptable to round 6.32 engines to 6 engines. It is up to the decision maker to decide the limit *L* below which it is not acceptable to round a fractional answer to a nearby integer. The modeling system allows the decision maker to specify a different *L* for each partial integer variable, if so desired. So partial integers provide some computational advantages in problems where it is acceptable to round the LP solution to an integer if the optimal value of a decision variable is quite large, but unacceptable if it is small.

Semi-continuous variables are useful where, if some variable is to be used at all, its value must be no less than some minimum amount. For example, in a blending problem we might be constrained by practical reasons either to use none of a component or at least 20 kg of it. This type of constraint occurs very frequently in practical problems.

Semi-continuous integer variables often arise in shift planning modeling. If we are going to set up a machine, then we have to run it for a whole number of shifts, and in any case for at least some minimum number of shifts. For example, it might be possible only to run the machine for 0, or 12, 13, 14, ... shifts, with the case of running it for 1, 2, ..., 11 shifts making no economic sense.

Special Ordered Sets of type 1 are often used in modeling choice problems, where we have to select at most one thing from a set of items. The choice may be from such sets as: the time period in which to start a job; one of a finite set of possible sizes for building a factory; which machine type to process a part on, *etc.*

Special Ordered Sets of type 2 are typically used to model non-linear functions of a variable. They are the natural extension of the concepts of Separable Programming, but when embedded in a Branch and Bound code (see below Section 3.2) enable truly global optima to be found, and not just local optima. (A local optimum is a point where all the nearest neighbors are worse than it, but where we have no guarantee that there is not a better point some way away. A global optimum is a point which we know to be the best. In the Himalayas the summit of K2 is a local maximum height, whereas the summit of Everest is the global maximum height.)

Theoretically, models that can be built with any of the entities we have listed above can equally well be modeled solely with binary variables. The reason why modern IP systems have some or all of the extra entities is that they often provide significant computational savings in computer time and storage when trying to solve the resulting model. Most books and courses on Integer Programming do not emphasize this point adequately. We have found that careful use of the non-binary global entities often yields very considerable reductions in solution times over ones that just use binary variables.

3.2 IP solving: the ideas behind Branch and Bound

Most commercial Integer Programming optimizers, including that in Xpress-MP, use a **Branch and Bound** (B&B) search as the basis of the algorithm which locates and proves the optimal solution. It is not the purpose of this book to go into great detail about the ideas behind the solution algorithms, but to understand why the modeling entities we have introduced are useful it is necessary to give here a short explanation of the ideas behind Branch and Bound. The presentation is intended to give you the flavor of what a B&B search does, and is not supposed to be an exact description.

The first idea underpinning B&B is **relaxation**. If we have global entities in the problem, we temporarily forget about their discreteness and solve the resulting problem (the relaxed problem) as an LP. We may be lucky, and the solution values that should be discrete happen by accident to take allowable values. The binaries are all 0 or 1, the integer variables take integer values *etc.* If this fortuitous event occurs, then we have got the integer solution, and the B&B search is over.

If not, we have at least one global entity which is not satisfied, and the solution is said to be integer

infeasible. A binary or integer variable is taking a fractional value, a partial integer is below its critical limit and is fractional, or a Special Ordered Set condition is not satisfied. So we have to do something to move towards integrality. We introduce the notion of **separation**, which is most easily described in the case of an integer variable. Suppose one particular integer variable's (say, variable v) optimal value in the LP solution is 12.3. Then the obvious remark that in a solution either $v \leq 12$ or $v \geq 13$ leads us to separate the initial problem into two sub-problems, one with the added constraint $v \leq 12$ (call this the down branch) and the other with the added constraint $v \geq 13$ (call this the up branch). If we can solve both sub-problems, then the better of the two solutions gives us the solution to the original problem. In other words, we have separated the original problem into two subproblems. We note that since each sub-problem is more constrained than the original problem, the solutions to the sub-problems will certainly be no better than the solution to the original problem.

What do we do now? We have replaced solving the original problem by the task of solving two sub-problems. So for both of the sub-problems we again apply the idea of relaxation, *i.e.* drop the discreteness conditions and solve as LP problems.

At first sight, this seems like nonsense — we have replaced solving one problem by having to solve two sub-problems. However, there is a good argument why the two sub-problems should naturally be more 'integer' than the original problem, where by 'integer' we mean 'more likely to have an integer solution to the LP relaxation'. In the LP solution to the original problem variable v wants to take the value 12.3, but in the two sub-problems we force it away from that value. So in the LP solution of the down branch sub-problem we will have $v = 12$ and in the LP solution of the up branch we will have $v = 13$. In both cases one of the previously fractional variables is taking an integer solution, so in some very vague way we can see that the sub-problems are naturally more 'integer' than the original problem.

However, the LP relaxation solutions to the sub-problems may still be integer infeasible, but now we know what to do. We apply the idea of separation again, selecting an integer infeasible variable.

So the B&B process consists of successively separating on (branching on) variables which are not satisfying their discreteness condition, thereby creating two new sub-problems every time we separate. We can represent this as a tree, drawn upside-down in the usual computer science manner.

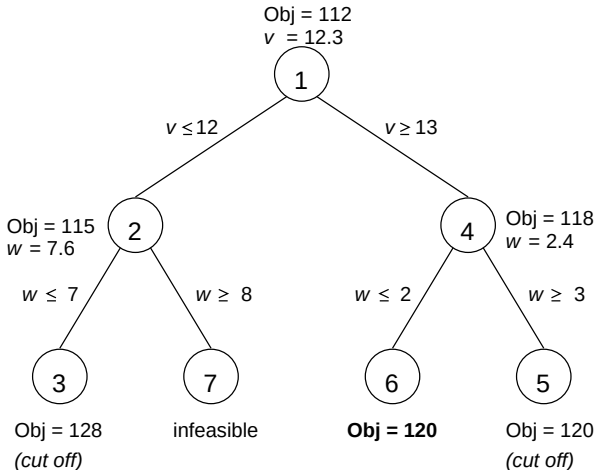

Figure 3.1: Branch and Bound tree

The nodes of the tree represent the solution of the LP relaxation. Beside each node we have written the value of the objective function of the LP relaxation at the node. Note that we assume we have been solving a minimization problem, so the values of the objective functions get worse (bigger) as we go down a branch of the tree, as we are continuously adding new constraints. If the LP solution at a node is integer feasible, we stop exploring down from that node, otherwise we separate on one of the integer infeasible variables, creating two descendant nodes. If the LP at a node is infeasible, we cannot continue from that node as adding constraints to an already infeasible problem will only make it more infeasible, and no descendant can possible give us a feasible integer solution.

If in the course of the search we find an integer solution, we may be able to exploit this fact through the notion of **fathoming**. In the figure above, we find an integer solution with value 120 at node 6. Looking at node 3 we observe that adding constraints will only make its descendants have objective function values worse than 128, but we already have a solution with value 120. So we do not need to look at any descendants of node 3: it has been fathomed or cut-off. Similarly, node 5 is fathomed. And node 7 is fathomed by the fact that it is infeasible.

If we are fortunate and find an integer solution early in the search, then you can see that a very large fraction of the possible nodes may be fathomed, and we will not have to search the entire tree. But if there are N discrete variables, in theory we might have to generate nearly 2^N nodes, and thus have to solve that many LP problems, a formidable task.

As you can see, there are three main decisions to make when doing a B&B search

1. which of the outstanding nodes to select for solving by LP (the **node selection strategy**)
2. which of the fractional integer variables to select for separation (the **variable selection strategy**)
3. which direction to branch on first when you have separated (the **branching direction strategy**)

Though any way of making these decisions will work in theory, since B&B is guaranteed to find the optimal integer solution, given long enough time, the practical effectiveness of a B&B algorithm depends crucially on making the decisions correctly. In many cases it is best to leave the choice to the IP solver; however there is some opportunity for the modeler to influence the solution process by passing knowledge of the semantics of the model to the 'out-of-the-box' solver: often it is helpful to assign branching priorities to decision variables if the values of certain (sets of) variables have a strong impact on the rest of the solution (for instance, if we need to decide whether to open a factory and how much to produce there in different time periods, the decision on the opening should usually be made first). The modeler may also indicate preferred branching directions for single variables ('up' or 'down' branch first) if he has some intuition of the value a variable is likely to take in the solution.

Like other solvers, Xpress-MP has a set of default strategies which have been found to work well on most problems. However, the user should note carefully that sophistication in modeling is important in large scale MIP work. Theoretical developments in the last two decades have lead to insights as to how best to model IP problems, and how to choose between alternative formulations. We cannot over-stress the point that a good formulation can often speed an IP search by orders of magnitude.

3.3 Modeling with binary variables

We have seen that a rich modeling environment supplies more than just binary variables, and we shall later see examples of using these extended entities, but we start the discussion of IP modeling by showing the use of binary variables.

3.3.1 Do/don't do decisions

The most common use of binary variables is when we are modeling activities that either have to be done in their entirety or not done at all. For instance, we might want to model whether we drive a particular truck between city *A* and city *B* on a particular day. In this case we either do drive the truck or we do not drive the truck at all: there are just two options. Typically we model the option of doing something by having a binary variable being equal to 1, and the option of not doing the thing with the binary variable being equal to 0. Thus we just have two choices, of which we must select one.

Associated with the activity and with its binary variable, there may be some other properties. Suppose for instance we are considering what we do with the three trucks that we own. Truck 1 has a capacity of 20 tonnes, truck 2 has a capacity of 25 tonnes and truck 3 has a capacity of 30 tonnes. If on a particular day we decide to send truck 1 to a customer then we either send the entire truck or we do not send the entire truck. So we might have a decision variable (a binary variable) $send_1$ taking the value 1 if we send truck 1 out or 0 if we do not send truck 1. Similarly we will have another binary variable $send_2$ which is 1 if truck 2 goes out and 0 otherwise; and variable $send_3$ taking on the value 1 if truck 3 goes out, and 0 otherwise.

Elsewhere in the model we will probably be interested in the total tonnage that goes out on that particular day and we can see that this is

$$20 \cdot send_1 + 25 \cdot send_2 + 30 \cdot send_3$$

because if truck 1 goes out ($send_1 = 1$) it takes 20 tonnes, if truck 2 goes out it independently takes 25 tonnes, and if truck 3 goes out it carries 30 tonnes, so the total is $20 \cdot send_1 + 25 \cdot send_2 + 30 \cdot send_3$. Suppose that we have a decision variable called *out* which gives the total tonnage leaving our depot on that particular day. Then we have the equation

$$out = 20 \cdot send_1 + 25 \cdot send_2 + 30 \cdot send_3$$

3.3.2 Logical conditions

To show many uses of binary variables we will take as an example a set of projects that we may or may not decide to do. We shall call the projects rather unimaginatively A, B, C, D, E, F, G and H and with each of these projects we will associate a decision variable (a binary variable) which is 1 if we decide to do the project and 0 if we decide not to do the project. We call the corresponding variables a, b, c, d, e, f, g and h. So decision variable a taking on the value 1 means that we do project A, whilst a taking on the value of 0 means that we do not do project A. We are now going to express some constraints in words and see how they can be modeled using these binary variables.

3.3.2.1 Choice among several possibilities

The first constraint that we might impose is 'we must choose no more than one project to do'. We can easily see that this can be expressed by the constraint

$$a + b + c + d + e + f + g + h \leq 1$$

Why is this true? Think of selecting one project, for instance project C. If c is 1 then the constraint immediately says that a, b, d, e, f, g and h must be 0 because there is no other setting for the values of these variables that will satisfy that constraint. So if c is 1 then all the others are 0 and, as there is nothing special about C, we can see immediately that the constraint means that we can only do one project.

If the constraint was that we could do no more than three projects then obviously all we have to do in the constraint above is to replace the 1 by 3 and then we can easily see that in fact we can have no more than three of the 0-1 variables being equal to 1. It is possible to have just two of them or one of them or even none of them being equal to 1 but we can certainly not have four or more being 1 and satisfy the constraint at the same time.

Now suppose that the constraint is that we must choose exactly two of the projects. Then the constraint we can use to model this is

$$a + b + c + d + e + f + g + h = 2$$

Since the binary variables can take only the values 0 or 1, we must have exactly two of them being 1 and all the rest of them being 0, and that is the only way that we can satisfy that constraint.

3.3.2.2 Simple implications

Now consider an entirely different sort of constraint. Suppose that 'if we do project A then we must do project B'. How do we model this? Like a lot of mathematics, it is a question of learning a trick that has been discovered. Consider the following constraint

$$b \geq a$$

To show that this formulation is correct, we consider all the possible combinations of settings (there are four of them) of a and b. First, what happens if we do not do project A. If we also do not do project B then the constraint is satisfied ($0 = b \geq a = 0$) and our word constraint is satisfied. If, on the other hand, we do do project B, then $1 = b \geq a = 0$, and the constraint is again satisfied. Now consider what happens if we actually do project A, *i.e.* $a = 1$. Then the constraint is violated if $b = 0$ and is satisfied (0 is not ≥ 1) if $b = 1$, in other words we do project B. The last case to consider is if we do not do project A, *i.e.* $a = 0$ and we do project B, *i.e.* $b = 1$. Then the constraint is satisfied ($1 \geq 0$) and the word constraint is indeed satisfied. We can lay these constraints out in a table. There are only four possible conditions: {do A, do B}, {do A, do not do B}, {do not do A, do not do B}, {do not do A, do B}, and we can see that the illegal one is ruled out by the algebraic constraint.

Table 3.1: Evaluation of a binary implication constraint

$b \geq a$	a=0	a=1
b=0	Yes	No
b=1	Yes	Yes

The next word constraint we consider is 'if we do project A then we must not do project B'. How might we set about modeling this? The way to think of it is to notice that the property of *notdoing* B can be modeled very easily when we already have a binary variable b representing doing B. We invent a new variable

$$\overline{b} = 1 - b$$

where $\overline{b}$ represents the doing of project $\overline{B}$ *i.e.*, the project 'not doing B'. If $b = 1$ then $\overline{b} = 0$ (in other words, if we do project B then we do not do 'not B') whereas if $b = 0$ then $\overline{b} = 1$ (if we do not do project B then we do project $\overline{B}$). This is very convenient and $\overline{b}$ is called the **complement** of b. We can use this trick frequently. Just above we learned how to model 'if we do A then we must do B' and now we are trying to model 'if we do A then we must not do B', *i.e.* 'if we do A then we must do $\overline{B}$'. As 'if we do A then we must do B' was modeled by $b \geq a$ we can immediately see that the constraint 'if we do A then we must do $\overline{B}$' can be obtained by replacing b by $\overline{b}$ in the constraint, in other words $\overline{b} \geq a$. Replacing $\overline{b}$ by $1 - b$ we get

$$1 - b \geq a$$

or

$$1 \geq a + b$$

or alternatively

$$a + b \leq 1$$

Now that we have obtained this constraint, it is quite obvious. What it says is that if we do project A ($a = 1$) then b must be 0. This is exactly what we wanted to model. The point of the somewhat long-winded argument we showed above, however, is that we have used the result from the first logical constraint that we wanted to model, plus the fact that we have now introduced the notion of the complement of the project, to build up a newer and more complicated constraint from these two primitive concepts. This is what we will do frequently in what follows.

We see an example of this by trying to model the word constraint 'if we do not do A then we must do B', in other word, 'if not A then B'. We can go back immediately to our first logical constraint 'if we do A then we must do B' which was modeled as $b \geq a$. Now we are actually replacing A by not A, so we can see immediately that our constraint is

$$b \geq 1 - a$$

which is

$$a + b \geq 1$$

Again this constraint is obvious now that we have got to it. If we do not do A then $a = 0$, so $b \geq 1$ and since the maximum value of b is 1 then this immediately means that $b = 1$. Again we have just taken our knowledge of how to model 'if A then B' and the notion of the complement of a variable to be able to build up a more complex constraint.

The next constraint to consider is 'if we do project A we must do project B, and if we do project B we must do project A'. We have seen that the first is modeled as $b \geq a$ and the second as $a \geq b$. Combining these two constraints we get

$$a = b$$

in other words projects A and B are selected or rejected together, which is exactly what we expressed in our word constraint.

3.3.2.3 Implications with three variables

The next constraint we consider is 'if we do project A then we must do project B and project C'. The first thing to note is that this is in fact two constraints. One, the first, is 'if we do A then we must do project B' and the second constraint is 'if we do A then we must do project C'. With this observation we can see that the word constraint can be modeled by the two inequalities

$$b \geq a \text{ and } c \geq a$$

so that if $a = 1$, then both $b = 1$ and $c = 1$.

Another constraint might be 'if we do project A then we must do project B or project C'. It is like the previous constraint, except we now have an 'or' in place of the 'and'. The constraint

$$b + c \geq a$$

models this correctly. To see this, consider the following situation. If a is 0 then $b + c$ can be anything, and so b and c are not constrained. If $a = 1$, then one or both of b and c must be 1.

We may also try to model the inverse situation: 'if we do Project B or project C then we must do A'. This is again a case that may be formulated as two separate constraints: 'if we do B then we must do A' and 'if we do C then we must do A', giving rise to the following two inequalities

$$a \geq b \text{ and } a \geq c$$

so that if either $b = 1$ or $c = 1$, then we necessarily have $a = 1$.

A harder constraint to model is the following 'if we do both B and C then we must do A'. How might we model this? One way to think about it is to express it in the following way: 'if we do both B and C then we must not do not-A', or, 'we can do at most two of B, C or not-A' which we would model as

$$b + c + (1 - a) \leq 2$$

or in other words

$$b + c - a \leq 1$$

or perhaps more conventionally

$$a \geq b + c - 1$$

Looking at this last inequality, we can see that there is no effect on a when b and c are 0, or when just one of b and c is 1, but a does have to be ≥ 1 when both b and c are 1. A binary variable having to be greater than or equal to 1 means that the binary variable has to be precisely 1.

3.3.2.4 Generalized implications

Generalizing the previous we now might try to model 'if we do two or more of B, C, D or E then we must do A', and our first guess to this might be the constraint

$$a \geq b + c + d + e - 1$$

Certainly if, say, b and c are both equal to 1 then we have $a \geq 1$ and so a must equal 1, but the problem comes when three of the variables are equal to 1, say b, c and d. Then the constraint says that $a \geq 3 - 1$, *i.e.* $a \geq 2$, which is impossible as a is a binary variable. So we have to modify the constraint as follows

$$a \geq \frac{1}{3} \cdot (b + c + d + e - 1)$$

The biggest value that the expression inside the parentheses can take is 3, if $b = c = d = e = 1$. The $\frac{1}{3}$ in front of the parenthesis means that in the worst case a must be ≥ 1 (so a is equal to 1). But we must verify that the constraint is true if, say, just $b = c = 1$ (and d and e are equal to 0). In this case we have that $a \geq \frac{1}{3} \cdot (1 + 1 + 0 + 0 - 1)$ *i.e.* $a \geq \frac{1}{3}$. But since a can only take on the values 0 or 1 then the constraint $a \geq \frac{1}{3}$ means that a must be 1. This is exactly what we want.

We can generalize this to modeling the statement 'if we do M or more of N projects (B, C, D, ...) then we must do project A' by the constraint

$$a \geq \frac{b + c + d + \ldots - M + 1}{N - M + 1}$$

So far we have only given one way of modeling each of these constraints. We return to the constraint 'if we do B or C then we must do A' which we modeled as two constraints $a \geq b$ and $a \geq c$. It is possible to model this with just one constraint if we take this as a special case of the 'M or more from N' constraint we have just modeled. 'If we do B or C then we must do A' is the same as 'if we do M or more of N projects (B, C, D, ...) then we must do project A' with $M = 1$ and $N = 2$. So the constraint is

$$a \geq \frac{b + c - M + 1}{N - M + 1} \quad \text{i.e.}$$

$$a \geq \frac{b + c - 1 + 1}{2 - 1 + 1} \quad \text{i.e.}$$

$$a \geq \frac{1}{2} \cdot (b + c)$$

So this single constraint is exactly the same in terms of binary variables as the two constraints which we produced before. Which of these two representations is better? In fact the representation in terms of two constraints is better. But both of the two are correct and both will give a correct answer if put into an Integer Programming system. It is just that the first pair of constraints will in general give a solution more rapidly.

More and more complicated constraints can be built up from the primitive ideas we have explored so far. Since these more complicated constraints do not occur very frequently in actual practical modeling we shall not explore them further. Table 3.2 summarizes the formulations of logical conditions we have seen in the different paragraphs of Section 3.3.2.

Table 3.2: Formulation of logical conditions using binary variables

At most one of A, B,...,H	$a+b+c+d+e+f+g+h \leq 1$
Exactly two of A, B,...,H	$a+b+c+d+e+f+g+h = 2$
If A then B	$b \geq a$
Not B	$\overline{b} = 1-b$
If A then not B	$a+b \leq 1$
If not A then B	$a+b \geq 1$
If A then B, and if B then A	$a = b$
If A then B and C	$b \geq a$ and $c \geq a$
If A then B or C	$b+c \geq a$
If B or C then A	$a \geq b$ and $a \geq c$
	or alternatively: $a \geq \frac{1}{2} \cdot (b+c)$
If B and C then A	$a \geq b+c-1$
If two or more of B, C, D or E then A	$a \geq \frac{1}{3} \cdot (b+c+d+e-1)$
If M or more of N projects (B, C, D, ...) then A	$a \geq \frac{b+c+d+\ldots-M+1}{N-M+1}$

3.3.3 Products of binary variables

We move on to modeling the product of binary variables. Suppose we have three binary variables b_1, b_2 and b_3 and we want to model the equation

$$b_3 = b_1 \cdot b_2$$

This is not a linear equation since it involves the product of two variables, so we have to express it in some linear form. There is a trick and it is another one of these tricks that one just has to learn. Consider the following set of three inequalities

$$b_3 \leq b_1$$
$$b_3 \leq b_2$$
$$b_3 \geq b_1 + b_2 - 1$$

Then we claim that this represents the product expression that we wish to model. To see we construct the following Table 3.3.

Table 3.3: Product of two binaries

b_1	b_2	b_3	$b_3 = b_1 \cdot b_2$?	$b_3 \leq b_1$?	$b_3 \leq b_2$?	$b_3 \geq b_1 + b_2 - 1$?
0	0	0	Yes	Yes	Yes	Yes
0	0	1	No	No	No	Yes
0	1	0	Yes	Yes	Yes	Yes
0	1	1	No	No	Yes	Yes
1	0	0	Yes	Yes	Yes	Yes
1	0	1	No	Yes	No	Yes
1	1	0	No	Yes	Yes	No
1	1	1	Yes	Yes	Yes	Yes

We can see that the column headed $b_3 = b_1 \cdot b_2$? is true if and only if we have a 'Yes' in the three columns $b_3 \leq b_1$?, $b_3 \leq b_2$? and $b_3 \geq b_1 + b_2 - 1$?, so the three linear equations do exactly represent and are true at exactly the same time as the product is true.

This is a particularly long winded way of demonstrating the equivalence of the product term and the three linear equations and in fact now we have got it it is actually quite easy to see why these three inequalities are correct. Since b_3 is b_1 multiplied by something that is less than or equal to 1, b_3 will always be less than or equal to b_1 and by a similar argument b_3 will always be less than or equal to b_2. The only further case we have to consider is when both b_1 and b_2 are equal to 1 and then we have to force b_3 to be equal to 1. This is done by the constraint $b_3 \geq b_1 + b_2 - 1$ which is non restrictive if only one or none of b_1 and b_2 are 1 but forces b_3 to be 1 when $b_1 = b_2 = 1$.

Looking at the constraint this way immediately enables us to model for instance

$$b_4 = b_1 \cdot b_2 \cdot b_3,$$

in other words the product of three variables, as the four constraints

$$b_4 \leq b_1$$

$$b_4 \leq b_2$$
$$b_4 \leq b_3$$
$$b_4 \geq b_1 + b_2 + b_3 - 2$$

If any of b_1, b_2 and b_3 are 0 then b_4 must be 0 but if b_1, b_2 and b_3 are 1 then b_4 must be greater than or equal to $3 - 2$, *i.e.* b_4 must be greater than or equal to 1 so b_4 must be 1.

3.3.4 Dichotomies: either/or constraints

All the constraints we have seen so far have had to be satisfied simultaneously, but sometimes we need to model that either Constraint 1 or Constraint 2 has to be satisfied, not necessarily both of them. Consider the problem:

minimize	$Z = x_1 + x_2$
subject to	$x_1 \geq 0, x_2 \geq 0$ and
	Either $2 \cdot x_1 + x_2 \geq 6$ (Constraint 1) or $x_1 + 2 \cdot x_2 \geq 7$ (Constraint 2)

Since we have just two variables, we can graph the feasible region, and we have done this in the figure below where the grey shaded area is the feasible region.

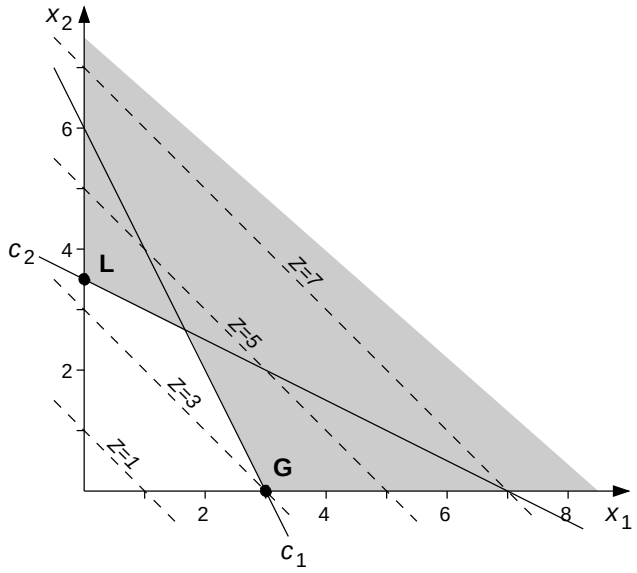

Figure 3.2: Example of either/or constraints

Point L ($x_1 = 0, x_2 = 3.5, Z = 3.5$) is a local optimum. Point G ($x_1 = 3, x_2 = 3, Z = 3$) is the global minimum.

We can model this with one additional binary variable b.

$$2 \cdot x_1 + x_2 \geq 6 \cdot b$$
$$x_1 + 2 \cdot x_2 \geq 7 \cdot (1 - b)$$

To show that this is true, we just have to consider the two cases:

$b = 0$:	$2 \cdot x_1 + x_2 + x_3 \geq 0$	$x_1 + 2 \cdot x_2 + 3 \cdot x_3 \geq 7$	Constraint 2 is satisfied
$b = 1$:	$2 \cdot x_1 + x_2 + x_3 \geq 6$	$x_1 + 2 \cdot x_2 + 3 \cdot x_3 \geq 0$	Constraint 1 is satisfied

3.4 Binary variables 'do everything'

All global entities (general integers, partial integers, semi-continuous variables, and both sorts of Special Ordered Sets) can be expressed in terms of binary variables. However, as shown by the examples in this section, it is usually preferable to use the specific global entities.

3.4.1 General integers

Consider an integer variable v which must take a value between 0 and 10. We could replace this integer variable with four binary variables b_1, b_2, b_3, and b_4 everywhere in the model using the expression

$$v = b_1 + 2 \cdot b_2 + 4 \cdot b_3 + 8 \cdot b_4$$

remembering that we also have to have the constraint

$$b_1 + 2 \cdot b_2 + 4 \cdot b_3 + 8 \cdot b_4 \leq 10$$

This is the binary expansion of v. It has the major disadvantage that we now have four global entities rather than just one global entity. Furthermore, a particular integer value of v, for example 6, can be achieved by many fractional settings of the binary variables. For instance, $v = 6$ can be achieved by $\{b_1 = b_2 = b_3 = 0, b_4 = 0.75\}$ or $\{b_1 = 0, b_2 = 1, b_3 = 1, b_4 = 0\}$, and there are a host of other possibilities. Thus even though v is integer it can be achieved by fractional values of the binary variables and these fractional values would force our integer programming code to do more branching.

Since all commercial integer programming codes allow one to use general integer variables, and with the disadvantage of expanding integers in terms of binary variables, it is usually silly to use binary variables where we really have an underlying integer.

3.4.2 Partial integers

We will postpone the demonstration of how to model partial integer variables in terms of binaries until later (Section 3.5.3), when we have discussed linking binary variables and real variables.

3.4.3 Semi-continuous variables

Suppose we have a semi-continuous variable s which can take on either the value 0 or any value between some lower limit L and some upper limit U. If we introduce a binary variable b then we can represent the semi-continuity of s by the following pair of constraints

$$L \cdot b \leq s$$
$$s \leq U \cdot b$$

We shall see why this is true. There are two cases to consider. Either $b = 0$, in which case s is constrained to be 0, or $b = 1$, in which case s is constrained to lie between L and U. These are exactly the two conditions that we want to impose.

Why are semi-continuous variables useful? It seems a very small penalty just to have to replace a semi-continuous variable by one extra binary variable. The answer is that the semi-continuous property may be satisfied by the variable s, but if we introduce a binary variable the integrality of that binary variable may not be satisfied. Consider the case where s either has to take on the value 0 or it has to take on a real value in the range between 5 and 10. Suppose that we solve the LP relaxation and we get a value for s of 7.5 and we get a value of b of 0.75. This certainly satisfies the two inequalities but the branch and bound search would note that the binary variable was not satisfied (it is not either 0 or 1) and so we would have to go on branching and bounding. But the underlying semi-continuity of s is satisfied; s does indeed lie between its lower bound 5 and its upper bound 10 and so we would not have to branch any further on the s. Thus not having semi-continuous variables in the modeling language or in the optimizer can lead to more branching and bounding in the branch and bound search, and indeed if you need to use semi-continuous variables you should look for an optimizer that has these built into it.

3.4.4 Special Ordered Sets of type 1 (SOS1)

SOS1s are a set of variables, at most one of which can take a strictly positive value, all others being at 0. They most frequently apply where a set of variables are actually 0-1 variables: in other words, we have to choose one from a set of possibilities. These might arise for instance where we are deciding on what size of factory to build, when we have a set of options, perhaps small, medium, large or no factory at all, and we have to chose one and only one size.

To take a small concrete example, suppose that we are considering building a factory and we have options S for a small factory, M for a medium sized factory, B for a big factory and N which is no factory at all. We can model this by introducing four binary variables b_S, b_M, b_B and b_N and model the fact that we can only choose one of these options by the constraint

$$b_N + b_S + b_M + b_B = 1$$

Clearly since these are binary variables adding up to 1, only one of them can be 1 and one of them must be 1. So the SOS1 property applies.

If that was all there was to special ordered sets then it would not be much of an invention. But in fact there is rather more that can be said because we have forgotten about the word **ordered**; in the definition. Suppose that the capacity of the various options for the factories are Small 10,000 tonnes, Medium 20,000 tonnes, Big 38,000 tonnes and of course the 'No factory' has 0 tonnes capacity. One of the things we will be interested in is the selected capacity of the factory that we will be building, and somewhere in the model we would have a decision variable *size* and an equation

$$size = 0 \cdot b_N + 10000 \cdot b_S + 20000 \cdot b_M + 38000 \cdot b_B.$$

We can use the coefficients in this equation (which we call the **reference row**) to order the variables. The order emerging is b_N, b_S, b_M, b_B where we have ordered them by their coefficients in the Reference Row. Now suppose that we have solved the linear programming relaxation and got the following solution:

$$b_N = 0.5,\ b_S = 0,\ b_M = 0,\ b_B = 0.5$$

indicating that we are going to build half of the 0 sized factory and half of the big factory (remember these are the values of the LP relaxation where we have relaxed the integrality of the binary variables).

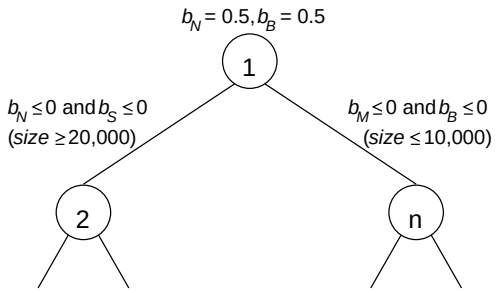

Figure 3.3: Possible SOS branching tree

The branch and bound search will look at that solution and say that there are only two fractional variables b_N and b_B and so it will branch on one or other of those variables. But there is a lot more information in the variables than the fact that they are fractional or not fractional. We are being told by the LP solution that the tonnage that we require from that factory is a half of the biggest one: in other words 19,000 tonnes. Now consider branching on, say, b_N. Either we are forcing the factory to have zero capacity if we branched on b_N up to 1, or we are forcing the factory to be open. It would be much better to have some branching scheme that understood and tried to exploit the fact that the indicated capacity is about 19,000 tonnes. This is precisely what the special ordered set property indicates to the branch and bound tree, and the branching occurs on **sets** of variables rather than on the individual variables which form the Special Ordered Set. The details of precisely how to branch on sets of variables rather than individual variables are outside the scope of this chapter. It is, however, an observed fact that modeling these selection constraints in terms of Special Ordered Sets generally gives a faster branch and bound search. Again as with the case of general integer variables, the solution we get will be exactly the same whichever way we model. The 'only' benefit of using Special Ordered Sets is that the search procedure will generally be noticeably faster.

3.4.5 Special Ordered Sets of type 2 (SOS2)

SOS2s are generally used for modeling piecewise approximations of functions of a single variable. As an example look at the figure below

We wish to represent f as a function of x and we are prepared to consider four line segments between five points. The five points are (R_1, F_1), (R_2, F_2), (R_3, F_3), (R_4, F_4) and (R_5, F_5) and associated with each point i is a weight variable y_i. We have a binary variable b_i associated with each of the intervals (R_1, R_2), (R_2, R_3), (R_3, R_4) and (R_4, R_5). b_i is 1 if the value of x lies between R_i and R_{i+1}.

Then

$$x = \sum_{i=1}^{5} R_i \cdot y_i \quad \text{(reference row)}$$

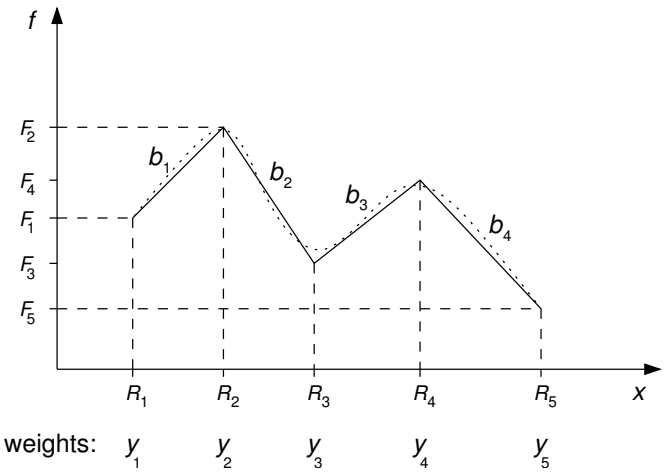

Figure 3.4: Function approximation with SOS2

$$f = \sum_{i=1}^{5} F_i \cdot y_i$$

$$\sum_{i=1}^{5} y_i = 1 \quad \text{(convexity row)}$$

The SOS2 property of having at most two non-zero y_i, and if there are two non-zero then they must be adjacent, means that we are always on the piece-wise linear function. If we did not have the adjacency condition, then by having, say, $y_1 = y_3 = 0.5$ (all the others 0) we could 'cheat' at $x = R_2$ as the function value is not F_2 but $(F_1 + F_3) / 2$, considerably lower (see Figure 3.5).

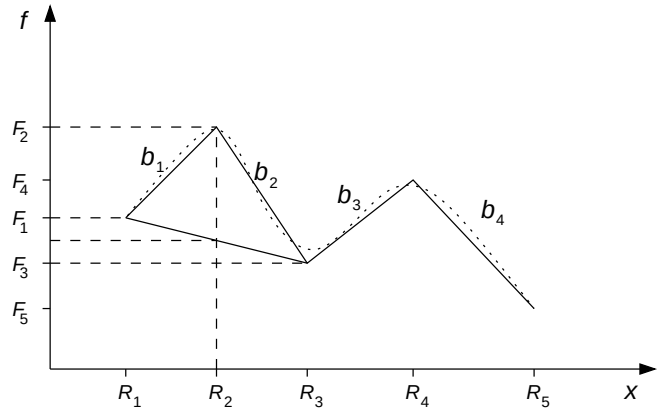

Figure 3.5: Function approximation with SOS2: adjacency condition

Consider the constraints

$$y_1 \leq b_1$$
$$y_2 \leq b_1 + b_2$$
$$y_3 \leq b_2 + b_3$$
$$y_4 \leq b_3 + b_4$$
$$y_5 \leq b_4$$
$$b_1 + b_2 + b_3 + b_4 = 1$$

The last constraint ensures that one and only one b_i is 1 in an acceptable solution. For instance, if b_3 is 1, the constraints on the y_i ensure that $y_1 = y_2 = y_5 = 0$, leaving only the adjacent y_3 and y_4 to be (possibly) non-zero. The other case to consider is when one of the 'end' b_i is 1, say b_1. Then $y_3 = y_4 = y_5 = 0$, and y_1 and y_2, which are adjacent, can be non-zero. So the constraints ensure the SOS2 property, at the expense of introducing $N - 1$ additional binary variables if there are N points.

The same remarks that applied to a reduction in the search times with SOS1 applies to SOS2. It is found in practice that branching on subsets of the variables within a SOS2, rather than on the individual binary variables, is generally beneficial.

3.5 Connecting real variables to binary variables

3.5.1 Modeling fixed costs

The first case we look at is where we want to model fixed costs, that is, where we incur a fixed cost K if a particular real variable x is strictly greater than 0. This fixed cost K might represent the cost of setting up a piece of machinery — we do not incur the fixed cost if we do not set up the piece of machinery but if the output level x of the machine is anything different from zero then we have to incur the fixed cost.

The graph of the cost against the output level x might be represented in the figure below.

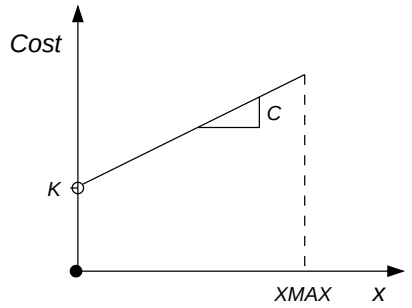

Figure 3.6: Combining fixed and variable cost

Here x represents the throughput of the piece of equipment and the cost consists of two parts. If the throughput is zero then the cost is zero, whereas if the throughput takes a value which is different from 0 then the cost is $Cost = K + C \cdot x$, where C is the per unit cost of output x.

We introduce a binary variable b which is equal to zero if x is equal to zero and equal to 1 if x is strictly greater than zero. Then the cost equals $b \cdot K + C \cdot x$, but we have to ensure that if b is zero then x is zero. We can do this by introducing the constraint

$$x \leq X_{max} \cdot b$$

where X_{max} is the largest value that x can take.

We shall see why this is correct. If $b = 0$ then x is constrained to be less than or equal to zero and so x is zero, whereas if $b = 1$ we just have the constraint that $x \leq X_{max}$, *i.e.* that x is not more than its maximum value (which might be for instance the maximum output of that particular piece of equipment).

3.5.2 Counting

A frequent use of binary variables in modeling is where we wish to be able to count whether a real variable x is different from 0. Often we want to have a constraint that only a certain number of real variables are strictly greater than 0.

An example might be where we have a whole variety of different ingredients that could go into a blend and we have a constraint that perhaps at most 10 of these ingredients can actually go into the blend. If the quantity of item i going into the blend is x_i and we require no more than 10 of these to be non zero, for each item i we introduce a binary variable b_i and have constraints of the form

$$x_i \leq VOL \cdot b_i$$

where VOL is the volume of the blend to be made. These constraints ensure that if a particular x_i is greater than zero then the corresponding b_i must be equal to 1. Then the constraint that says that no more than 10 of the ingredients must be non zero could be expressed as

$$\sum_{i=1}^{10} b_i \leq 10$$

3.5.3 Partial integers again

Earlier we postponed the discussion of how partial integer variables can be expressed in terms of binary variables. Recall that a partial integer is a variable which has to be integral in any acceptable solution if the value is less than some bound, and otherwise can be real. The natural way to express a partial integer p is

$$p = v + x$$

where variable v is an integer that must be less than or equal to U (*i.e.* $0 \leq v \leq U$) and x is a real variable. We have already seen that the integer variable v can be expressed in terms of several binary variables using the binary expansion. Now we have to express the fact that if v is strictly less than U then x must be zero. Suppose that we know some value X_{max} which x is guaranteed not to exceed, *i.e.* an upper bound for x. Then we can introduce a new binary variable b and the following constraints

$$x \leq X_{max} \cdot b$$
$$b \leq v \;/\; U \quad \text{(or, equivalently, } v \geq U \cdot b \text{)}$$

To show that this formulation is valid, consider the two possible values of b. First, if $b = 0$ then $x \leq 0$, *i.e.* x is equal to 0 and $v \geq 0$. If $b = 1$ then $x \leq X_{max}$ and $v \geq U$, which combined with the fact that $v \leq U$ means $v = U$. So if $b = 1$, v is at its maximum, and we can then start having non-zero amounts of x.

As was the case with general integer variables, it seems profligate to express partial integers in terms of binary variables. Firstly, we have to express the integer part in terms of binary variables and then we have to introduce a new binary variable b to model the partial integrality of variable p. Furthermore, as with semi-continuous variables, introducing the extra binary variable will never make the branch and bound tree search quicker and may often make the tree search worse.

Consider the following example. Suppose that $U = 10$ and that the LP solution gives the optimum value of v to be 3 and x to be 0, but the binary variable b is at 0.3. We can see that this satisfies all the inequalities we have given but that the binary variable b is fractional and we will have to branch on b to satisfy integrality. But v is feasible as far as partial integrality is concerned, so we have a case where partial integer variables enable us to say that we are integer feasible but the introduction of the binary variable results in it not being integer feasible and we have to branch further. In models with many partial integer variables this can lead to a great expansion of the branch and bound tree and a lengthening of the search.

3.5.4 Price breaks and economies of scale

Consider Figure 3.7 which represents the situation where we are buying a certain number of items and we get discounts on **all** items that we buy if the quantity we buy lies in certain price bands.

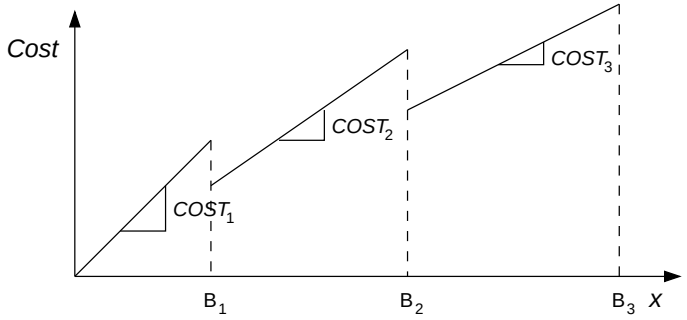

Figure 3.7: All items discount

In the figure we see that if we buy a number of items between 0 and B_1 then for each item we pay an amount $COST_1$, whereas if the quantity we buy lies between B_1 and B_2 we pay an amount $COST_2$ for each item. Finally if we buy an amount between B_2 and B_3 then we pay an amount $COST_3$ for each item. For the sake of concreteness we assume that if we order exactly B_1 items then we get the unit price $COST_2$, whereas if we order just a little less than B_1 then we would get the unit price $COST_1$, and similarly for all the other price breaks. This sort of pricing is called **all item discount pricing**.

To model these item price breaks we should use binary variables b_1, b_2 and b_3, where b_i is 1 if we pay a unit cost of $COST_i$.

Now introduce the real decision variables x_1, x_2 and x_3 which represent the number of items bought at price $COST_1$, $COST_2$ and $COST_3$ respectively. Note that only one of these variables will be non-zero since any quantity we may buy falls into exactly one price range.

The total amount x that we buy is given by

$$x = x_1 + x_2 + x_3$$

We claim that the following set of constraints models the all-item discounts.

$$b_1 + b_2 + b_3 = 1 \tag{3.5.1}$$
$$x_1 \leq B_1 \cdot b_1 \tag{3.5.2}$$
$$B_1 \cdot b_2 \leq x_2 \leq B_2 \cdot b_2 \tag{3.5.3}$$
$$B_2 \cdot b_3 \leq x_3 \leq B_3 \cdot b_3 \tag{3.5.4}$$

Equation (3.5.1) says that exactly one of the prices $COST_i$ is applied. Equations (3.5.2) to (3.5.4) state the relations between the quantity bought and the price intervals. Equation (3.5.2) says that if we buy more then B_1 items ($b_1 = 0$) then $x_1 = 0$, otherwise x_1 indicates the quantity bought at the price $COST_1$. Equations (3.5.3) ensure that if $b_2 = 0$, then $x_2 = 0$, and *vice versa*, whereas if $b_2 = 1$ then x_2 lies within the bounds of the second price range, that is, between B_1 and B_2. Equations (3.5.4) ensure that $x_3 = 0$ if $b_3 = 0$, and *vice versa*, and that if $b_3 = 1$ we buy at least B_2 and at most B_3 items, paying a unit price of $COST_3$.

Only one of the price range indicator variables b_1, b_2, and b_3 will take a non-zero value. Instead of making them explicitly binary variables we can therefore formulate this problem by defining a Special Ordered Set of type 1 (SOS1) over these variables, where the order is given by the values of the breakpoints B_i.

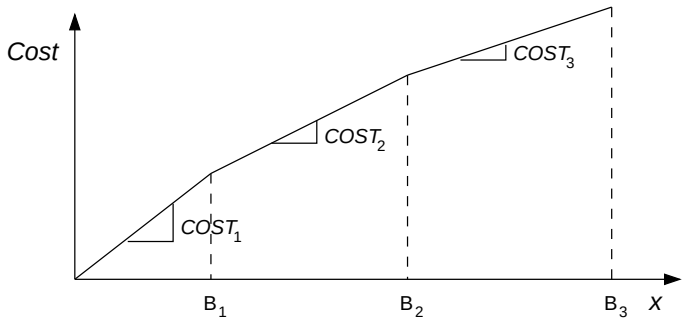

Figure 3.8: Incremental price breaks

Now consider Figure 3.8 which represents the situation where we are buying a certain number of items and we get discounts incrementally. The unit cost for items between 0 and B_1 is C_1, whereas items between B_1 and B_2 cost C_2 each, and items between B_2 and B_3 cost C_3 each.

We can model this using Special Ordered Sets of type 2 (SOS2). At the points 0, B_1, B_2 and B_3 we introduce real valued decision variables w_i ($i = 0, 1, 2, 3$). We also define cost break points CBP_i that correspond to the total cost of buying quantities 0, B_1, B_2 and B_3. So

$$CBP_0 = 0$$
$$CBP_1 = C_1 \cdot B_1$$
$$CBP_2 = CBP_1 + C_2 \cdot (B_2 - B_1)$$
$$CBP_3 = CBP_2 + C_3 \cdot (B_3 - B_2)$$

We then have

$$w_0 + w_1 + w_2 + w_3 = 1$$
$$TotalCost = 0 \cdot w_0 + CBP_1 \cdot w_1 + CBP_2 \cdot w_2 + CBP_3 \cdot w_3$$
$$x = 0 \cdot w_0 + B_1 \cdot w_1 + B_2 \cdot w_2 + B_3 \cdot w_3$$

and the w_i form a SOS2 with reference row coefficients given by the coefficients in the definition of x.

For a solution to be valid, at most two of the w_i can be non-zero, and if there are two non-zero they must be contiguous, thus defining one of the line segments.

Alternative formulation: instead of the SOS2 formulation we may also model the incremental price breaks with the help of binary variables. We use binary variables b_1, b_2 and b_3, where b_i is 1 if we have bought any items at a unit cost of $COST_i$. In addition we introduce the real decision variables x_1, x_2 and x_3 for the number of items bought at price $COST_1$, $COST_2$ and $COST_3$ respectively. Note that we cannot buy any items at price $COST_2$ until we have bought the maximum number of items at the higher price $COST_1$, otherwise the solution would be easy as we buy all items at the least expensive unit price!

The total amount x that we buy is given by

$$x = x_1 + x_2 + x_3$$

and the following set of constraints models the incremental discounts.

$$B_1 \cdot b_2 \leq x_1 \leq B_1 \cdot b_1 \tag{3.5.5}$$
$$(B_2 - B_1) \cdot b_3 \leq x_2 \leq (B_2 - B_1) \cdot b_2 \tag{3.5.6}$$
$$x_3 \leq (B_3 - B_2) \cdot b_3 \tag{3.5.7}$$
$$b_1 \geq b_2 \geq b_3 \tag{3.5.8}$$

Equations (3.5.5) say that if we have bought any in the second price range ($b_1 = b_2 = 1$) then x_1 is fixed to B_1. Equations (3.5.6) ensure that if $b_2 = 0$, then $x_2 = 0$, whereas if $b_2 = 1$ then x_2 is only constrained by the maximum amount that can be bought at the price $COST_2$. Equation (3.5.7) ensures that $x_3 = 0$ if $b_3 = 0$, and that if $b_3 = 1$ then we cannot buy more than the maximum number at price $COST_3$. Equations (3.5.8) ensure that we can only buy at a lower price if we have bought at all the higher prices.

3.5.5 The product of a binary and a real variable

With two real variables, x and y, and a binary variable b, we want to model

$$y = b \cdot x$$

Suppose we have some upper bound U on the value of x. Then consider the following constraints

$$y \leq x \tag{3.5.9}$$
$$y \geq x - U \cdot (1 - b) \tag{3.5.10}$$
$$y \leq U \cdot b \tag{3.5.11}$$

If $b = 0$, then (3.5.11) means that $y = 0$. If $b = 1$, then (3.5.11) is harmless, and we have $y \leq x$ from (3.5.9) and $y \geq x$ from (3.5.10), *i.e.* $y = x$, which is what is desired.

3.6 References and further material

Most of the books cited in the 'References' section of Chapter 1 also contain a part on Integer Programming, for instance the one by Schrijver [Sch86]. An excellent text on Integer Programming is the monograph by Wolsey [Wol98]. The book on combinatorial optimization by Papadimitriou and Steiglitz [PS98] also deals with this topic. Syslo et al. [SDK83] provide Pascal code for some algorithms.

The idea of Branch and Bound stems from Land and Doig [LD60]. For certain huge problems this algorithm may be combined with **column generation**, leading to **Branch and Price** algorithms, a description of this technique and its main application areas is given in [BJN+98].

The notion of Special Ordered Sets was introduced by Beale et al. ([BT70] and [BF76]). For semi-continuous variables see Ashford and Daniel [AD93].

Similarly to presolving for linear problems, commercial MIP software usually performs some work to obtain an improved problem formulation before starting the branch and bound search, and sometimes also at the nodes of the search tree. The techniques that are used may be summarized as **integer preprocessing** and **cut generation**. Integer preprocessing tries to reduce the size of a problem and tighten bounds based on integrality considerations; an overview on standard techniques is given in [Sav94]. Cut generation techniques generate additional constraints (**valid inequalities**, **cutting planes**, or in general **cuts**) that **tighten** the problem formulation: these cuts do not change the solution of the MIP but draw the LP relaxation closer to the MIP. Research related to cuts is a very active field (*e.g.* [GP01] and [MW01]). Cuts are generated based on the identification of certain mathematical structures, but the user may also 'help' by specifying so-called **model cuts** (see the example in Section 10.4).

As mentioned in Section 3.2, to solve large applications efficiently it may be worthwhile experimenting with the different search strategy options provided by a solver. A computational study of search strategies is described in [LS98]. Integer Programming algorithms have made rapid progress in recent years [JNS00]: problems with several thousand discrete variables may now be solved to optimality. However, certain classes of problems still remain extremely difficult, like the example given by [CD99] for which instances with more than forty variables cannot be solved with commercial solvers.

An entirely different method for representing and solving problems with discrete variables is **Constraint Programming** (CP) [VH98]. An introduction to CP is given in the textbook by Marriott and Stuckey [MS98]. Many concepts in (M)IP have their correspondence in CP and *vice versa*; Heipcke [Hei99b] compares and explains the most common terms of both fields. With CP it is often relatively easy to find feasible solution(s) to a problem but proving optimality is a much more difficult task (there is no real correspondence to the bounding information provided by the LP relaxation in MIP). A very active field of research is the integration of both approaches to exploit their complementarity and also to be able to work with continuous variables in the context of Constraint Programming (see for instance [BDB95], [RWH99], [BK98], [Hei99a]).

Chapter 4
Quadratic Programming

Quadratic Programming (QP) is the name given to the problem of finding the minimum (or maximum) of a quadratic function of the decision variables subject to linear equality or inequality constraints. Note that QP problems do not allow quadratic terms in the constraints, only in the objective function. QP problems occur much less frequently than LP or MIP problems, so we shall not discuss them in much depth.

Mixed Integer Quadratic Programs (MIQP) are QPs where some or all of the decision variables have to take on discrete values; in other words, they are MIPs with a quadratic objective function. The discrete constraints can be any of the global objects that we have seen above (binaries, integers, *etc.*)

Since QPs are just LPs with quadratic objective functions, the only thing special to note is modeling the objective function. We now give two typical examples.

4.1 Revenue optimization

If a company wishes to plan the amount q of a product to be sold, and at the same time determine the unit price p at which it is to be sold, then the objective function will contain a quadratic term $p \cdot q$, which is the revenue (price · quantity) from selling the product.

4.2 Portfolio optimization

Portfolio optimization is the classic example of QP. Given various opportunities (assets) in which an investor can place his money, the problem is to minimize the risk of achieving a given level of return. The problem explicitly takes into account the fact that overall risk is not just the sum of the riskiness of the individual assets, as their future performances are highly likely to be correlated: shares tend to go up and down together, and shares in a particular sector tend to be highly correlated. The past variability of a share's price movements can be considered a measure of its risk.

We shall take a small example. Suppose we have € 1 million to invest in three stocks. Let R_i be the random variable representing the annual return on € 1 invested in stock i. Analysis of historical data has lead us to estimate the returns as $E(R_1) = 0.09$, $E(R_2) = 0.07$, and $E(R_3) = 0.06$ where $E(x)$ denotes the expectation of x. The variance of the annual returns have been estimated as $var(R_1) = 0.20$, $var(R_2) = 0.07$, and $var(R_3) = 0.15$, and the covariances as $cov(R_1, R_2) = 0.03$, $cov(R_1, R_3) = 0.04$, $cov(R_2, R_3) = 0.05$.

If x_i is the number of (millions of) Euros invested in stock i, then the annual return is

$$x_1 \cdot R_1 + x_2 \cdot R_2 + x_3 \cdot R_3$$

and the expected annual return is

$$x_1 \cdot E(R_1) + x_2 \cdot E(R_2) + x_3 \cdot E(R_3)$$

If we are seeking a return of at least 7.5%, we must have

$$0.09 \cdot x_1 + 0.07 \cdot x_2 + 0.06 \cdot x_3 \geq 0.075$$

and the constraint that says we spend all the money

$$x_1 + x_2 + x_3 = 1$$

The variance of the portfolio, which we want to minimize, is

$$\begin{aligned} var(x_1 \cdot R_1 + x_2 \cdot R_2 + x_3 \cdot R_3) &= var(x_1 \cdot R_1) + var(x_2 \cdot R_2) + var(x_3 \cdot R_3) \\ &\quad +2 \cdot cov(x_1 \cdot R_1, x_2 \cdot R_2) + 2 \cdot cov(x_1 \cdot R_1, x_3 \cdot R_3) \\ &\quad +2 \cdot cov(x_2 \cdot R_2, x_3 \cdot R_3) \\ &= x_1^2 \cdot var(R_1) + x_2^2 \cdot var(R_2) + x_3^2 \cdot var(R_3) \\ &\quad +2 \cdot x_1 \cdot x_2 \cdot cov(R_1, R_2) + 2 \cdot x_1 \cdot x_3 \cdot cov(R_1, R_3) \\ &\quad +2 \cdot x_2 \cdot x_3 \cdot cov(R_2, R_3) \\ &= 0.20 \cdot x_1^2 + 0.07 \cdot x_2^2 + 0.15 \cdot x_3^2 \\ &\quad +0.06 \cdot x_1 \cdot x_2 + 0.08 \cdot x_1 \cdot x_3 + 0.10 \cdot x_2 \cdot x_3 \end{aligned}$$

The usual non-negativity constraints apply to the x_i. So we have a QP: the objective function has quadratic terms (there are no linear terms, which are allowable in a general QP), and we have linear (in)equalities.

4.3 References and further material

Many standard works on Mathematical Programming or Operations Research also include a section on Quadratic Programming. A good general reference for Quadratic Programming is Fletcher [Fle87]. Beale has published a book entirely dedicated to Quadratic Programming [Bea59].

QP problems occur much less frequently in practice than LP or MIP problems, so we shall just give one application example (Section 13.7).

II. Application examples

Classification of the example problems

The application examples in this part are grouped by application area into ten chapters:

- Chapter 6: Mining and process industries
- Chapter 7: Scheduling problems
- Chapter 8: Planning problems
- Chapter 9: Loading and cutting problems
- Chapter 10: Ground transport
- Chapter 11: Air transport
- Chapter 12: Telecommunication problems
- Chapter 13: Economics and finance
- Chapter 14: Timetabling and personnel planning
- Chapter 15: Local authorities and public services

The following tables give an overview of the examples, including their classification (theoretical problem type), a rating between * (easy) and ***** (difficult), and a list of their modeling specifics and any Mosel features used in the implementation that go beyond the basics explained in Chapter 5. The entry 'NN' in the classification field means that the problem does not correspond to any standard type: the problem may be a mixture of standard types, or have application-specific features for which no classification is available.

The reference numbers of the problems in the tables indicate the names given to the Mosel implementations of the models. For instance, the Mosel file for problem *A-1 Production of alloys* is `a1alloy.mos` and the data for this example is contained in the file `a1alloy.dat`. All application examples belonging to one chapter start with the same letter, the number corresponds to the section number in this chapter.

Table 4.1: Classification in order of appearance (Chapters 6 and 7)

	Problem name and type	Difficulty	Features
A-1	**Production of alloys**		
	Blending problem	*	formulation of blending constraints; data with numerical indices, solution printout, `if-then`, `getsol`
A-2	**Animal food production**		
	Blending problem	*	formulation of blending constraints; data with string indices, `as`, formatted solution printout, use of `getsol` with linear expressions, `strfmt`
A-3	**Refinery**		
	Blending problem	**	formulation of blending constraints; sparse data with string indices, dynamic initialization, dynamic arrays, `finalize`, `create`, union of sets
A-4	**Cane sugar production**		
	Minimum cost flow (in a bipartite graph)	*	`ceil`, `is_binary`
A-5	**Opencast mining**		
	Minimum cost flow	**	encoding of arcs, solving LP-relaxation only
A-6	**Production of electricity**		
	Dispatch problem	**	inline `if`, `is_integer`
B-1	**Construction of a stadium**		
	Project scheduling (Method of Potentials)	***	2 problems; selection with '|', sparse/dense format, naming and redefining constraints, subroutine: `procedure` for solution printing, `forward` declaration
B-2	**Flow shop scheduling**		
	Flow shop scheduling	****	alternative formulation using SOS1
B-3	**Job shop scheduling**		
	Job shop scheduling	***	formulating disjunctions (BigM); `dynamic array`, `range`, `exists`, `forall-do`
B-4	**Sequencing jobs on a bottleneck machine**		
	Single machine scheduling	***	3 different objectives; subroutine: `procedure` for solution printing, `if-then`
B-5	**Paint production**		
	Asymmetric Traveling Salesman Problem (TSP)	***	solution printing, `repeat-until`, cast to `integer`, selection with '|', `round`
B-6	**Assembly line balancing**		
	Assembly line balancing	**	encoding of arcs, `range`

Table 4.2: Classification in order of appearance (Chapters 8 and 9)

	Problem name and type	Difficulty	Features
C-1	**Planning the production of bicycles**		
	Production planning (single product)	***	modeling inventory balance; inline `if`, `forall-do`
C-2	**Production of drinking glasses**		
	Multi-item production planning	**	modeling stock balance constraints; inline `if`, index value 0
C-3	**Material requirement planning**		
	Material requirement planning (MRP)	**	working with index (sub)sets, dynamic initialization, `finalize`, `create`, `as`
C-4	**Planning the production of electronic components**		
	Multi-item production planning	**	modeling stock balance constraints; inline `if`
C-5	**Planning the production of fiberglass**		
	Production planning with time-dependent production cost	***	representation of multi-period production as flow; encoding of arcs, `exists`, `create`, `isodd`, `getlast`, inline `if`
C-6	**Assignment of production batches to machines**		
	Generalized assignment problem	*	
D-1	**Wagon load balancing**		
	Nonpreemptive scheduling on parallel machines	****	heuristic solution requiring sorting algorithm, formulation of maximin objective; nested subroutines: `function` returning heuristic solution value and sorting procedure, `ceil`, `getsize`, `if-then`, `break`, `exit`, all loop types (`forall-do`, `repeat-until`, `while-do`), `setparam`, cutoff value
D-2	**Barge loading**		
	Knapsack problem	**	incremental problem definition with 3 different objectives, `procedure` for solution printing
D-3	**Tank loading**		
	Loading problem	***	2 objectives; data preprocessing, `as`, dynamic creation of variables, procedure for solution printing, `if-then-else`
D-4	**Backing up files**		
	Bin-packing problem	**	2 versions of mathematical model, symmetry breaking; data preprocessing, `ceil`, `range`
D-5	**Cutting sheet metal**		
	Covering problem	*	
D-6	**Cutting steel bars for desk legs**		
	Cutting-stock problem	**	set operation(s) on range sets, `set of integer` (data as set contents)

Table 4.3: Classification in order of appearance (Chapters 10 and 11)

	Problem name and type	Difficulty	Features
E-1	**Car rental**		
	Transport problem	***	data preprocessing, set operations, `sqrt` and `^2`, `if-then-elif`
E-2	**Choosing the mode of transport**		
	Minimum cost flow	**	formulation with extra nodes for modes of transport; encoding of arcs, `finalize`, union of sets, nodes labeled with strings
E-3	**Depot location**		
	Facility location problem	***	modeling flows as fractions, definition of model cuts
E-4	**Heating oil delivery**		
	Vehicle routing problem (VRP)	****	elimination of inadmissible subtours, cuts; selection with '|', definition of model cuts
E-5	**Combining different modes of transport**		
	NN	***	modeling implications, weak and strong formulation of bounding constraints; triple indices
E-6	**Fleet planning for vans**		
	NN	***	`maxlist`, `minlist`, `max`, `min`
F-1	**Flight connections at a hub**		
	Assignment problem	*	
F-2	**Composing flight crews**		
	Bipartite matching	****	2 problems, data preprocessing, incremental definition of data array, encoding of arcs, logical `or` (cumulative version) and `and`, `procedure` for printing solution, `forall-do`, `max`, `finalize`
F-3	**Scheduling flight landings**		
	Scheduling problem with time windows	***	generalization of model to arbitrary time windows; calculation of specific BigM, `forall-do`
F-4	**Airline hub location**		
	Hub location problem	***	quadruple indices; improved (re)formulation (first model not usable with student version), union of index (`range`) sets
F-5	**Planning a flight tour**		
	Symmetric traveling salesman problem	*****	loop over problem solving, TSP subtour elimination algorithm; procedure for generating additional constraints, recursive subroutine calls, working with sets, `forall-do`, `repeat-until`, `getsize`, `not`

Table 4.4: Classification in order of appearance (Chapters 12 and 13)

	Problem name and type	Difficulty	Features
G-1	**Network reliability**		
	Maximum flow with unitary capacities	***	encoding of arcs, `range`, `exists`, `create`, algorithm for printing paths, `forall-do`, `while-do`, `round`
G-2	**Dimensioning of a mobile phone network**		
	NN	**	`if-then`, `exit`
G-3	**Routing telephone calls**		
	Multi-commodity network flow problem	***	encoding of paths, `finalize`, `getsize`
G-4	**Construction of a cabled network**		
	Minimum weight spanning tree problem	***	formulation of constraints to exclude subcycles
G-5	**Scheduling of telecommunications via satellite**		
	Preemptive open shop scheduling	*****	data preprocessing, algorithm for preemptive scheduling that involves looping over optimization, "Gantt chart" printing
G-6	**Location of GSM transmitters**		
	Covering problem	*	modeling an equivalence; sparse data format
H-1	**Choice of loans**		
	NN	*	calculation of net present value
H-2	**Publicity campaign**		
	NN	*	`forall-do`
H-3	**Portfolio selection**		
	NN	**	sets of integers, second formulation with semi-continuous, parameters
H-4	**Financing an early retirement scheme**		
	NN	**	inline `if`, selection with '|'
H-5	**Family budget**		
	NN	**	formulation of monthly balance constraints including different payment frequencies; `as`, `mod`, inline `if`, selection with '|'
H-6	**Choice of expansion projects**		
	NN	**	experiment with solutions: solve LP problem explicitly, "round" some almost integer variable and re-solve
H-7	**Mean variance portfolio selection**		
	Quadratic Programming problem	***	parameters, `forall-do`, `min`, `max`, loop over problem solving

Table 4.5: Classification in order of appearance (Chapters 14 and 15)

	Problem name and type	Difficulty	Features
I-1	**Assigning personnel to machines**		
	Assignment problem	****	formulation of maximin objective; heuristic solution + 2 different problems (incremental definition) solved, working with sets, `while-do`, `forall-do`, negative index values
I-2	**Scheduling nurses**		
	NN	***	2 problems, using `mod` to formulate cyclic schedules; `forall-do`, `set of integer`
I-3	**Establishing a college timetable**		
	NN	***	many specific constraints, tricky (pseudo) objective function; `finalize`
I-4	**Exam schedule**		
	NN	**	symmetry breaking, no objective
I-5	**Production planning with personnel assignment**		
	NN	***	2 problems, defined incrementally with partial re-definition of constraints (named constraints), `exists`, `create`, `dynamic array`
I-6	**Planning the personnel at a construction site**		
	NN	**	formulation of balance constraints using inline `if`
J-1	**Water conveyance / water supply management**		
	Maximum flow problem	**	encoding of arcs, `finalize`, selection with '|'
J-2	**CCTV surveillance**		
	Maximum vertex cover problem	**	encoding of network, `exists`
J-3	**Rigging elections**		
	Partitioning problem	****	algorithm for data preprocessing; file inclusion, 3 nested/recursive procedures, working with sets, `if-then`, `forall-do`, `exists`, `finalize`
J-4	**Gritting roads**		
	Directed Chinese postman problem	****	algorithm for finding Eulerian path/graph for printing; encoding of arcs, `dynamic array`, `exists`, 2 functions implementing Eulerian circuit algorithm, `round`, `getsize`, `break`, `while-do`, `if-then-else`
J-5	**Location of income tax offices**		
	p-median problem	****	modeling an implication, all-pairs shortest path algorithm (Floyd-Warshall); `dynamic array`, `exists`, `procedure` for shortest path algorithm, `forall-do`, `if-then`, selection with '|'
J-6	**Efficiency of hospitals**		
	Data Envelopment Analysis (DEA)	***	description of DEA method; loop over problem solving with complete re-definition of problem every time, `finalize`, naming and declaring constraints

Chapter 5

The basics of Xpress-MP

All problems in this book are formulated using the Xpress-Mosel (for short, Mosel) Language. To run these models, the user has the choice between using the Mosel Command Line Interpreter or the graphical user interface Xpress-IVE. To solve the optimization problems in this book we use the Xpress-Optimizer linear and mixed integer solver by accessing it from the Mosel Language.

All software with reference manuals and the complete set of examples discussed in this book can be downloaded from

http://www.dashoptimization.com/applications_book.html

In the first section of this chapter we show how to execute the small production planning problem from Chapter 1 using the Mosel Command Line Interpreter or Xpress-IVE. Section 5.2 introduces the basics of the Mosel Language.

5.1 Introductory example

A small joinery makes two different sizes of boxwood chess sets. The small set requires 3 hours of machining on a lathe, and the large set requires 2 hours. There are four lathes with skilled operators who each work a 40 hour week, so we have 160 lathe-hours per week. The small chess set requires 1 kg of boxwood, and the large set requires 3 kg. Unfortunately, boxwood is scarce and only 200 kg per week can be obtained. When sold, each of the large chess sets yields a profit of $20, and one of the small chess set has a profit of $5. The problem is to decide how many sets of each kind should be made each week to so as to maximize profit.

In Chapter 1 the transformation of this description into a mathematical model was discussed in some detail and will therefore not be repeated here. An implementation of the model has also already been given (see Section 1.3). We assume that the following Mosel model has been entered into a text file named `chess.mos` (`mos` is the standard file extension expected by Mosel):

```
model Chess
 uses "mmxprs"                    ! We shall use Xpress-Optimizer

 declarations
  xs, xl: mpvar                   ! Decision variables: produced quantities
 end-declarations

 Profit:=  5*xs + 20*xl           ! Objective function
 Boxwood:= 1*xs + 3*xl <=  200    ! kg of boxwood
 Lathe:=   3*xs + 2*xl <=  160    ! Lathehours

 maximize(Profit)                 ! Solve the problem

 writeln("LP Solution:")          ! Solution printing
 writeln(" Objective: ", getobjval)
 writeln("Make ", getsol(xs), " small sets")
 writeln("Make ", getsol(xl), " large sets")
end-model
```

5.1.1 Using Xpress-Mosel

Mosel is an advanced modeling and solving language and environment, where optimization problems can be specified and solved with the utmost precision and clarity. The modeling component of Mosel provides an easy to use yet powerful language for describing optimization problems. Through its modular architecture, Mosel provides access to data in different formats (including spreadsheets and databases) and gives access to a variety of solvers, which can find optimal or near-optimal solutions to a problem. Mosel is provided either as a standalone program (the Mosel Command Line Interpreter used in this book) or in the form of libraries that make it possible to embed a model into a larger application written in a programming language.

To run the model we have entered into the file `chess.mos`, we start Mosel at the command prompt, and type the following sequence of commands

```
mosel
exec chess
quit
```

which will start Mosel, compile the model and (if no syntax error has been detected) run the model, and then quit Mosel. We will see output something like that below, where we have highlighted Mosel's output in bold face.

```
mosel
** Xpress-Mosel **
(c) Copyright Dash Associates 1998-2006
> exec chess
LP Solution:
Objective:  1333.33
Make 0 small sets
Make 66.6667 large sets
Returned value:  0
> quit
Exiting.
```

The same steps may be done immediately from the command line:

```
mosel -c "exec chess"
```

The `-c` option is followed by a list of commands enclosed in double quotes.

If after having started Mosel you type a command that is not recognized by the Mosel Command Line Interpreter (for instance: `h`), Mosel displays the full list of commands (or the possible completions to valid commands) with short explanations.

The different options that may be used from the operating system's command line can be obtained by typing `mosel -h`.

The distribution of Mosel contains several **modules** that add extra functionality to the language. A full list of the functionality of a module can be obtained by using the `exam` command; for instance to see what is provided by the Xpress-Optimizer module `mmxprs`:

```
mosel -c "exam mmxprs"
```

For a complete description of the Mosel Language and the Mosel Command Line Interpreter, the reader is referred to the Mosel Reference Manual, available at

http://www.dashoptimization.com/applications_book.html

From the same address, individual manuals for the Mosel modules can also be downloaded.

5.1.2 Using Xpress-IVE

Xpress-IVE, sometimes called just IVE, is the Xpress **I**nteractive **V**isual **E**nvironment, a complete modeling and optimization development environment running under Microsoft Windows. It presents Mosel in an easy-to-use Graphical User Interface (GUI), with a built-in text editor. IVE can be used for the development, management and execution of multiple models and is ideal for developing and debugging prototype models.

To execute the model file `chess.mos` you need to carry out the following steps.

- Start up IVE.
- Open the model file by choosing *File* > *Open*. The model source is then displayed in the central window (the **IVE Editor**).
- Click the *Run* button (green triangle) or alternatively, choose *Build* > *Run*. The resulting screen display is shown in Figure 5.1.

The **Build** pane at the bottom of the workspace is automatically displayed when compilation starts. 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. Clicking on the error takes the user to the offending line.

When a model is run, the **Output/Input** pane at the right hand side of the workspace window is selected to display program output. Any output generated by the model is sent to this window. IVE will also provide graphical representations of how the solution is obtained, which are generated by default whenever a problem is optimized. The right hand window contains a number of panes for this purpose, dependent on the type of problem solved and the particular algorithm used. IVE also allows the user to draw graphs by embedding subroutines in Mosel models (see the documentation on the website for further detail).

IVE makes all information about the solution available through the **Entities** pane in the left hand window. By expanding the list of decision variables in this pane and hovering over one with the mouse pointer, its solution and reduced cost are displayed. Dual and slack values for constraints may also be obtained.

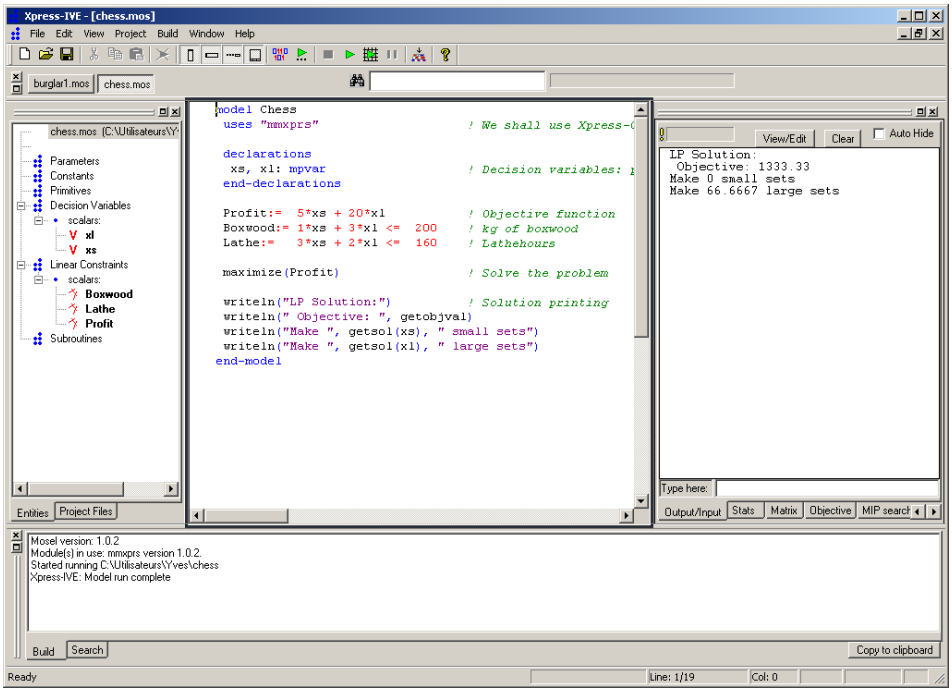

Figure 5.1: Xpress-IVE display after running model `chess.mos`

5.2 Modeling with Mosel

Let us now consider a second, slightly larger model which represents the problem faced by a burglar. With the help of this model we shall explain the basic features of the Mosel language that are used repeatedly in the implementation of the example problems in the following chapters.

5.2.1 The burglar problem

A burglar sees eight items, of different values and weights. He wants to take the items of greatest total value whose total weight is not more than the maximum *WTMAX* he can carry.

We introduce binary variables $take_i$ for all i in the set of all items (*ITEMS*) to represent the decision whether item i is taken or not. $take_i$ has the value 1 if item i is taken and 0 otherwise. Furthermore, let

$VALUE_i$ be the value of item i and $WEIGHT_i$ its weight. A mathematical formulation of the problem is then given by:

$$\text{maximize} \sum_{i \in ITEMS} VALUE_i \cdot take_i \quad \text{(maximize the total value)}$$

$$\sum_{i \in ITEMS} WEIGHT_i \cdot take_i \leq WTMAX \quad \text{(weight restriction)}$$

$$\forall i \in ITEMS : take_i \in \{0, 1\}$$

This problem is an example of a **knapsack problem**. It may be implemented with Mosel as follows:

```
model "Burglar 1"
 uses "mmxprs"

 declarations
  ITEMS = 1..8                  ! Index range for items
  WTMAX = 102                   ! Maximum weight allowed

  VALUE: array(ITEMS) of real   ! Value of items
  WEIGHT: array(ITEMS) of real  ! Weight of items

  take: array(ITEMS) of mpvar   ! 1 if we take item i; 0 otherwise
 end-declarations

! Item:      1   2   3   4   5   6   7   8
  VALUE :: [15, 100, 90, 60, 40, 15, 10, 1]
  WEIGHT:: [ 2,  20, 20, 30, 40, 30, 60, 10]

! Objective: maximize total value
 MaxVal:= sum(i in ITEMS) VALUE(i)*take(i)

! Weight restriction
 sum(i in ITEMS) WEIGHT(i)*take(i) <= WTMAX

! All variables are 0/1
 forall(i in ITEMS) take(i) is_binary

 maximize(MaxVal)                ! Solve the MIP-problem

! Print out the solution
 writeln("Solution:\n Objective: ", getobjval)
 forall(i in ITEMS)  writeln(" take(", i, "): ", getsol(take(i)))
end-model
```

When running this model we get the following output:

```
Solution:
 Objective: 280
 take(1): 1
 take(2): 1
 take(3): 1
 take(4): 1
 take(5): 0
 take(6): 1
 take(7): 0
 take(8): 0
```

The **structure** of this model and Mosel models in general is the following:

- **Model:** Every Mosel program starts with the keyword `model`, followed by a name, and terminates with `end-model`.
- **Declarations:** All objects must be declared in a `declarations` block, unless they are defined unambiguously through an assignment (*e.g.* `i:=1` defines `i` as an integer and assigns it the value 1; in our example the objective function `MaxVal` is defined by assigning it a linear expression). There may be several such `declarations` blocks at different places in a model.
- **Problem definition:** Typically, a model starts with the specification of the data (here: definition of values for `VALUE` and `WEIGHT`), followed by the statement of the problem (here: definition of the objective function `MaxVal`, definition of one inequality constraint, and restricting the variables to be binaries)

- **Solving:** With the procedure `maximize`, we call Xpress-Optimizer to maximize the objective function `MaxVal`. Since there is no 'default' solver in Mosel, we specify that Xpress-Optimizer is to be used with the statement `uses "mmxprs"` at the beginning of the program.
- **Output printing:** The last two lines print out the value of the optimal solution and the solution values for the decision variables.
- **Line breaks:** It is possible to place several statements on a single line, separating them by semicolons (like `x1 <= 4; x2 >= 7`). Conversely, 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.
- **Comments:** As shown in the example, the symbol `!` signifies the start of a comment, which continues to the end of the line. Comments over multiple lines start with `(!` and terminate with `!)`.

We shall now explain certain features used in this model in more detail:

- **Ranges and sets:**

  ```
  ITEMS = 1..8
  ```

  defines a **range set**, that is, a set of consecutive integers from 1 to 8. This range is used as an **index set** for the data arrays (`VALUE` and `WEIGHT`) and for the array of decision variables `take`.

  Instead of using numerical indices, we could, for instance, have defined `ITEMS` as a set of strings by replacing the current definition `ITEMS = 1..8` with the following definition:

  ```
  ITEMS = {"camera", "necklace", "vase", "picture", "tv", "video",
           "chest", "brick"}                  ! Index set for items
  ```

- **Arrays**:

  ```
  VALUE: array(ITEMS) of real
  ```

  defines a one-dimensional array of real values indexed by the range `ITEMS`.

  Multi-dimensional arrays are declared in the obvious way *e.g.*

  ```
  VAL3: array(ITEMS, 1..20, ITEMS) of real
  ```

  declares a 3-dimensional real array. Arrays of decision variables (type `mpvar`) are declared likewise, as shown in our example.

  All objects (scalars and arrays) declared in Mosel are always initialized with a default value:

  `real`, `integer`: 0
  `boolean`: `false`
  `string`: `''` (*i.e.* the empty string)

  The values of data arrays may either be assigned in the model as we show in the example or initialized from file (see Section 5.2.2).

- **Summations:**

  ```
  MaxVal:= sum(i in Items) VALUE(i)*x(i)
  ```

  defines a linear expression called `MaxVal` as the sum

  $$\sum_{i \in Items} VALUE_i \cdot x_i$$

- **Simple looping:**

  ```
  forall(i in ITEMS) take(i) is_binary
  ```

  illustrates looping over all values in an index range. Recall that the index range `ITEMS` is 1, ..., 8, so the statement says that `take(1)`, `take(2)`, ..., `take(8)` are all binary variables.
  There is another example of the use of `forall` at the penultimate line of the model when writing out all the solution values.
  Other types of loops are used in some of the application examples (see the classification tables at the beginning of Part II).

- **Integer Programming variable types:**

  To make an `mpvar` variable, say variable `xbinvar`, into a binary (0/1) variable, we just have to say

  ```
  xbinvar is_binary
  ```

  To make an `mpvar` variable an integer variable, *i.e.* one that can only take on integral values in a MIP problem, we would have

  ```
  xintvar is_integer
  ```

5.2.2 Reading data from text files

The following example illustrates how data may be read into tables from text files. In the Burglar problem instead of having the item data embedded in the model file we have the data in a file. We might have the following Mosel model in a file `burglar2.mos`.

```
model "Burglar 2"
 uses "mmxprs"

 declarations
  ITEMS: set of string           ! Set of items
  WTMAX = 102                    ! Maximum weight allowed
  VALUE: array(ITEMS) of real    ! Value of items
  WEIGHT: array(ITEMS) of real   ! Weight of items
 end-declarations

 initializations from 'burglar.dat'
  VALUE
  WEIGHT
 end-initializations

 declarations
  take: array(ITEMS) of mpvar    ! 1 if we take item i; 0 otherwise
 end-declarations

! Objective: maximize total value
 MaxVal:= sum(i in ITEMS) VALUE(i)*take(i)

! Weight restriction
 sum(i in ITEMS) WEIGHT(i)*take(i) <= WTMAX

! All variables are 0/1
 forall(i in ITEMS) take(i) is_binary

 maximize(MaxVal)                ! Solve the MIP-problem

! Print out the solution
 writeln("Solution:\n Objective: ", getobjval)
 forall(i in ITEMS)  writeln(" take(", i, "): ", getsol(take(i)))
end-model
```

The file `burglar.dat` contains

```
VALUE: [("camera") 15 ("necklace") 100 ("vase") 90 ("picture") 60
         ("tv") 40 ("video") 15 ("chest") 10 ("brick") 1]
WEIGHT:[("camera") 2 ("necklace") 20 ("vase") 20 ("picture") 30
         ("tv") 40 ("video") 30 ("chest") 60 ("brick") 10]
```

The `initializations` block tells Mosel where to get data from to initialize sets and arrays. The order of the data items in the file does not have to be the same as that in the `initializations` block. Note that the contents of the set `ITEMS` is defined indirectly through the index values of the arrays `VALUE` and `WEIGHT`. We only declare the variables once the data has been initialized and hence, the set `ITEMS` is known.

In the application examples, where appropriate, we show how to work with **dynamic** arrays of data and decision variables (see the classification tables at the beginning of Part II).

The data may also be given in the form of a single record, say, `KNAPSACK`. The initialization then takes the following form:

```
 initializations from 'burglar2.dat'
  [VALUE, WEIGHT] as 'KNAPSACK'
 end-initializations
```

and the data file `burglar2.dat` has the following contents:

```
KNAPSACK: [ ("camera")   [ 15  2]
            ("necklace") [100 20]
```

```
("vase")     [ 90 20]
("picture")  [ 60 30]
("tv")       [ 40 40]
("video")    [ 15 30]
("chest")    [ 10 60]
("brick")    [  1 10] ]
```

In the examples of this book we always read data from text files. However, with Mosel it is also possible to read and write data from/to other sources (such as spreadsheets and databases) or input data in memory. For further information, the reader is referred to the documentation on the website.

5.2.3 Reserved words

The following words are reserved in Mosel. The upper case versions are also reserved (*i.e.* `AND` and `and` are keywords but not `And`). Do not use them in a model except with their built-in meaning.

```
and, array, as
boolean, break
case
declarations, div, do, dynamic
elif, else, end
false, forall, forward, from, function
if, import, in, include, initialisations, initializations, integer, inter,
is_binary, is_continuous, is_free, is_integer, is_partint, is_semcont,
is_semint, is_sos1, is_sos2
linctr, list
max, min, mod, model, mpvar
next, not
of, options, or
package, parameters, procedure, public, prod
range, real, record, repeat, requirements
set, string, sum
then, to, true
union, until, uses
version
while
```

Chapter 6
Mining and process industries

This chapter is about fairly simple linear programming problems where one tries to mix or extract ingredients from raw materials subject to quality constraints with the objective of minimizing the total cost. These **blending** or **product mix** problems are typical for process industries that uses big quantities of product in an almost continuous manner: refineries, chemical, metallurgical, and farm-produce industries. These problems are relatively simple because they deal with fractional quantities, so Mixed Integer Programming is not required.

We are going to study three problems of this type: the production of alloys in the metallurgical industry (Section 6.1), animal food production (Section 6.2), and refining of petrol (Section 6.3). These problems concern the **primary sector**, that is, economical activities related to the production of raw materials like agriculture or mining. In addition to the blending problems, this chapter also presents some other problems that belong to this industrial sector: the treatment of cane sugar lots that degrade quickly through fermentation (Section 6.4), the exploitation of an opencast mine (Section 6.5), and the planning of electricity production by a set of power generators (Section 6.6).

6.1 Production of alloys

The company Steel has received an order for 500 tonnes of steel to be used in shipbuilding. This steel must have the following characteristics ('grades').

Table 6.1: Characteristics of steel ordered

Chemical element	Minimum grade	Maximum grade
Carbon (C)	2	3
Copper (Cu)	0.4	0.6
Manganese (Mn)	1.2	1.65

The company has seven different raw materials in stock that may be used for the production of this steel. Table 6.2 lists the grades, available amounts and prices for all raw materials.

Table 6.2: Raw material grades, availabilities, and prices

Raw material	C %	Cu %	Mn %	Availability in t	Cost in €/t
Iron alloy 1	2.5	0	1.3	400	200
Iron alloy 2	3	0	0.8	300	250
Iron alloy 3	0	0.3	0	600	150
Copper alloy 1	0	90	0	500	220
Copper alloy 2	0	96	4	200	240
Aluminum alloy 1	0	0.4	1.2	300	200
Aluminum alloy 2	0	0.6	0	250	165

The objective is to determine the composition of the steel that minimizes the production cost.

6.1.1 Model formulation

We use *RAW* to represent the set of raw materials and *COMP* for the set of components (chemical elements) that are relevant for the grade requirements. We want to determine the quantity use_r of every

raw material r that is used to produce the quantity *produce* of steel to satisfy the given demand *DEM*. We denote by P_{rc} the percentage of the chemical element c in raw material r and $COST_r$ the buying price per kg of r. Furthermore, the minimum and maximum grades $PMIN_c$ and $PMAX_c$ are given for every component c. We obtain the following mathematical model:

$$\text{minimize} \sum_{r \in RAW} COST_r \cdot use_r \tag{6.1.1}$$

$$produce = \sum_{r \in RAW} use_r \tag{6.1.2}$$

$$\forall c \in COMP : \sum_{r \in RAW} P_{rc} \cdot use_r \geq PMIN_c \cdot produce \tag{6.1.3}$$

$$\forall c \in COMP : \sum_{r \in RAW} P_{rc} \cdot use_r \leq PMAX_C \cdot produce \tag{6.1.4}$$

$$\forall r \in RAW : use_r \leq AVAIL_r \tag{6.1.5}$$

$$produce \geq DEM \tag{6.1.6}$$

$$\forall r \in RAW : use_r \geq 0, produce \geq 0 \tag{6.1.7}$$

The objective, given by (6.1.1), is to minimize the total cost that is calculated as the sum of the raw material prices times the quantity used. The constraint (6.1.2) indicates that the resulting product weight is equal to the sum of the weights of the raw materials used for its production.
The constraints (6.1.3) and (6.1.4) impose the limits on the grade of the final product. They have been obtained based on the observation that if the steel contains use_r tonnes of a raw material that contains P_{rc} percent of a chemical element c, then the quantity of this element in the steel is given by the sum

$$\sum_{r \in RAW} P_{rc} \cdot use_{rf}$$

and the percentage of the element c in the final product is given by the ratio

$$\frac{\sum_{r \in RAW} P_{rc} \cdot use_r}{produce}$$

Adding the lower bound $PMIN_c$ to express the constraint on the minimum grade of c results in the following relation:

$$\frac{\sum_{r \in RAW} P_{rc} \cdot use_{rf}}{produce} \geq PMIN_c$$

This relation is non-linear but it can be transformed by multiplying it with *produce* – and we finally obtain the linear constraint (6.1.3). The constraint (6.1.4) is derived following the same scheme.

The constraints (6.1.5) make sure that only the available quantities of raw material are used. The constraint (6.1.6) guarantees that the produced amount of steel satisfies the demand. The last set of constraints (6.1.7) establishes the non-negativity condition for all variables of the problem.

6.1.2 Implementation

The algebraic model translates into the following Mosel program. The correspondence between the two is easy to see. In this model, just like for most others in this book, we read in all data from a separate file. Note that whilst we simply number the raw materials in the model formulation, we read in their names after solving the problem to obtain a more readable solution output.

```
model "A-1 Production of Alloys"
 uses "mmxprs"

 declarations
  COMP = 1..3                    ! Components (chemical elements)
  RAW  = 1..7                    ! Raw materials (alloys)

  P: array(RAW,COMP) of real     ! Composition of raw materials (in percent)
  PMIN,PMAX: array(COMP) of real ! Min. & max. requirements for components
  AVAIL: array(RAW) of real      ! Raw material availabilities
  COST: array(RAW) of real       ! Raw material costs per tonne
  DEM: real                      ! Amount of steel to produce

  use: array(RAW) of mpvar       ! Quantity of raw mat. used
```

```
  produce: mpvar                ! Quantity of steel produced
 end-declarations

 initializations from 'a1alloy.dat'
  P PMIN PMAX AVAIL COST DEM
 end-initializations

! Objective function
 Cost:= sum(r in RAW) COST(r)*use(r)

! Quantity of steel produced = sum of raw material used
 produce = sum(r in RAW) use(r)

! Guarantee min. and max. percentages of every chemical element
 forall(c in COMP) do
  sum(r in RAW) P(r,c)*use(r) >= PMIN(c)*produce
  sum(r in RAW) P(r,c)*use(r) <= PMAX(c)*produce
 end-do

! Use raw materials within their limit of availability
 forall(r in RAW) use(r) <= AVAIL(r)

! Satisfy the demand
 produce >= DEM

! Solve the problem
 minimize(Cost)

! Solution printing
 declarations
  NAMES: array(RAW) of string
 end-declarations

 initializations from 'a1alloy.dat'   ! Get the names of the alloys
  NAMES
 end-initializations

 writeln("Total cost: ", getobjval)
 writeln("Amount of steel produced: ", getsol(produce))
 writeln("Alloys used:")
 forall(r in RAW)
  if(getsol(use(r))>0) then
   write(NAMES(r), ": ", getsol(use(r)),"  ")
  end-if
 write("\nPercentages (C, Cu, Mn): ")
 forall(c in COMP)
  write( getsol(sum(r in RAW) P(r,c)*use(r))/getsol(produce), "%  ")
 writeln

end-model
```

The data file `a1alloy.dat` has the following contents:

```
P: [2.5  0   1.3                          ! Raw material composition
    3    0   0.8
    0    0.3 0
    0   90   0
    0   96   4
    0    0.4 1.2
    0    0.6 0]

PMIN: [2 0.4 1.2]
PMAX: [3 0.6 1.65]                        ! Min. & max. requirements

AVAIL: [400 300 600 500 200 300 250]      ! Raw material availabilities
COST:  [200 250 150 220 240 200 165]      ! Raw material costs

DEM: 500                                  ! Amount of steel to produce

NAMES: ["iron 1" "iron 2" "iron 3" "copper 1" "copper 2"
        "aluminum 1" "aluminum 2"]
```

The solution printout introduces the `if-then` statement of Mosel, it may also be used in its forms

`if-then-else` or `if-then-elif-then-else`. Another observation concerning the solution printout is that the `getsol` function may be applied not only to variables but also to linear expressions.

6.1.3 Results

The required 500 tonnes of steel are produced with 400 tonnes of iron alloy 1, 39.776 tonnes of iron alloy 3, 2.761 tonnes of copper alloy 2, and 57.462 tonnes of aluminum alloy 1. The percentages of Carbon, Copper, and Manganese are 2%, 0.6%, and 1.2% respectively. The total cost of production is €98121.60.

6.2 Animal food production

The company CowFood produces food for farm animals that is sold in two forms: powder and granules. The raw materials used for the production of the food are: oat, maize and molasses. The raw materials (with the exception of molasses) first need to be ground, and then all raw materials that will form a product are blended. In the last step of the production process the product mix is either transformed to granules or sieved to obtain food in the form of powder.

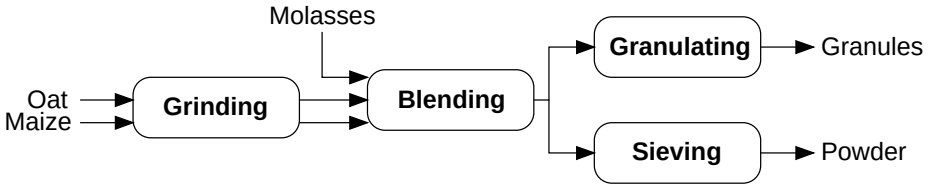

Figure 6.1: Animal food production process

Every food product needs to fulfill certain nutritional requirements. The percentages of proteins, lipids and fibers contained in the raw materials and the required percentages in the final products are listed in Table 6.3.

Table 6.3: Contents of nutritional components in percent

Raw material	Proteins	Lipids	Fiber
Oat	13.6	7.1	7
Maize	4.1	2.4	3.7
Molasses	5	0.3	25
Required contents	≥ 9.5	≥ 2	≤ 6

There are limits on the availability of raw materials. Table 6.4 displays the amount of raw material that is available every day and the respective prices.

Table 6.4: Raw material availabilities and prices

Raw material	Available amount in kg	Cost in €/kg
Oat	11900	0.13
Maize	23500	0.17
Molasses	750	0.12

The cost of the different production steps are given in the following table.

Table 6.5: Production costs in €/kg

Grinding	Blending	Granulating	Sieving
0.25	0.05	0.42	0.17

With a daily demand of nine tonnes of granules and twelve tonnes of powder, which quantities of raw materials are required and how should they be blended to minimize the total cost?

6.2.1 Model formulation

This model is very similar to the preceding except that we now want to produce **two** products and that instead of the minimum and maximum grades we have to fulfill certain nutritional requirements. Let $FOOD = \{1, 2\}$ (1 = granules, 2 = powder) be the set of food product types that are produced, RAW the set of raw materials, and $COMP = \{1, 2, 3\}$ the set of nutritional components (with 1 = protein, 2 = lipids, 3 = fiber). We further use $COST_r$ to denote the price per kg of raw material r, $PCOST_p$ for the cost of production process p in €/kg, REQ_c for the required content of nutritional component c, P_{rc} for the content of c in raw material r, $AVAIL_r$ for the maximum available quantity of raw material r, and DEM_f for the daily demand of food product f.
With variable use_{rf} representing the quantity of raw material r used for the production of food type f and $produce_f$ the amount of food f produced, we obtain the following model:

$$\begin{aligned}\text{minimize } & \sum_{r\in RAW}\sum_{f\in FOOD} COST_r \cdot use_{rf} + \sum_{r\in RAW}\sum_{\substack{f\in FOOD\\ r\neq molasses}} PCOST_{grinding}\cdot use_{rf} \\ & + \sum_{r\in RAW}\sum_{f\in FOOD} PCOST_{blending}\cdot use_{rf} + \sum_{r\in RAW} PCOST_{granulating}\cdot use_{r1} \\ & + \sum_{r\in RAW} PCOST_{sieving}\cdot use_{r2} \qquad (6.2.1)\end{aligned}$$

$$\forall f \in FOOD: \sum_{r\in RAW} use_{rf} = produce_f \qquad (6.2.2)$$

$$\forall f \in FOOD, c \in 1..2: \sum_{r\in RAW} P_{rc}\cdot use_{rf} \geq REQ_c \cdot produce_f \qquad (6.2.3)$$

$$\forall f \in FOOD: \sum_{r\in RAW} P_{r3}\cdot use_{rf} \leq REQ_3 \cdot produce_f \qquad (6.2.4)$$

$$\forall r \in RAW: \sum_{f\in FOOD} use_{rf} \leq AVAIL_r \qquad (6.2.5)$$

$$\forall f \in FOOD: produce_f \geq DEM_f \qquad (6.2.6)$$

$$\forall r \in RAW, f \in FOOD: use_{rf} \geq 0,\ \forall f \in FOOD: produce_f \geq 0 \qquad (6.2.7)$$

The objective function (6.2.1) that is to be minimized is the sum of all costs. The first term is the price paid for the raw materials, the following terms correspond to the production costs of the different process steps. Grinding is applied to all raw materials except molasses, blending to all raw material. Granules ($f = 1$) are obtained through granulating and powder ($f = 2$) by sieving the blended product. The constraint (6.2.2) expresses the fact that the produced quantity of every food type corresponds to the sum of the raw material used for its production. Through the constraints (6.2.3) and (6.2.4) we obtain the required content of nutritional components. They are based on the same reasoning that we saw for the 'grades' in the previous example. The percentage for the content of a nutritional component c in a food product f is given by the following ratio:

$$\frac{\sum_{r\in RAW} P_{rc}\cdot use_{rf}}{produce_f} \qquad (6.2.8)$$

This percentage must be at least REQ_c for the first two components (protein, lipid) and at most REQ_c for the third one (fiber). For the first two, this results in a relation of type (6.2.9), for the third in the relation (6.2.10).

$$\frac{\sum_{r\in RAW} P_{rc}\cdot use_{rf}}{produce_f} \geq REQ_c \qquad (6.2.9)$$

$$\frac{\sum_{r\in RAW} P_{rc}\cdot use_{rf}}{produce_f} \leq REQ_c \qquad (6.2.10)$$

Both relations are non-linear since the variables $produce_f$ appear in the denominator, but as before it is possible to transform them into linear constraints by multiplying them by $produce_f$. Doing so, we obtain the constraints (6.2.3) and (6.2.4).

The constraints (6.2.5) guarantee that the quantities of raw material used remain within their limits of availability. The constraints (6.2.6) serve to satisfy the demand. And the last set of constraints (6.2.7) establishes that all variables are non-negative.

6.2.2 Implementation

Below follows the implementation of this problem with Mosel. Note that in this implementation some of the indexing sets are of numerical type and others are sets of strings. Any indexing set may be assigned its contents at its declaration (a constant set), like *FOOD* and *COMP*, or it may be created dynamically as is the case for for the (unnamed) indexing set of *PCOST* that is filled in when this array is read from file.

```
model "A-2 Animal Food Production"
 uses "mmxprs"

 declarations
  FOOD = 1..2                    ! Food types
  COMP = 1..3                    ! Nutritional components
  RAW  = {"oat", "maize", "molasses"}   ! Raw materials

  P: array(RAW,COMP) of real     ! Composition of raw materials (in percent)
  REQ: array(COMP) of real       ! Nutritional requirements
  AVAIL: array(RAW) of real      ! Raw material availabilities
  COST: array(RAW) of real       ! Raw material prices
  PCOST: array(set of string) of real  ! Cost of processing operations
  DEM: array(FOOD) of real       ! Demands for food types

  use: array(RAW,FOOD) of mpvar  ! Quantity of raw mat. used for a food type
  produce: array(FOOD) of mpvar  ! Quantity of food produced
 end-declarations

 initializations from 'a2food.dat'
  P REQ PCOST DEM
  [AVAIL, COST] as 'RAWMAT'
 end-initializations

! Objective function
 Cost:= sum(r in RAW,f in FOOD) COST(r)*use(r,f) +
        sum(r in RAW,f in FOOD|r<>"molasses") PCOST("grinding")*use(r,f) +
        sum(r in RAW,f in FOOD) PCOST("blending")*use(r,f) +
        sum(r in RAW) PCOST("granulating")*use(r,1) +
        sum(r in RAW) PCOST("sieving")*use(r,2)

! Quantity of food produced corresponds to raw material used
 forall(f in FOOD) sum(r in RAW) use(r,f) = produce(f)

! Fulfill nutritional requirements
 forall(f in FOOD,c in 1..2)
  sum(r in RAW) P(r,c)*use(r,f) >= REQ(c)*produce(f)
 forall(f in FOOD) sum(r in RAW) P(r,3)*use(r,f) <= REQ(3)*produce(f)

! Use raw materials within their limit of availability
 forall(r in RAW) sum(f in FOOD) use(r,f) <= AVAIL(r)

! Satisfy demands
 forall(f in FOOD) produce(f) >= DEM(f)

! Solve the problem
 minimize(Cost)

! Solution printing
 writeln("Total cost: ", getobjval)
 write("Food type"); forall(r in RAW) write(strfmt(r,9))
 writeln("  protein  lipid   fiber")
 forall(f in FOOD) do
  write(strfmt(f,-9))
  forall(r in RAW) write(strfmt(getsol(use(r,f)),9,2))
  forall(c in COMP) write("   ",
   strfmt(getsol(sum(r in RAW) P(r,c)*use(r,f))/getsol(produce(f)),3,2),"%")
  writeln
 end-do

end-model
```

The data file `a2food.dat` has the following contents. Whereas strings in Mosel model files must always be surrounded by single or double quotes, in data files this is only required if a string contains blanks or other non-alphanumeric characters, or starts with a numerical value.

Similarly to the way the raw material availabilities and prices are listed in Table 6.4, the data for the arrays AVAIL and COST are given in a single record labeled RAWMAT. In the initializations block, the names of the data arrays that are to be read from a single record need to be surrounded by [and], followed by the keyword as and the label (name) of the record.

```
P: [(oat 1)     13.6 7.1 7          ! Composition of raw materials
    (maize 1)    4.1 2.4 3.7
    (molasses 1) 5   0.3 25 ]

REQ: [9.5 2 6]                      ! Nutritional requirements

RAWMAT: [(oat)     [11900 0.13]     ! Raw material availabilities and prices
        (maize)    [23500 0.17]
        (molasses)  [750 0.12]]

                                    ! Cost of processing operations
PCOST: [(grinding) 0.25 (blending) 0.05 (granulating) 0.42 (sieving) 0.17]

DEM: [9000 12000]                   ! Demands for food types
```

In this example we again show how the solution may be printed after the problem has been solved. The output is printed in table format with the help of strfmt that for a given string (first parameter) reserves the indicated space (second parameter) or for a number the total space and the number of digits after the decimal point (third parameter). If the second parameter has a negative value, the output is printed left justified.

Since there is only little variation in the solution printing for the large majority of examples, it will be omitted in the listings from now on.

6.2.3 Results

The minimum cost for producing the demanded nine tonnes of granule and twelve tonnes of powder is € 15086.80. The composition of the food products is displayed in Table 6.6.

Table 6.6: Optimal composition of food products

Food type	Oat	Maize	Molasses	Protein	Lipid	Fiber
Granule	5098.56	3719.53	181.91	9.50%	5.02%	6.00%
Powder	6798.07	4959.37	242.55	9.50%	5.02%	6.00%

6.3 Refinery

A refinery produces butane, petrol, diesel oil, and heating oil from two crudes. Four types of operations are necessary to obtain these products: separation, conversion, upgrading, and blending.

The separation phase consists of distilling the raw product into, among others, butane, naphtha, gasoil, and a residue. The residue subsequently undergoes a conversion phase (catalytic cracking) to obtain lighter products. The different products that come out of the distillation are purified (desulfurization or sweetening) or upgraded by a reforming operation that augments their octane value. Finally, to obtain the products that will be sold, the refinery blends several of the intermediate products in order to fulfill the prescribed characteristics of the commercial products. The following drawing gives a simplified overview on the production processes in this refinery.

After the distillation, crude 1 gives 3% butane, 15% naphtha, 40% gasoil, and 15% residue. Crude 2 results in 5% butane, 20% naphtha, 35% gasoil, and 10% residue. The reforming of the naphtha gives 15% butane and 85% of reformate (reformed naphtha). The catalytic cracking of the residue results in 40% of cracked naphtha and 35% of cracked gasoil (note that these percentages do not add up to 100% because the process also produces 15% of gas, 5% coke and another type of residue that are not taken into consideration in our example). The petrol is produced with three ingredients: reformed naphtha (reformate), butane, and cracked naphtha. The diesel oil is obtained by blending sweetened gasoil, cracked gasoil, and cracked naphtha. The heating oil may contain gasoil and cracked naphtha without any restrictions on their proportions.

Certain conditions on the quality of the petrol and diesel oil are imposed by law. There are three important characteristics for petrol: the octane value, vapor pressure and volatility. The octane value is a

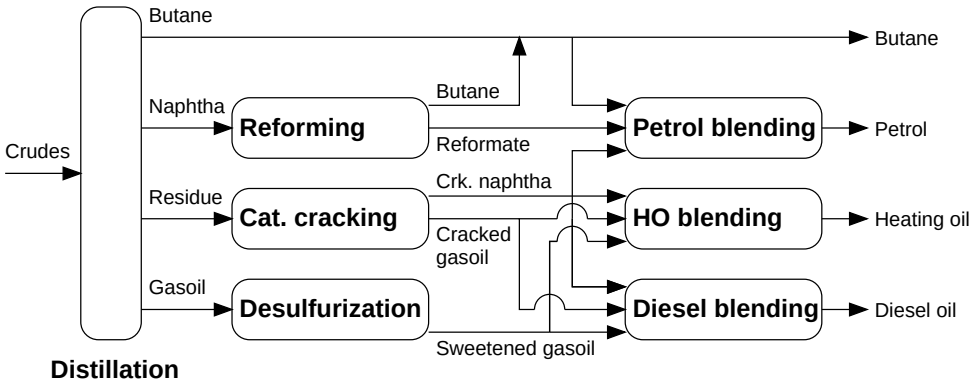

Figure 6.2: Simplified representation of a refinery

measure of the anti-knock power in the motor. The vapor pressure is a measure of the risk of explosion during storage, especially with hot weather. The volatility is a measure for how easy the motor is started during cold weather. Finally, the maximum sulfur content of the diesel oil is imposed by antipollution specifications. The following table summarizes the required characteristics of the final products and the composition of the intermediate ones. Fields are left empty if no particular limit applies. We work with the assumption that all these characteristics blend linearly by weight (in reality, this is only true for the sulfur contents).

Table 6.7: Characteristics of intermediate and final products

Characteristic	Butane	Reformate	Cracked naphtha	Cracked gasoil	Desulfurized gasoil	Petrol	Diesel oil
Octane value	120	100	74	–	–	$\geq$ 94	–
Vapor pressure	60	2.6	4.1	–	–	$\leq$ 12.7	–
Volatility	105	3	12	–	–	$\geq$ 17	–
Sulfur (in %)	–	–	0.12	0.76	0.03	–	$\leq$ 0.05

In the next month the refinery needs to produce 20,000 tonnes of butane, 40,000 tonnes of petrol, 30,000 tonnes of diesel oil, and 42,000 tonnes of heating oil. 250,000 tonnes of crude 1 and 500,000 tonnes of crude 2 are available. The monthly capacity of the reformer are 30,000 tonnes, for the desulfurization 40,000 tonnes and for the cracking 50,000 tonnes. The cost of processing is based on the use of fuel and catalysts for the different operations: the costs of distillation, reforming, desulfurization, and cracking are € 2.10, € 4.18, € 2.04 and € 0.60 per tonne respectively.

6.3.1 Model formulation

Let $produce_{butane}$, $produce_{petrol}$, $produce_{diesel}$, and $produce_{heating}$ be the quantities to produce of butane, petrol, diesel oil, and heating oil. These four products form the set *FINAL* of final products. We need to determine the composition of each of these products. We shall use the following variables for the intermediate products: $produce_{petbutane}$, $produce_{reformate}$, $produce_{petcrknaphtha}$ for the quantities of butane, reformate and cracked naphtha used in the production of petrol (the three products are grouped in the set *IPETROL*); similarly, $produce_{dslgasoil}$, $produce_{dslcrknaphtha}$, and $produce_{dslcrkgasoil}$ for the quantities of sweetened gasoil, cracked naphtha, and cracked gasoil used in the production of diesel oil (set *IDIESEL*) and $produce_{hogasoil}$, $produce_{hocrknaphtha}$, and $produce_{hocrkgasoil}$ for the quantities of sweetened gasoil, cracked naphtha, and cracked gasoil blended to heating oil (set *IHO*). We also define production variables $produce_p$ for the quantities of intermediate products resulting from the different production processes, namely for the sets $IDIST = \{distbutane, naphtha, residue, gasoil\}$ (products resulting from distillation), $IREF = \{refbutane, reformate\}$ (products resulting from reforming), and $ICRACK = \{crknaphtha, crkgasoil\}$ (products resulting from cracking). All final and intermediate products are regrouped in the set $ALLPRODS = FINAL \cup IDIST \cup IREF \cup ICRACK \cup IPETROL \cup IHO \cup IDIESEL$.

In addition to the production variables, we define variables use_c for the quantity of crude $c \in CRUDES$ used in the production.

Our objective is to minimize the production costs, that is, the cost of distilling the crudes plus the cost of reforming, desulfurizing or cracking the intermediate products resulting from the distillation. Hence the

objective function is:

$$\text{minimize} \sum_{c \in CRUDES} COST_c \cdot use_c + \sum_{p \in IDIST} COST_p \cdot produce_p \tag{6.3.1}$$

The following constraints (6.3.2) establish restrictions on the maximum quantities of every component of the raw materials, where $DIST_{cp}$ stands for the contents of the intermediate product p of the crude c.

$$\forall p \in IDIST : produce_p \leq \sum_{c \in CRUDES} DIST_{cp} \cdot use_c \tag{6.3.2}$$

For instance, the total quantity of naphtha going into the reformer ($produce_{naphtha}$) cannot exceed the maximum quantity of naphtha that is obtained from the distillation of the crudes $\left(\sum_{c \in CRUDES} DIST_{c,naphtha} \cdot use_c\right)$.

Similar relations are given for the reforming (6.3.3) and cracking (6.3.4) processes (in both cases, there is only a single input product) where REF_p denotes the percentage composition of naphtha and $CRACK_p$ the percentage composition of the residue:

$$\forall p \in IREF : produce_p \leq REF_p \cdot produce_{naphtha} \tag{6.3.3}$$

$$\forall p \in ICRACK : produce_p \leq CRACK_p \cdot produce_{residue} \tag{6.3.4}$$

The products coming out of the cracking and desulfurization processes may be used for several of the final products. We therefore have the following three equations. Cracked naphtha is used in the production of petrol, heating oil, and diesel oil (6.3.5). Cracked gasoil (6.3.6) and sweetened gasoil (6.3.7) are both used for heating oil and diesel oil.

$$produce_{crknaphtha} = produce_{petcrknaphtha} + produce_{hocrknaphtha} + produce_{dslcrknaphtha} \tag{6.3.5}$$

$$produce_{crkgasoil} = produce_{hocrkgasoil} + produce_{dslcrkgasoil} \tag{6.3.6}$$

$$produce_{gasoil} = produce_{hogasoil} + produce_{dslgasoil} \tag{6.3.7}$$

The following four constraints (6.3.8) – (6.3.11) state how the quantities of the final products are obtained from the intermediate products: butane is produced by distillation and reforming and used in the production of petrol (6.3.8). The quantities of the three other final products (petrol, diesel oil, and heating oil) result from blending as the sum of the intermediate product quantities used in their production.

$$produce_{butane} = produce_{distbutane} + produce_{refbutane} - produce_{petbutane} \tag{6.3.8}$$

$$produce_{petrol} = \sum_{p \in IPETROL} produce_p \tag{6.3.9}$$

$$produce_{diesel} = \sum_{p \in IDIESEL} produce_p \tag{6.3.10}$$

$$produce_{heating} = \sum_{p \in IHO} produce_p \tag{6.3.11}$$

Furthermore, we have a set of constraints to obey the legal restrictions on the composition of petrol and diesel oil. For instance, the octane value of petrol must be better than 94. This constraint translates to the following relation (with OCT_p the octane value of the components):

$$\frac{\sum_{p \in IPETROL} OCT_p \cdot produce_p}{produce_{petrol}} \geq 94 \tag{6.3.12}$$

This constraint is non-linear since it contains a variable in the denominator. To turn it into a linear constraints, we simply multiply everything by $produce_{petrol}$. We then obtain the following constraint (6.3.13):

$$\sum_{p \in IPETROL} OCT_p \cdot produce_p \geq 94 \cdot produce_{petrol} \tag{6.3.13}$$

The following constraints (6.3.14) – (6.3.16) establish the corresponding relations for the vapor pressure and volatility of petrol and the sulfur contents of diesel oil.

$$\sum_{p \in IPETROL} VAP_p \cdot produce_p \leq 12.7 \cdot produce_{petrol} \tag{6.3.14}$$

$$\sum_{p \in IPETROL} VOL_p \cdot produce_p \geq 17 \cdot produce_{petrol} \tag{6.3.15}$$

$$\sum_{p \in IDIESEL} SULF_p \cdot produce_p \leq 0.05 \cdot produce_{diesel} \tag{6.3.16}$$

We also need constraints that establish the limits on production capacities for reforming (6.3.17), desulfurization (6.3.18) and cracking (6.3.19). The amount use_c of every crude c that is used in production is limited by the available amount $AVAIL_c$. The constraints (6.3.21) guarantee that the demand DEM_p for all final products is satisfied. To this system we add the non-negativity condition for all variables (6.3.22) and (6.3.23).

$$produce_{np} \leq 30000 \tag{6.3.17}$$
$$produce_{gd} + produce_{gh} \leq 50000 \tag{6.3.18}$$
$$produce_{rp} + produce_{rd} + produce_{rh} \leq 40000 \tag{6.3.19}$$
$$\forall c \in CRUDES : use_c \leq AVAIL_c \tag{6.3.20}$$
$$\forall p \in FINAL : produce_p \geq DEM_p \tag{6.3.21}$$
$$\forall p \in ALLPRODS : produce_p \geq 0 \tag{6.3.22}$$
$$\forall c \in CRUDES : use_c \geq 0 \tag{6.3.23}$$

The lines (6.3.1) to (6.3.11), and (6.3.13) to (6.3.23) define the LP model.

Compared to the two other blending problems in Sections 6.1 and 6.2 the formulation of this problem looks quite 'messy': this is typical for production situations where we have a relatively small number of products going through a large number of stages that are related in complicated ways. There are many different types of constraints, and often lots of exceptions and special cases.

6.3.2 Implementation

The translation of the model in the previous section into the following formulation with Mosel is straightforward.
Contrary to the implementations of the previous examples, this time we do not define the index sets for the data arrays directly in the model but initialize the contents of the data arrays (and the corresponding index sets) dynamically from file. The only set defined directly in the model is *IHO* since it does not appear as an index set for any data array – its definition could also be read from file. The set of all products *ALLPRODS* is defined as the union of all final and intermediate products.

```
model "A-3 Refinery planning"
 uses "mmxprs"

 declarations
  CRUDES: set of string                  ! Set of crudes
  ALLPRODS: set of string                ! Intermediate and final products
  FINAL: set of string                   ! Final products
  IDIST: set of string                   ! Products obtained by distillation
  IREF: set of string                    ! Products obtained by reforming
  ICRACK: set of string                  ! Products obtained by cracking
  IPETROL: set of string                 ! Interm. products for petrol
  IDIESEL: set of string                 ! Interm. products for diesel
  IHO={"hogasoil", "hocrknaphtha", "hocrkgasoil"}
                                         ! Interm. products for heating oil
  DEM: array(FINAL) of real              ! Min. production
  COST: array(set of string) of real     ! Production costs
  AVAIL: array(CRUDES) of real           ! Crude availability
  OCT, VAP, VOL: array(IPETROL) of real  ! Octane, vapor pressure, and
                                         ! volatility values
  SULF: array(IDIESEL) of real           ! Sulfur contents
  DIST: array(CRUDES,IDIST) of real      ! Composition of crudes (in %)
  REF: array(IREF) of real               ! Results of reforming (in %)
  CRACK: array(ICRACK) of real           ! Results of cracking (in %)
 end-declarations

 initializations from 'a3refine.dat'
  DEM COST OCT VAP VOL SULF AVAIL DIST REF CRACK
 end-initializations

 finalize(FINAL); finalize(CRUDES); finalize(IPETROL); finalize(IDIESEL)
 finalize(IDIST); finalize(IREF); finalize(ICRACK)
 ALLPRODS:= FINAL+IDIST+IREF+ICRACK+IPETROL+IHO+IDIESEL

 declarations
  use: array(CRUDES) of mpvar            ! Quantities used
  produce: array(ALLPRODS) of mpvar      ! Quantities produced
```

```
 end-declarations

! Objective function
 Cost:= sum(c in CRUDES) COST(c)*use(c) + sum(p in IDIST) COST(p)*produce(p)

! Relations intermediate products resulting of distillation - raw materials
 forall(p in IDIST) produce(p) <= sum(c in CRUDES) DIST(c,p)*use(c)

! Relations between intermediate products
! Reforming:
 forall(p in IREF) produce(p) <= REF(p)*produce("naphtha")
! Cracking:
 forall(p in ICRACK) produce(p) <= CRACK(p)*produce("residue")
 produce("crknaphtha") >= produce("petcrknaphtha") +
                          produce("hocrknaphtha") + produce("dslcrknaphtha")
 produce("crkgasoil") >= produce("hocrkgasoil") + produce("dslcrkgasoil")
! Desulfurization:
 produce("gasoil") >= produce("hogasoil") + produce("dslgasoil")

! Relations final products - intermediate products
 produce("butane") = produce("distbutane") + produce("refbutane") -
                     produce("petbutane")
 produce("petrol") = sum(p in IPETROL) produce(p)
 produce("diesel") = sum(p in IDIESEL) produce(p)
 produce("heating") = sum(p in IHO) produce(p)

! Properties of petrol
 sum(p in IPETROL) OCT(p)*produce(p) >= 94*produce("petrol")
 sum(p in IPETROL) VAP(p)*produce(p) <= 12.7*produce("petrol")
 sum(p in IPETROL) VOL(p)*produce(p) >= 17*produce("petrol")

! Limit on sulfur in diesel oil
 sum(p in IDIESEL) SULF(p)*produce(p) <= 0.05*produce("diesel")

! Crude availabilities
 forall(c in CRUDES) use(c) <= AVAIL(c)

! Production capacities
 produce("naphtha") <= 30000               ! Reformer
 produce("gasoil") <= 50000                ! Desulfurization
 produce("residue") <= 40000               ! Cracker

! Satisfy demands
 forall(p in FINAL) produce(p) >= DEM(p)

! Solve the problem
 minimize(Cost)

end-model
```

The dynamic initialization of the index sets necessitates a special treatment of the arrays of variables *use* and *produce*.
If we declared these arrays at the same time as the data arrays, this would result in empty, dynamic arrays since the index sets are not known before the data has been read from file. Whilst the entries of dynamic arrays of types other than `mpvar` are created when the entry is addressed, this is not the case for arrays of variables. There are two possibilities for handling this situation:

- In the implementation shown above we define the variables in a separate `declarations` block after the data has been read from file. This block is preceded by a series of `finalize` statements. With the help of the `finalize` procedure, the (dynamic) index sets are turned into constant sets, that is, their contents cannot be modified any more. As a consequence, all arrays with these sets as their indices and that are declared after this point are created as arrays of fixed size.
- Alternatively, we may declare the array of type `mpvar` together with the data arrays and create the required variables explicitly by adding the following lines after the data has been read from file and the two sets *ALLPROD* and *CRUDES* have been finalized:

```
forall(p in ALLPROD) create(produce(p))
forall(c in CRUDES) create(use(c))
```

In larger applications, typically most data (and the corresponding index sets) are initialized dynamically so that the variable definition needs to be done with one of the two methods described above.

To keep the example implementations in this book as easy as possible, we usually define the index sets as constants directly in the model so that the definition of variables does not require any special care.

6.3.3 Results

The total production cost is € 1,175,400. We produce 40,000 tonnes of petrol from 6,500 tonnes of butane, 25,500 tonnes of reformate, and 8,000 tonnes of cracked naphtha. With this composition, we obtain an octane value of 98.05, the vapor pressure is 12.23, and the volatility of the petrol is 21.38. The required 30,000 tonnes of diesel oil are produced with 30,000 tonnes of sweetened gasoil, which gives a sulfur content of 0.03%. The 42,000 tonnes of heating oil are composed of 20,000 tonnes of sweetened gasoil, 8,000 tonnes of cracked naphtha, and 14,000 tonnes of cracked gasoil. In addition to this production we also have the required 20,000 tonnes of butane.

6.4 Cane sugar production

The harvest of cane sugar in Australia is highly mechanized. The sugar cane is immediately transported to a sugar house in wagons that run on a network of small rail tracks. The sugar content of a wagon load 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 wagon load will entirely lose its value after a certain time. At this moment, eleven wagons all loaded with the same quantity have arrived at the sugar house. 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 6.8: 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	6	8	8

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

6.4.1 Model formulation

This problem also concerns the primary sector but it is different from the blending problems we have seen so far in this chapter.

Let $WAGONS = \{1, \ldots, NW\}$ 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 integer. The maximum value of s is the number of time slots (of length DUR) that the sugar house will work, namely $NS = ceil(NW / NL)$, where $ceil$ stands for 'rounded to the next larger 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 $SLOTS = \{1, \ldots, NS\}$ the set of time slots.

Every lot needs to be assigned to a time slot. We define binary variables $process_{ws}$ that take the value 1 if and only if lot w is assigned to slot s (constraints (6.4.1)). Every lot needs to be assigned to a slot (6.4.2), and any slot may take up to NL lots because there are NL parallel lines (6.4.3).

$$\forall w \in WAGONS, s \in SLOTS : process_{ws} \in \{0, 1\} \tag{6.4.1}$$

$$\forall w \in WAGONS : \sum_{s \in SLOTS} process_{ws} = 1 \tag{6.4.2}$$

$$\forall s \in SLOTS : \sum_{w \in WAGONS} process_{ws} \leq NL \tag{6.4.3}$$

The life span of the sugar lots is given in hours and not in periods of DUR hours. The maximum slot number for a wagon load w is therefore $LIFE_w / DUR$. The constraints (6.4.4) establish this bound for

every lot w. Note that the number of the time slot is expressed through a sum of decision variables: due to the constraints (6.4.2) only the index of the slot s for which $process_{ws} = 1$ will be counted for lot w.

$$\forall w \in WAGONS: \sum_{s \in SLOTS} s \cdot process_{ws} \leq LIFE_w \;/\; DUR \tag{6.4.4}$$

The loss of sugar per wagon load w and time slot s is $s \cdot DUR \cdot LOSS_w$. The objective function (total loss of sugar) is therefore given by the following expression (6.4.5):

$$\text{minimize} \sum_{w \in WAGONS} \sum_{s \in SLOTS} s \cdot DUR \cdot LOSS_w \cdot process_{ws} \tag{6.4.5}$$

The lines (6.4.1) to (6.4.5) define the model for our problem. This linear problem is a **minimum cost flow problem** of a particular type. It is related to the **transportation problems** (like the car rental problem in Chapter 10). For minimum cost flow problems, the optimal LP solution calculated by the simplex algorithm is integer feasible. We could therefore replace the integrality constraints in our model by non-negativity constraints.

6.4.2 Implementation

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

```
model "A-4 Cane sugar production"
 uses "mmxprs"

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

  LOSS: array(WAGONS) of real      ! Loss in kg/hour
  LIFE: array(WAGONS) of real      ! Remaining time per lot
  DUR: integer                     ! Duration of the production

  process: array(WAGONS,SLOTS) of mpvar  ! 1 if wagon processed in slot,
                                         ! 0 otherwise
 end-declarations

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

! Objective function
 TotalLoss:= sum(w in WAGONS, s in SLOTS) s*DUR*LOSS(w)*process(w,s)

! Assignment
 forall(w in WAGONS) sum(s in SLOTS) process(w,s) = 1

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

! Limit on raw product life
 forall(w in WAGONS) sum(s in SLOTS) s*process(w,s) <= LIFE(w)/DUR

 forall(w in WAGONS, s in SLOTS) process(w,s) is_binary

! Solve the problem
 minimize(TotalLoss)
```

6.4.3 Results

If we solve the problem without the constraints limiting the life span of the wagon loads we obtain a minimum loss of 1518 kg of sugar. In this case the lots are processed in the order of decreasing loss. With the complete set of constraints we obtain a total loss of 1620 kg of sugar. The corresponding schedule of lots is shown in the following table (there are several equivalent solutions).

Table 6.9: Optimal schedule for the cane sugar lots

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

6.5 Opencast mining

An opencast uranium mine is being prospected. Based on the results of some test drillings the mine has been subdivided into exploitation units called **blocks**. The pit needs to be terraced to allow the trucks to drive down to its bottom. The uranium deposit extends from east to west. The pit is limited in the west by a village and in the east by a group of mountains. Taking into account these constraints, 18 blocks of 10,000 tonnes on three levels have been identified (Figure 6.3). To extract a block, three blocks of the level above it need to be extracted: the block immediately on top of it, and also, due to the constraints on the slope, the blocks to the right and to the left.

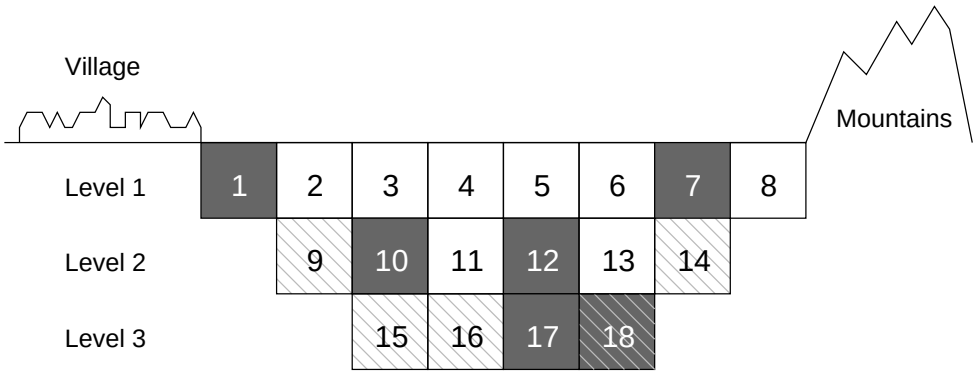

Figure 6.3: Length cut through the opencast mine

It costs € 100 per tonne to extract a block of level 1, € 200 per tonne for a block of level 2, and € 300 per tonne for a block of level 3, with the exception of the hatched blocks that are formed of a very hard rock rich in quartz and cost € 1000 per ton. The only blocks that contain uranium are those displayed in a gray shade (1, 7, 10, 12, 17, 18). Their market value is 200, 300, 500, 200, 1000, and € 1200/tonne respectively. Block 18, although rich in ore, is made of the same hard rock as the other hatched blocks. Which blocks should be extracted to maximize the total benefit?

6.5.1 Model formulation

Let $BLOCKS$ be the set of blocks, $VALUE_b$ the value per tonne of a block b, and $COST_b$ the cost per tonne of extracting it. The benefit of extracting block b per tonne is then given by $VALUE_b - COST_b$. The possible extraction sequences of blocks can be represented by a directed graph $G = (BLOCKS, ARCS)$, where $ARCS$ stands for the set of arcs between blocks. An arc (b, a) means that block b can only be extracted if block a has already been taken. For instance, block 16 gives rise to three arcs in G: (16,10), (16,11), and (16,12) and blocks 10, 11, and 12 again to three arcs each, as shown in the following graph.

To decide which blocks are extracted we introduce binary variables $extract_b$ that take the value 1 if and only if block b is extracted. We obtain the following, very compact problem with 0-1 variables:

$$\text{maximize} \sum_{b \in BLOCKS} (VALUE_b - COST_b) \cdot extract_b \tag{6.5.1}$$

$$\forall (b, a) \in ARCS : extract_b \leq extract_a \tag{6.5.2}$$

$$\forall b \in BLOCKS : extract_b \in \{0, 1\} \tag{6.5.3}$$

The objective function (6.5.1) that is to be maximized is the sum of the benefits from the blocks that are extracted. The constraints (6.5.2) make sure that the blocks are extracted in the right order: if b is extracted ($extract_b = 1$) then a also needs to be extracted ($extract_a = 1$). The constraints (6.5.3) define the binary variables.

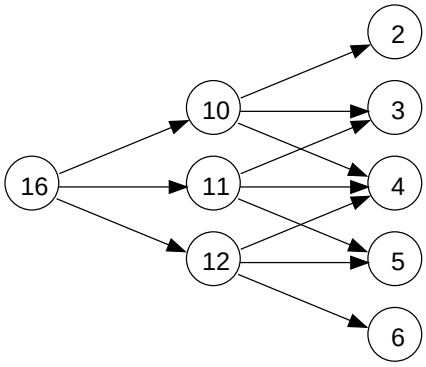

Figure 6.4: Precedence graph for extraction of block 16

It would be possible to replace the constraints (6.5.2) by their aggregate form (6.5.4). In large problems that also include many other variables and constraints this formulation may help reduce the total number of constraints. This reformulation provides a weaker, *i.e.* relaxed, formulation than the original constraints. In our example this has the disadvantage that whilst for the original formulation the solution of the linear relaxation is already integer, this is no longer true for the relaxed formulation.

$$\forall b \in BLOCKS, \exists (b,a) \in ARCS : 3 \cdot extract_b \leq \sum_{\substack{a \in BLOCKS \\ \exists (b,a) \in ARCS}} extract_a \qquad (6.5.4)$$

6.5.2 Implementation

The following model implements the LP of the previous section with Mosel. The graph is coded as a two-dimensional array that for every block of levels 2 and 3 contains its three predecessors in the level above it. For instance for block 16, it contains the line `(16 1) 10 11 12` which is the short form of `(16 1) 10 (16 2) 11 (16 3) 12`.

```
model "A-5 Opencast mining"

 uses "mmxprs"

 declarations
  BLOCKS = 1..18                      ! Set of blocks
  LEVEL23: set of integer             ! Blocks in levels 2 and 3
  COST: array(BLOCKS) of real         ! Exploitation cost of blocks
  VALUE: array(BLOCKS) of real        ! Value of blocks
  ARC: array(LEVEL23,1..3) of integer ! Arcs indicating order of extraction

  extract: array(BLOCKS) of mpvar     ! 1 if block b is extracted
 end-declarations

 initializations from 'a5mine.dat'
  COST VALUE ARC
 end-initializations

! Objective: maximize total profit
 Profit:= sum(b in BLOCKS) (VALUE(b)-COST(b))* extract(b)

! Extraction order
 forall(b in LEVEL23)
  forall(i in 1..3) extract(b) <= extract(ARC(b,i))

 forall(b in BLOCKS) extract(b) is_binary

! Solve the problem
 maximize(Profit)

end-model
```

As mentioned earlier, for the model formulation we are using in the implementation the solution to the linear problem is integer. As a general rule, it may be helpful not to declare any special types of

variables at the first attempt at solving a problem: the problem studied may have a specific structure for which the integrality constraints are redundant (this is the case for the present example) or for a certain data set all variables may be integral in the optimal LP solution. Switching integrality conditions off without removing them from the model can also be obtained by adding a parameter to the call to the optimization algorithm. For instance in this example

```
maximize(XPRS_LIN,Profit)
```

Another reason not to establish immediately all integrality constraints may be size limitations in the optimization software that is used (the student version of Xpress-Optimizer is limited to 800 variables + constraints out of which at most 500 may be MIP variables or SOS sets). A problem that defines too many integer variables cannot be executed, even if the optimal integer solution could be obtained by solving the LP.

6.5.3 Results

The maximal profit is € 4,000,000. Blocks 1 to 7, 10 to 13, and 17 are extracted. Blocks 8, 9, 14 to 16, and 18 remain in the pit.

6.6 Production of electricity

Power generators of four different types are available to satisfy the daily electricity demands (in megawatts) summarized in the following table. We consider a sliding time horizon: the period 10pm-12am of day d is followed by the period 0am-6am of day $d + 1$.

Table 6.10: Daily electricity demands (in MW)

Period	0am–6am	6am–9am	9am–12pm	12pm–2pm	2pm–6pm	6pm–10pm	10pm–12am
Demand	12000	32000	25000	36000	25000	30000	18000

The power generators of the same type have a maximum capacity and may be connected to the network starting from a certain minimal power output. They have a start-up cost, a fixed hourly cost for working at minimal power, and an hourly cost per additional megawatt for anything beyond the minimal output. These data are given in the following table.

Table 6.11: Description of power generators

	Available number	Min. output in MW	Max. capacity in MW	Fix cost €/h	Add. MW cost €/h	Start-up cost
Type 1	10	750	1750	2250	2.7	5000
Type 2	4	1000	1500	1800	2.2	1600
Type 3	8	1200	2000	3750	1.8	2400
Type 4	3	1800	3500	4800	3.8	1200

A power generator can only be started or stopped at the beginning of a time period. As opposed to the start, stopping a power plant does not cost anything. At any moment, the working power generators must be able to cope with an increase by 20% of the demand forecast. Which power generators should be used in every period in order to minimize the total daily cost?

6.6.1 Model formulation

Let $TIME = \{1, \ldots, NT\}$ be the set of time periods per day and $TYPES$ the set of generator types. For a given time period t, DEM_t denotes the electricity demand in the network and LEN_t the length (number of hours) of the period. For any power generator type p, $PMIN_p$ and $PMAX_p$ are the minimum and maximum capacity respectively, and $AVAIL_p$ is the available number of generators. The start-up cost is $CSTART_p$, the hourly cost for working at minimum level is $CMIN_p$, and the cost per additional MW hour is $CADD_p$.

To formulate this model and account for the different types of cost we need three sets of variables. $start_{pt}$ is the integer number of power generators of type p that start working at the beginning of time period t (6.6.1). $work_{pt}$ is the number of power generators of type p that are working in time period t. This integer number is bounded by the available number of generators $AVAIL_p$, constraints (6.6.2). $padd_{pt}$ represents the additional production beyond the minimum output level of generators of type p in time

period t.

$$\forall p \in TYPES, t \in TIME : start_{pt} \in \mathbb{N} \tag{6.6.1}$$
$$\forall p \in TYPES, t \in TIME : work_{pt} \in \{0, 1, 2, \ldots, AVAIL_p\} \tag{6.6.2}$$
$$\forall p \in TYPES, t \in TIME : padd_{pt} \geq 0 \tag{6.6.3}$$

For a power generator of type p the additional production is limited by $PMAX_p - PMIN_p$. For every type and time period we can therefore link the additional output with the number of working generators:

$$\forall p \in TYPES, t \in TIME : padd_{pt} \leq (PMAX_p - PMIN_p) \cdot work_{pt} \tag{6.6.4}$$

To satisfy the demand in every period we write the constraints (6.6.5). The first sum is the total basis output by the generators of different types. The second sum is the total additional output beyond the minimum level by all power generators.

$$\forall t \in TIME : \sum_{p \in TYPES} \left(PMIN_p \cdot work_{pt} + padd_{pt}\right) \geq DEM_t \tag{6.6.5}$$

Without starting any new generators, those that are working must be able to produce 20% more than planned:

$$\forall t \in TIME : \sum_{p \in TYPES} PMAX_p \cdot work_{pt} \geq 1.20 \cdot DEM_t \tag{6.6.6}$$

If no generators were stopped, the number of power generators starting to work in a period t would correspond to the difference between the numbers of generators working in periods t and $t-1$. Since generators may be stopped (not explicitly counted as there is no impact on the cost), the number of generators starting to work is at least the value of this difference (6.6.7). The constraints (6.6.8) are for the particular case of the transition from the last period of a day to the first period of the following day.

$$\forall p \in TYPES, t \in \{2, \ldots, NT\} : start_{pt} \geq work_{pt} - work_{p,t-1} \tag{6.6.7}$$
$$\forall p \in TYPES : start_{p1} \geq work_{p1} - work_{p,NT} \tag{6.6.8}$$

The daily cost (6.6.9) comprises the start-up cost, the cost for working at the minimum level and the cost for any additional production. These last two costs are proportional to the length of the time periods.

$$\text{minimize} \sum_{p \in TYPES} \sum_{t \in TIME} \left(CSTART_p \cdot start_{pt} + LEN_t \cdot \left(CMIN_p \cdot work_{pt} + CADD_p \cdot padd_{pt}\right)\right) \tag{6.6.9}$$

The lines (6.6.1) to (6.6.9) form the final mixed integer model.

6.6.2 Implementation

The algebraic model translates into the following Mosel model. The constraints (6.6.7) and (6.6.8) are obtained with a single line, using the inline `if` function of Mosel.
Note further that we do not define variables *start* as integer since they result from a difference of integer variables *work* through constraints (6.6.7) and therefore automatically take integer values.

```
model "A-6 Electricity production"
 uses "mmxprs"

 declarations
  NT = 7
  TIME = 1..NT                        ! Time periods
  TYPES = 1..4                        ! Power generator types

  LEN, DEM: array(TIME) of integer    ! Length and demand of time periods
  PMIN,PMAX: array(TYPES) of integer  ! Min. & max output of a generator type
  CSTART: array(TYPES) of integer     ! Start-up cost of a generator
  CMIN: array(TYPES) of integer       ! Hourly cost of gen. at min. output
  CADD: array(TYPES) of real          ! Cost/hour/MW of prod. above min. level
  AVAIL: array(TYPES) of integer      ! Number of generators per type

  start: array(TYPES,TIME) of mpvar   ! No. of gen.s started in a period
  work: array(TYPES,TIME) of mpvar    ! No. of gen.s working during a period
  padd: array(TYPES,TIME) of mpvar    ! Production above min. output level
```

```
 end-declarations

 initializations from 'a6electr.dat'
  LEN DEM PMIN PMAX CSTART CMIN CADD AVAIL
 end-initializations

! Objective function: total daily cost
 Cost:= sum(p in TYPES, t in TIME) (CSTART(p)*start(p,t) +
          LEN(t)*(CMIN(p)*work(p,t) + CADD(p)*padd(p,t)))

! Number of generators started per period and per type
 forall(p in TYPES, t in TIME)
  start(p,t) >= work(p,t) - if(t>1, work(p,t-1), work(p,NT))

! Limit on power production above minimum level
 forall(p in TYPES, t in TIME) padd(p,t) <= (PMAX(p)-PMIN(p))*work(p,t)

! Satisfy demands
 forall(t in TIME) sum(p in TYPES) (PMIN(p)*work(p,t) + padd(p,t)) >= DEM(t)

! Security reserve of 20%
 forall(t in TIME) sum(p in TYPES) PMAX(p)*work(p,t) >= 1.20*DEM(t)

! Limit number of available generators; numbers of generators are integer
 forall(p in TYPES, t in TIME) do
  work(p,t) <= AVAIL(p)
  work(p,t) is_integer
 end-do

! Solve the problem
 minimize(Cost)

end-model
```

6.6.3 Results

The optimal LP solution to this problem is a cost of € 985164.3. The optimal integer solution has a cost of € 1,465,810. For every time period and generator type the following table lists the total output and the part of the production that is above the minimum level.

Table 6.12: Plan of power generator use

Type		0am–6am	6am–9am	9am–12pm	12pm–2pm	2pm–6pm	6pm–10pm	10pm–12am
1	No. used	3	4	4	7	3	3	3
	Tot. output	2250	4600	3000	8600	2250	2600	2250
	Add. output	0	1600	0	3350	0	350	0
2	No. used	4	4	4	4	4	4	4
	Tot. output	5750	6000	4200	6000	4950	6000	5950
	Add. output	1750	2000	200	2000	950	2000	1950
3	No. used	2	8	8	8	8	8	4
	Tot. output	4800	16000	16000	16000	16000	16000	8000
	Add. output	1600	6400	6400	6400	6400	6400	3200
4	No. used	0	3	1	3	1	3	1
	Tot. output	0	5400	1800	5400	1800	5400	1800
	Add. output	0	0	0	0	0	0	0

By looking at this table we may deduce when power generators are started and stopped: one generator of type 1 starts at the beginning of period 2, and three at the beginning of period 4, four generators stop after period 4. The four generators of type 2 work continuously. Six generators of type 3 start working in period 2, four generators of this type are stopped after period 6 and two more at the end of every day. All three generators of type 4 start at the beginning of period 2, two of which are switched off in the next period, then switched on, switched off, and switched on again and at the end of every day all three are stopped.

6.7 References and further material

Blending problems are historically the first civil applications of Linear Programming. References for the first three problems in this chapter: Glen describes a problem of mixing food for cattle [Gle80], Sutton and Coates deal with blending in the iron and steel industry [SC81], and McColl presents applications of LP in the petrol industry [McC69].

The problem of the sugar house is a member of the big family of flow problems, dealt with in detail in the book by Ahuja, Magnanti and Orlin [AMO93] which is a reference for this topic. The opencast mining problem boils down to calculating the maximum weight closure of a graph, a problem that can equally be solved by a flow algorithm (Chapter 19 of the book by Ahuja et al.). An example where blocks are arranged in a three-dimensional way is described by Williams [Wil93].

The electricity generation problem has been studied by Garver [Gar63]. In this family of dispatching problems, the aim is to satisfy at the least cost the demand of each period, with costs for activating and deactivating resources (see for example [Gar00]). The main difference from the production planning problems in Chapter 8 lies in the costs and the fact that the power plants cannot be used below a certain minimum output level. The variation in power supply may even be very narrow for water barrages which usually leads to combining them with thermal or nuclear power plants to follow the demand more easily [SM74].

Chapter 7
Scheduling problems

Scheduling problems are an important class of problems in Operations Research. They consist of distributing the execution of a set of tasks over time subject to various constraints (sequence of tasks, due dates, limited resource availability), with the objective to optimize a criterion like the total duration, the number of tasks that finish late etc. These problems may come from many different fields: project management, industrial production, telecommunications, information systems, transport, timetabling. With the exception of the well known case of project scheduling, these problems are difficult to solve to optimality if they grow large. Nevertheless, instances of smaller size have become solvable by LP, due to the power of current software.

This chapter only deals with project and production scheduling problems. The Section 7.1 concerns the construction of a stadium, a typical case of project scheduling in civil engineering. The next three problems are about the management of workshops: Section 7.2 presents a workshop organized in lines (flow-shop), that all products are run through in the same order. In the job-shop of Section 7.3 every product has a different processing order on the machines of the workshop. Section 7.4 deals with the case of a critical machine, the bottleneck of a workshop. The problem in Section 7.5 consists of distributing the workload of an amplifier assembly line to maximize the throughput.

This book also discusses scheduling problems in other domains: in air transport (Chapter 11), telecommunications (Chapter 12) and for establishing a timetable (Chapter 14).

7.1 Construction of a stadium

A town council wishes to construct a small stadium in order to improve the services provided to the people living in the district. After the invitation to tender, a local construction company is awarded the contract and wishes to complete the task within the shortest possible time. All the major tasks are listed in the following table. The durations are expressed in weeks. Some tasks can only start after the completion of certain other tasks. The last two columns of the table refer to question 2 which we shall see later.

Question 1: Which is the earliest possible date of completing the construction?

Question 2: The town council would like the project to terminate earlier than the time announced by the builder (answer to question 1). To obtain this, the council is prepared to pay a bonus of € 30K for every week the work finishes early. The builder needs to employ additional workers and rent more equipment to cut down on the total time. In the preceding table he has summarized the maximum number of weeks he can save per task (column "Max. reduct.") and the associated additional cost per week. When will the project be completed if the builder wishes to maximize his profit?

7.1.1 Model formulation for question 1

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 $TASKS = \{1, \ldots, N\}$ where N is the fictitious end task. Let DUR_i be the duration of task i. To establish the precedences between tasks, we use a precedence graph $G = (TASKS, ARCS)$ where $ARCS$ stands for the set of arcs (an arc (i, j) symbolizes that task i precedes task j). It is easy to construct such a graph starting from the lists of predecessors in Table 7.1. The fictitious task follows all tasks that have no successor.

We introduce variables $start_i$ to represent the earliest start time of tasks i. The only constraints that are given are the precedences. A task j may only start if all its predecessors have finished, which translates

Table 7.1: Data for stadium construction

Task	Description	Duration	Predecessors	Max. reduct.	Add. cost per week (in 1000 €)
1	Installing the construction site	2	none	0	–
2	Terracing	16	1	3	30
3	Constructing the foundations	9	2	1	26
4	Access roads and other networks	8	2	2	12
5	Erecting the basement	10	3	2	17
6	Main floor	6	4,5	1	15
7	Dividing up the changing rooms	2	4	1	8
8	Electrifying the terraces	2	6	0	–
9	Constructing the roof	9	4,6	2	42
10	Lighting of the stadium	5	4	1	21
11	Installing the terraces	3	6	1	18
12	Sealing the roof	2	9	0	–
13	Finishing the changing rooms	1	7	0	–
14	Constructing the ticket office	7	2	2	22
15	Secondary access roads	4	4,14	2	12
16	Means of signalling	3	8,11,14	1	6
17	Lawn and sport accessories	9	12	3	16
18	Handing over the building	1	17	0	–

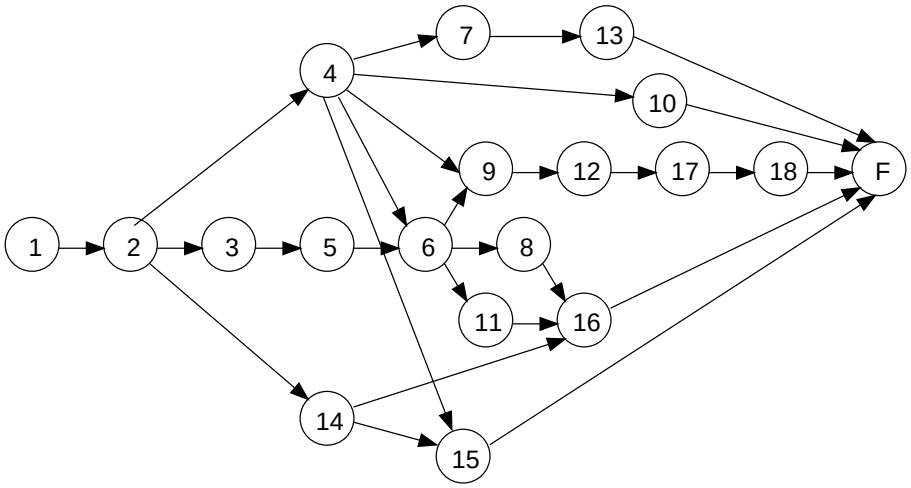

Figure 7.1: Precedence graph of construction tasks

into constraints (7.1.1): if there is an arc between i and j, then the completion time of i ($start_i + DUR_i$) must not be larger than the start time of j.

$$\forall (i,j) \in ARCS : start_i + DUR_i \leq start_j \tag{7.1.1}$$

The objective is to minimize the completion time of the project, that is the start time of the last, fictitious task N. We complete the mathematical model with the following lines:

$$\text{minimize } start_N \tag{7.1.2}$$

$$\forall i \in TASKS : start_i \geq 0 \tag{7.1.3}$$

7.1.2 Implementation of question 1

The translation of the mathematical model into Mosel is straightforward. The set of arcs is implemented as a two-dimensional binary matrix ARC, an element $ARC_{ij} = 1$ if and only if the arc (i,j) exists. At its definition, we do not fix the index sets of this array and use the keyword `range` instead, which results in the definition of a dynamic array for which only those entries exist that are read in from the data file. The latter only contains the list of defined arcs (using an array format that is usually referred to as **sparse format** as opposed to the **dense format** where all entries must be defined). Note that the objective function of this problem is simply a single variable (the fictitious end task).

```
model "B-1 Stadium construction (First part)"
 uses "mmxprs"

 declarations
  N = 19                              ! Number of tasks in the project
                                      ! (last = fictitious end task)
  TASKS=1..N
  ARC: array(range,range) of real     ! Matrix of the adjacency graph
  DUR: array(TASKS) of real           ! Duration of tasks
  start: array(TASKS) of mpvar        ! Start times of the tasks
 end-declarations

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

! Precedence relations between tasks
 forall(i,j in TASKS | ARC(i,j)=1) start(i) + DUR(i) <= start(j)

! Solve the first problem: minimize the total duration
 minimize(start(N))

end-model
```

7.1.3 Results for question 1

The execution of the program gives us a duration of 64 weeks to complete the construction of the stadium, which correspond to 1 year and 3 months. A possible solution is represented in Figure 7.2 (there are several possible plans, since for instance task 10 may start any time in the interval [26,59] without having any effect on the total completion time). In this bar chart every task is represented by a rectangle the length of which is proportional to the duration of the task. The vertical position of the tasks has no signification. The shaded areas in the rectangles representing the tasks correspond to the maximum time by which the task may be shortened (see question 2).

7.1.4 Model formulation for question 2

This second problem is called **scheduling with project crashing**. To reduce the total duration of the project, we need to take into account the result of the preceding optimization run. We define variables $save_i$ that correspond to the number of weeks that we wish to save for every task i. The column 'Max. reduct.' in Table 7.1 gives an upper bound on these variables $save_i$. We call $MAXW_i$ this maximum reduction (in weeks). The constraints (7.1.4) must be satisfied for all tasks i except the last, fictitious task N.

$$\forall i \in TASKS \backslash \{N\} : save_i \leq MAXW_i \tag{7.1.4}$$

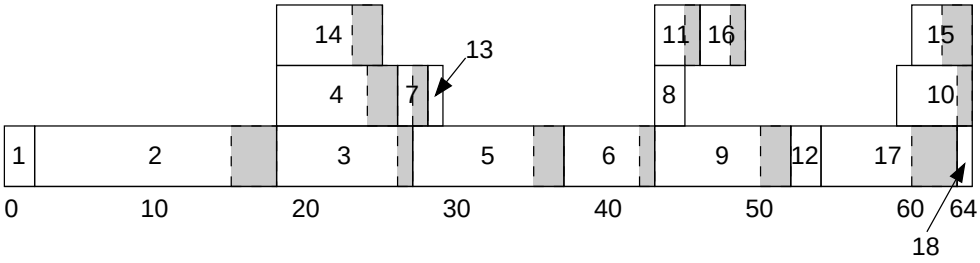

Figure 7.2: Solution to question 1

For the last task, the variable $save_N$ represents the number of weeks the project finishes earlier than the solution obj_1 calculated in answer to question 1. The new completion time of the project $start_N$ must be equal to the previous completion time minus the advance $save_N$, which leads to the constraint (7.1.5).

$$start_N = obj_1 - save_N \tag{7.1.5}$$

The constraints (7.1.1) need to be modified to take into account the new variables $save_i$. The new completion time of a task is equal to its start, plus its duration, minus the savings, or $start_i + DUR_i - save_i$. We rename these constraints to (7.1.6).

$$\forall (i,j) \in ARCS : start_i + DUR_i - save_i \leq start_j \tag{7.1.6}$$

The objective has also been redefined by the second question. We now want to maximize the builder's profit. For every week finished early, he receives a bonus of BONUS k€. In exchange, the savings in time for a task i costs $COST_i$ k€ (column 'Add. cost per week' of Table 7.1). The new objective function will be labelled (7.1.7).

$$\text{maximize } BONUS \cdot save_N - \sum_{i \in TASKS \setminus \{N\}} COST_i \cdot save_i \tag{7.1.7}$$

The new mathematical model consists of relations (7.1.6), (7.1.7), (7.1.3), (7.1.4), and (7.1.5), and the non-negativity conditions for variables $save_i$.

7.1.5 Implementation of question 2

It is possible to implement the model for question 2 as an addition to the model for question 1 shown above, all in a single Mosel program. This makes it possible to retrieve the solution value of the first problem and use it in the definition of the second problem without the need for any interaction.

Since we need to make some additions to the first model, we repeat the complete example. The additions to the previous implementation of the first part are the following:

- A solution printing subroutine: we want to print the solution to the problem twice (that is, the start times assigned to tasks). To avoid having to repeat the same lines of code several times we define a **subroutine** that is called whenever we want to print a solution. The third line of the model now contains the declaration of the procedure `print_sol` that is defined at the end of the program. This declaration (using the keyword `forward`) is required if a user-defined subroutine is called before it is defined. A subroutine has a similar structure to a `model` in Mosel; we shall see more examples of subroutines in the following chapters.
- The precedence constraints are **named**: Mosel defines models incrementally. If we simply add the new definition of the precedence constraints in the second part of the model, we also keep the previous definition of these constraints. By naming the constraints and re-assigning them we override the first definition.
- A real number `obj1` is used to store the solution value of the first problem.

After solving the second problem, we print the solution value of this problem using the function `getsol` (we could equally have used `getobjval`, this is just to make clear which is the constraint that defines the current objective function). If it is applied to a constraint, the function `getsol` returns its evaluation with the current solution (note that a constraint is stored in the form *variableterms* − *RHSvalue* which is the expression that is evaluated by the function).

```
model "B-1 Stadium construction (Complete model)"
 uses "mmxprs"

 forward procedure print_sol

 declarations
  N = 19                               ! Number of tasks in the project
                                       ! (last = fictitious end task)
  TASKS=1..N
  ARC: array(range,range) of real      ! Matrix of the adjacency graph
  DUR: array(TASKS) of real            ! Duration of tasks
  start: array(TASKS) of mpvar         ! Start times of tasks
  obj1: real                           ! Solution of the first problem
 end-declarations

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

! Precedence relations between tasks
 forall(i,j in TASKS | ARC(i,j)=1)
  Prec(i,j):= start(i) + DUR(i) <= start(j)

! Solve the first problem: minimize the total duration
 minimize(start(N))
 obj1:=getobjval

! Solution printing
 print_sol

! **** Extend the problem ****

 declarations
  BONUS: integer                       ! Bonus per week finished earlier
  MAXW: array(TASKS) of real           ! Max. reduction of tasks (in weeks)
  COST: array(TASKS) of real           ! Cost of reducing tasks by a week
  save: array(TASKS) of mpvar          ! Number of weeks finished early
 end-declarations

 initializations from 'b1stadium.dat'
  MAXW BONUS COST
 end-initializations

! Second objective function
 Profit:= BONUS*save(N) - sum(i in 1..N-1) COST(i)*save(i)

! Redefine precedence relations between tasks
 forall(i,j in TASKS | ARC(i,j)=1)
  Prec(i,j):= start(j) - start(i) + save(i) >= DUR(i)

! Total duration
 start(N) + save(N) = obj1

! Limit on number of weeks that may be saved
 forall(i in 1..N-1) save(i) <= MAXW(i)

! Solve the second problem: maximize the total profit
 maximize(Profit)

! Solution printing
 writeln("Total profit: ", getsol(Profit))
 print_sol

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

 procedure print_sol
  writeln("Total duration: ", getsol(start(N)), " weeks")
  forall(i in 1..N-1)
   write(strfmt(i,2), ": ", strfmt(getsol(start(i)),-3),
    if(i mod 9 = 0,"\n",""))
  writeln
 end-procedure

end-model
```

7.1.6 Results for question 2

The solution of the complete problem shows that it is possible to complete the project 10 weeks earlier with a total profit of €87k for the builder. The new start times are shown in the schedule that is represented below (note that this is again one of several possible plans).

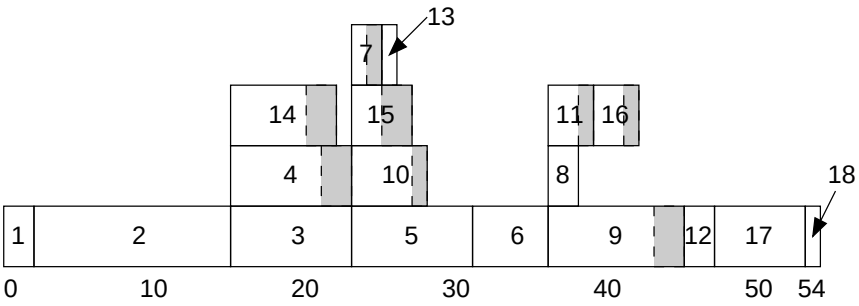

Figure 7.3: Solution to question 2

7.2 Flow-shop scheduling

A workshop that produces metal pipes on demand for the automobile industry has three machines for bending the pipes, soldering the fastenings, and assembling the links. The workshop has to produce six pieces, for which the durations of the processing steps (in minutes) are given in the following table. Every workpiece first goes to bending, then to soldering, and finally to assembly of the links. Once started, any operations must be carried out without interruption, but the workpieces may wait between the machines.

Table 7.2: Processing durations in minutes

Workpiece	1	2	3	4	5	6
Bending	3	6	3	5	5	7
Soldering	5	4	2	4	4	5
Assembly	5	2	4	6	3	6

Every machine only processes one piece at a time. A workpiece may not overtake any other by passing onto the following machine. This means that if at the beginning a sequence of the workpieces is established, they will be processed on every machine in exactly this order. Which is the sequence of workpieces that minimizes the total time for completing all pieces?

7.2.1 Model formulation

From now on, we work with $MACH = \{1, \ldots, NM\}$ the set of machines and $JOBS = \{1, \ldots, NJ\}$ the set of pieces (jobs) to produce. The processing duration of a piece j on a machine m is given by $DUR_{m,j}$. Every workpiece has to run through the machines $1, \ldots, NM$ in exactly this order, without being able to overtake other pieces. We can therefore define a schedule by the initial order of the workpieces. The duration of the schedule is given as the time instant when the machine NM finishes the last job.

The sequence of the jobs can be defined with the help of binary variables $rank_{jk}$ that are 1 if and only if piece j has the rank (position) k in the starting sequence (7.2.1). The set of starting positions $RANKS$ is the same as the set of jobs $JOBS$ since every job needs to be assigned a rank (7.2.2), and every rank must be occupied by one job only (7.2.3). These constraints are typical of assignment problems (see also the assignment of personnel to workstations in Chapter 14).

$$\forall j \in JOBS, k \in RANKS : rank_{jk} \in \{0, 1\} \tag{7.2.1}$$

$$\forall k \in RANKS : \sum_{j \in JOBS} rank_{jk} = 1 \tag{7.2.2}$$

$$\forall j \in JOBS : \sum_{k \in RANKS} rank_{jk} = 1 \tag{7.2.3}$$

In this problem it is relatively difficult to calculate the starting or completion times of the operations from the ranks. To obtain these values, we introduce two additional sets of variables $empty_{mk}$ and $wait_{mk}$ (7.2.4) – (7.2.5). The variables $empty_{mk}$ (with m in $MACH$ and k in $1, \ldots, NJ-1$) denote the time between the processing of the jobs with rank k and $k+1$ on a machine m, *i.e.* the time that the machine m is idle after the termination of the workpiece with rank k. A variable $wait_{mk}$ (with m in $1, \ldots, NM$ and k in $RANKS$) is the waiting time for the job with rank k between its processing on machines m and $m+1$.

The workpieces can be processed without any pause on the first machine that does not have to wait for any preceding machines, so the variables $empty_{1k}$ are therefore always 0 (7.2.6). Similarly, the first workpiece in the sequence can pass through all machines without any waiting times, which means the variables $wait_{m1}$ can also be fixed to 0 (7.2.7).

$$\forall m \in MACH, k = 1, \ldots, NJ-1 : empty_{mk} \geq 0 \tag{7.2.4}$$

$$\forall m = 1, \ldots, NM-1, k \in RANKS : wait_{mk} \geq 0 \tag{7.2.5}$$

$$\forall k = 1, \ldots, NJ-1 : empty_{1k} = 0 \tag{7.2.6}$$

$$\forall m = 1, \ldots, NM-1 : wait_{m1} = 0 \tag{7.2.7}$$

To simplify the formulation of the following constraints, we introduce the notation dur_{mk} for the processing duration of the job of rank k on machine m, defined by the relations (7.2.8). Through the constraints (7.2.3) only a single variable $rank_{jk}$ takes the value 1 in this sum, and only the duration of the corresponding workpiece will be counted. Using this notation and the variables $empty_{mk}$ we are now able to write down the objective function (7.2.9).

$$\forall m \in MACH, k \in RANKS : dur_{mk} = \sum_{j \in JOBS} DUR_{mj} \cdot rank_{jk} \tag{7.2.8}$$

$$\text{minimize} \sum_{m=1}^{NM-1} dur_{m1} + \sum_{k=1}^{NJ-1} empty_{NM,k} = \sum_{m=1}^{NM-1} \sum_{j \in JOBS} DUR_{mj} \cdot rank_{j1} + \sum_{k=1}^{NJ-1} empty_{NM,k} \tag{7.2.9}$$

The first sum results in the point of time when the last machine (machine NM) starts working: this is the total time required for the first job on all preceding machines ($1, \ldots, NM-1$). The second sum is the total duration of idle times between processing operations on the last machine. The following graphic represents these quantities. Normally, we also needed to count the durations of all operations on the last machine, but this is a constant and may be omitted.

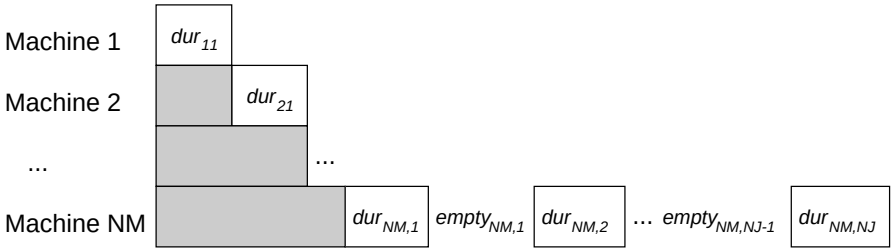

Figure 7.4: Relation between production and idle time

The tricky bit that is now left is to link the variables $wait_{mk}$ and $empty_{mk}$ to the variables $rank_{jk}$. To do so, we introduce $dnext_{mk}$, the time between the completion of job k on machine m and the start of job $k+1$ on machine $m+1$.

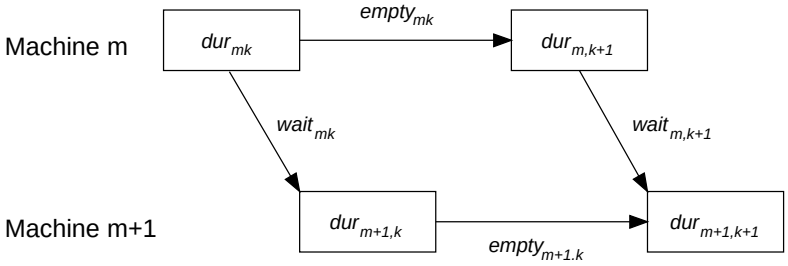

Figure 7.5: Calculation of the total duration

With the help of Figure 7.5, it is easy to see that the equations (7.2.10) hold. These can be developed to the constraints (7.2.11).

$$\forall m = 1, \ldots, NM - 1, k = 1, \ldots, NJ - 1 :$$
$$dnext_{mk} = empty_{mk} + dur_{m,k+1} + wait_{m,k+1} = wait_{mk} + dur_{m+1,k} + empty_{m+1,k} \tag{7.2.10}$$
$$\forall m = 1, \ldots, NM - 1, k = 1, \ldots, NJ - 1 :$$
$$empty_{mk} + \sum_{j \in JOBS} DUR_{mj} \cdot rank_{j,k+1} + wait_{m,k+1} = wait_{mk} + \sum_{j \in JOBS} DUR_{m+1,j} \cdot rank_{jk} + empty_{m+1,k} \tag{7.2.11}$$

The resulting model is given through the lines (7.2.1) to (7.2.7), (7.2.9), and (7.2.11). This is a mixed-integer problem since only the variables $rank_{jk}$ are constrained to be integer (more exactly, binary).

7.2.2 Implementation

The following Mosel program implements the mathematical model of the previous section. In order to extract the resulting schedule more easily, variables $start_{mk}$ have been added that indicate the start of the job with rank k on machine m. These variables are linked to the $rank_{jk}$ variables via an additional set of constraints.

```
model "B-2 Flow shop"
 uses "mmxprs"

 declarations
  NM = 3                                ! Number of machines
  NJ = 6                                ! Number of jobs
  MACH = 1..NM
  RANKS = 1..NJ
  JOBS = 1..NJ

  DUR: array(MACH,JOBS) of integer      ! Durations of jobs on machines

  rank: array(JOBS,RANKS) of mpvar      ! 1 if job j has rank k, 0 otherwise
  empty: array(MACH,1..NJ-1) of mpvar   ! Space between jobs of ranks k and k+1
  wait: array(1..NM-1,RANKS) of mpvar   ! Waiting time between machines m
                                        ! and m+1 for job of rank k
  start: array(MACH,RANKS) of mpvar     ! Start of job rank k on machine m
                                        ! (optional)
 end-declarations

 initializations from 'b2flowshop.dat'
  DUR
 end-initializations

! Objective: total waiting time (= time before first job + times between
! jobs) on the last machine
 TotWait:= sum(m in 1..NM-1,j in JOBS) (DUR(m,j)*rank(j,1)) +
           sum(k in 1..NJ-1) empty(NM,k)

! Every position gets a jobs
 forall(k in RANKS) sum(j in JOBS) rank(j,k) = 1

! Every job is assigned a rank
 forall(j in JOBS) sum(k in RANKS) rank(j,k) = 1

! Relations between the end of job rank k on machine m and start of job on
! machine m+1
 forall(m in 1..NM-1,k in 1..NJ-1)
  empty(m,k) + sum(j in JOBS) DUR(m,j)*rank(j,k+1) + wait(m,k+1) =
   wait(m,k) + sum(j in JOBS) DUR(m+1,j)*rank(j,k) + empty(m+1,k)

! Calculation of start times (to facilitate the interpretation of results)
 forall(m in MACH, k in RANKS)
  start(m,k) = sum(u in 1..m-1,j in JOBS) DUR(u,j)*rank(j,1) +
               sum(p in 1..k-1,j in JOBS) DUR(m,j)*rank(j,p) +
               sum(p in 1..k-1) empty(m,p)

! First machine has no idle times
```

```
 forall(k in 1..NJ-1) empty(1,k) = 0

! First job has no waiting times
 forall(m in 1..NM-1) wait(m,1) = 0

 forall(j in JOBS, k in RANKS) rank(j,k) is_binary

! Solve the problem
 minimize(TotWait)

end-model
```

It may be noted that it is possible to use an alternative formulation with Special Ordered Sets of type 1 (SOS1) instead of constraining variables $rank_{jk}$ to be binary. The corresponding line in the program above needs to be replaced by one of the following two lines:

```
 forall(j in JOBS) sum(k in RANKS) k*rank(j,k) is_sos1
```

or

```
 forall(k in RANKS) sum(j in JOBS) j*rank(j,k) is_sos1
```

In the first case, we define a SOS1 for every job, where the rank number is used as weight coefficient in the set; in the second case, a SOS1 is established for every rank position using the job numbers as weight coefficients. Exactly one variable of a SOS1 takes a value greater than 0, and due to the constraints (7.2.2) and (7.2.3) this value will be 1.

7.2.3 Results

The minimum waiting time calculated for the last machine is 9. With the variables $start_{mk}$ we can deduce a total duration of 35. A schedule (there are several possibilities) that has this minimum duration is shown in the following table.

Table 7.3: Optimal schedule of workpieces

Rank number	**1**	**2**	**3**	**4**	**5**	**6**
Workpiece number	3	1	4	6	5	2
Start on machine 1:	0	3	6	11	18	23
Start on machine 2:	3	6	11	18	23	29
Start on machine 3:	5	11	16	23	29	33

7.3 Job Shop Scheduling

A company has received an order for three types of wallpapers: one (paper 1) has a blue background with yellow patterns, another (paper 2) a green background and blue and yellow patterns, and the last (paper 3) has a yellow background with blue and green patterns. Every paper type is produced as a continuous roll of paper that passes through several machines, each printing a different color. The order in which the papers are run through the machines depends on the design of the paper: for paper 1 first the blue background and then the yellow pattern is printed. After the green background for paper 2, first the blue and then the yellow patterns are printed. The printing of paper 3 starts with the yellow background, followed by the blue and then the green patterns.

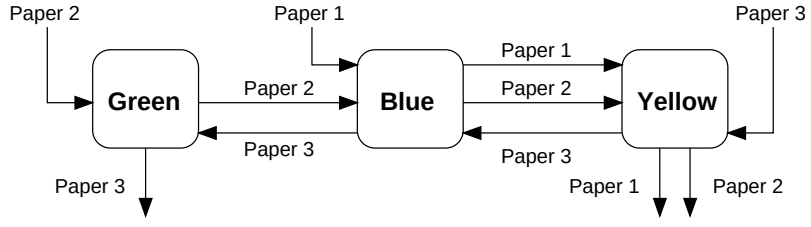

Figure 7.6: Production flows through printing machines

The processing times differ depending on the surface that needs to be printed. The times (in minutes) for applying every color of the three paper types are given in the following table.

Table 7.4: Times required for applying every color

Machine	Color	Paper 1	Paper 2	Paper 3
1	Blue	45	20	12
2	Green	-	10	17
3	Yellow	10	34	28

Knowing that every machine can only process one wallpaper at a time and that a paper cannot be processed by several machines simultaneously, how should the paper printing be scheduled on the machines in order to finish the order as early as possible?

7.3.1 Model formulation

Let *JOBS* be the set of jobs (paper types) and *MACH* the set of machines. We are going to use the variables $start_{mj}$ for the start of job j on machine m (supposing the the schedule starts at time 0). We write DUR_{mj} for the processing duration of job j on machine m. The variable *finish* indicates the completion time of the entire schedule. Since we want to minimize the completion time, the objective function is simply (7.3.1).

$$\text{minimize } finish \tag{7.3.1}$$

The schedule terminates when all paper types are completed, that is, when the last operation for every types finishes. The completion time of the schedule therefore needs to satisfy the following constraints (7.3.2) – (7.3.4).

$$finish \geq start_{31} + DUR_{31} \tag{7.3.2}$$
$$finish \geq start_{32} + DUR_{32} \tag{7.3.3}$$
$$finish \geq start_{23} + DUR_{23} \tag{7.3.4}$$

The constraints between processing operations are of two types: the so-called **conjunctive** constraints represent the precedences between the operations for a single paper type, and the **disjunctive** constraints express the fact that a machine can only execute a single operation at a time.

To start, we consider the conjunctive constraints. Paper 1 first goes onto the machine printing the blue color (machine 1), and then onto the one printing yellow (machine 3). This means that the processing of paper 1 on machine 1 needs to be finished when the processing on machine 3 starts, and hence, the constraint (7.3.5) needs to hold.

$$start_{11} + DUR_{11} \leq start_{31} \tag{7.3.5}$$

Similarly, paper 2 is first processed on machine 2, then on machine 1 and then on machine 3, which leads to the constraints (7.3.6) and (7.3.7).

$$start_{22} + DUR_{22} \leq start_{12} \tag{7.3.6}$$
$$start_{12} + DUR_{12} \leq start_{32} \tag{7.3.7}$$

For paper 3, that is processed in the order machine 3, machine 1, machine 2, we obtain the following constraints (7.3.8) and (7.3.9).

$$start_{33} + DUR_{33} \leq start_{13} \tag{7.3.8}$$
$$start_{13} + DUR_{13} \leq start_{23} \tag{7.3.9}$$

We now still need to model the disjunctions. Machine 1 has to process all three wallpaper types. Since a machine can only perform a single operation at any time, either paper 1 is printed before paper 2 or paper 2 is printed before paper 1. This is expressed by the following:

$$start_{11} + DUR_{11} \leq start_{12} \vee start_{12} + DUR_{12} \leq start_{11}$$

These two mutually exclusive constraints can be written as the constraints (7.3.10) and (7.3.11), where M is a large positive number and y_1 is a binary variable that takes the value 1 if paper 1 comes before paper 2 on machine 1, and 0 otherwise.

$$start_{11} + DUR_{11} \leq start_{12} + M \cdot (1 - y_1) \tag{7.3.10}$$
$$start_{12} + DUR_{12} \leq start_{11} + M \cdot y_1 \tag{7.3.11}$$

Let us see why this claim is correct. If paper 1 comes before paper 2 on machine 1, then y_1 has the value 1 and we obtain the constraints $start_{11} + DUR_{11} \leq start_{12}$ and $start_{12} + DUR_{12} \leq start_{11} + M$. The first constraint requires that the processing of paper 2 on machine 1 takes places after the printing of paper 1 is terminated. Whatever value $start_{11}$ and $start_{12}$ may take, the second constraint is automatically fulfilled due to the large value M. If, on the contrary, paper 2 is processed before paper 1 on machine 1, then y_1 has the value 0 and we obtain the constraints $start_{11} + DUR_{11} \leq start_{12} + M$ and $start_{12} + DUR_{12} \leq start_{11}$. This time, the first constraint is automatically fulfilled and the second guarantees that the processing of wallpaper 2 is terminated before the start of paper 1.

Proceeding in this way, we can translate all pairs of disjunctions between operations on the same machine by defining a binary variable y_d for every disjunction ($d = 1, \ldots, ND$). We thus obtain the following constraints (7.3.12) – (7.3.23):

$$start_{11} + DUR_{11} \leq start_{13} + M \cdot (1 - y_2) \tag{7.3.12}$$
$$start_{13} + DUR_{13} \leq start_{11} + M \cdot y_2 \tag{7.3.13}$$
$$start_{12} + DUR_{12} \leq start_{13} + M \cdot (1 - y_3) \tag{7.3.14}$$
$$start_{13} + DUR_{13} \leq start_{12} + M \cdot y_3 \tag{7.3.15}$$
$$start_{22} + DUR_{22} \leq start_{23} + M \cdot (1 - y_4) \tag{7.3.16}$$
$$start_{23} + DUR_{23} \leq start_{22} + M \cdot y_4 \tag{7.3.17}$$
$$start_{31} + DUR_{31} \leq start_{32} + M \cdot (1 - y_5) \tag{7.3.18}$$
$$start_{32} + DUR_{32} \leq start_{31} + M \cdot y_5 \tag{7.3.19}$$
$$start_{31} + DUR_{31} \leq start_{33} + M \cdot (1 - y_6) \tag{7.3.20}$$
$$start_{33} + DUR_{33} \leq start_{31} + M \cdot y_6 \tag{7.3.21}$$
$$start_{32} + DUR_{32} \leq start_{33} + M \cdot (1 - y_7) \tag{7.3.22}$$
$$start_{33} + DUR_{33} \leq start_{32} + M \cdot y_7 \tag{7.3.23}$$

The constraints (7.3.12) – (7.3.15) establish the rest of the disjunctions on machine 1, constraints (7.3.16) and (7.3.17) the disjunction of papers 2 and 3 on machine 2, and the constraints (7.3.18) – (7.3.23) the disjunctions between the processing of the three papers on machine 3.

To avoid numerical instabilities, the value of M should not be chosen too large. It is, for instance, possible to use as its value an upper bound UB on the schedule determined through some heuristic. In this case the following constraint (7.3.24) needs to be added to the mathematical model, which may also help to reduce the number of nodes explored by the tree search.

$$finish \leq UB \tag{7.3.24}$$

In our small example, we are going to use as the value for M the sum of all processing times as this gives a rough but safe upper bound.

To complete the formulation of the model, we need to add the non-negativity constraints for the start time variables (7.3.25) and constrain the y_d to be binaries (7.3.26).

$$\forall m \in MACH, j \in JOBS : start_{mj} \geq 0 \tag{7.3.25}$$
$$\forall d \in \{1, \ldots, ND\} : y_d \in \{0, 1\} \tag{7.3.26}$$

By renumbering the operations (tasks) using a single subscript (resulting in the set *TASKS*) we obtain a more general model formulation. The precedence relations and disjunctions can be represented as a **disjunctive graph** $G = (TASKS, ARCS, DISJS)$ where *ARCS* is the set of arcs (precedences) and *DISJS* the set of disjunctions. There is an arc (i, j) between two tasks i and j if i is the immediate predecessor of j. A disjunction $[i, j]$ indicates that tasks i and j are disjoint. With these conventions we obtain the following model (7.3.27) – (7.3.32).

$$\text{minimize } finish \tag{7.3.27}$$
$$\forall j \in TASKS : start_j + DUR_j \leq finish \tag{7.3.28}$$
$$\forall (i, j) \in ARCS : start_i + DUR_i \leq start_j \tag{7.3.29}$$
$$\forall [i, j] \in DISJ : start_i + DUR_i \leq start_j + M \cdot y_{ij} \wedge start_j + DUR_j \leq start_i + M \cdot (1 - y_{ij}) \tag{7.3.30}$$
$$\forall j \in TASKS : start_j \geq 0 \tag{7.3.31}$$
$$\forall [i, j] \in DISJ : y_{ij} \in \{0, 1\} \tag{7.3.32}$$

The objective function remains the same as before. The constraints (7.3.2) – (7.3.4) are generalized by the constraints (7.3.28): the completion time of the schedule is greater than or equal to the completion

time of the last operation for every paper time, and hence to the completion times of all operations. The constraints (7.3.29) state the conjunctions (precedences) and (7.3.30) the disjunctions. The last two sets of constraints are the non-negativity conditions for the start time variables (7.3.31) and the constraints turning the disjunction variables y_{ij} into binaries.

7.3.2 Implementation

The following Mosel program implements the mathematical model given in lines (7.3.27) – (7.3.32) of the previous section, that is, it uses a single index set *TASKS* for numbering the operations instead of a double machine-paper type index for start time variables and durations. In the set of examples on the book's website a second implementation is provided that uses the double subscript of the first mathematical model (lines (7.3.1) – (7.3.25) of the previous section).

The sequence of operations for every paper type is given in the form of a list of precedences: an entry in the table ARC_{ij} is defined if task *i* immediately precedes task *j*. The disjunctions are input in a similar way: an entry in the table $DISJ_{ij}$ is defined if and only if tasks *i* and *j* are processed on the same machine – and are therefore in disjunction.

```
model "B-3 Job shop"
 uses "mmxprs"

 declarations
  TASKS=1..8                          ! Set of tasks (operations)

  DUR: array(TASKS) of integer        ! Durations of jobs on machines
  ARC: dynamic array(TASKS,TASKS) of integer   ! Precedence graph
  DISJ: dynamic array(TASKS,TASKS) of integer  ! Disjunctions between jobs

  start: array(TASKS) of mpvar        ! Start times of tasks
  finish: mpvar                       ! Schedule completion time
  y: array(range) of mpvar            ! Disjunction variables
 end-declarations

 initializations from 'b3jobshop.dat'
  DUR ARC DISJ
 end-initializations

 BIGM:= sum(j in TASKS) DUR(j)        ! Some (sufficiently) large value

! Precedence constraints
 forall(j in TASKS) finish >= start(j)+DUR(j)
 forall(i,j in TASKS | exists(ARC(i,j)) ) start(i)+DUR(i) <= start(j)

! Disjunctions
 d:=1
 forall(i,j in TASKS | i<j and exists(DISJ(i,j)) ) do
  create(y(d))
  y(d) is_binary
  start(i)+DUR(i) <= start(j)+BIGM*y(d)
  start(j)+DUR(j) <= start(i)+BIGM*(1-y(d))
  d+=1
 end-do

! Bound on latest completion time
 finish <= BIGM

! Solve the problem: minimize latest completion time
 minimize(finish)

end-model
```

The implementation of this model uses some features of the Mosel language that have not yet been introduced:

In the `declarations` block the reader may have noted the keywords `dynamic array` and `range`. Both indicate that the corresponding arrays are defined as dynamic arrays. `range` indicates an unknown index set that is a consecutive sequence of integers (which may include 0 or negative values). We do not have to enumerate the array of variables *y* so that there is no need to name its index set. We could proceed in the same way for the arrays *ARC* and *DISJ*, but since we know that their index sets are subsets of *TASKS* (which is also used later on for enumerating the array entries) we use this name for the index sets.

However, the set *TASKS* is defined as a constant set directly at its declaration, which means that any array with this index will be created with a fixed size. For both arrays, *ARC* and *DISJ* only very few entries are defined, and it is therefore preferable to force them to be initialized dynamically with the data read from file – this is done with the keyword `dynamic`.

In the `forall` loops enumerating the precedence arcs and the disjunctions, we use the function `exists` to test whether an array entry is defined. For arrays with few defined entries (**sparse arrays**) this is a very efficient way of enumerating these entries. Note that for an efficient use of `exists` the index sets in the loop must be the same as those used in the definition of the array that the condition bears on.

Another new feature is the `forall-do` loop used in the formulation of the disjunctions: so far we have only seen the inline version that loops over a single statement. The full version needs to be used if the loop applies to a block of program statements.

7.3.3 Results

After solving this problem we obtain a total completion time of 97. The start and completion times for all jobs are given in the following table (there are several possible solutions).

Table 7.5: Starting times of operations

	Paper 1	Paper 2	Paper 3
Blue	42 – 87	10 – 30	30 – 42
Green	–	0 – 10	42 – 59
Yellow	87 – 97	30 – 64	0 – 28

This schedule may be represented as a bar chart, also called a **Gantt chart**, that has the time as its horizontal axis. Every task (job) is represented by a rectangle, the length of which is proportional to its duration. All operations on a single machine are placed in the same line.

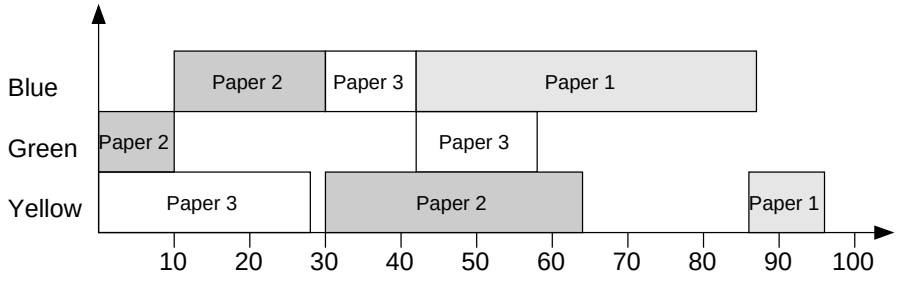

Figure 7.7: Gantt chart

7.4 Sequencing jobs on a bottleneck machine

In workshops it frequently happens that a single machine determines the throughput of the entire production (for example, a machine of which only one is available, the slowest machine in a production line, *etc.*). This machine is called the **critical machine** or the **bottleneck**. In such a case it is important to schedule the tasks that need to be performed by this machine in the best possible way.

The aim of the problem is to provide a simple model for scheduling operations on a single machine and that may be used with different objective functions. 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 and duration are given. For the last optimization criterion (total tardiness), a due date (latest completion time) is also required to measure the tardiness, that is, the amount of time by which the completion of jobs exceeds their respective due dates. The following table lists the data for our problem.

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?

Table 7.6: 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

7.4.1 Model formulation

We are going to deal with the different objective functions in sequence, but the body of the model will remain the same. For writing a model that corresponds to the three objectives at a time, we are going to use binary variables $rank_{jk}$ ($j, k \in JOBS = \{1, \ldots, NJ\}$) that have the value 1 if job j has the position (rank) k and 0 in all other cases. There can be only one job per position k and every job j takes a single position. This leads to constraints (7.4.1) and (7.4.2).

$$\forall k \in JOBS : \sum_{j \in JOBS} rank_{jk} = 1 \tag{7.4.1}$$

$$\forall j \in JOBS : \sum_{k \in JOBS} rank_{jk} = 1 \tag{7.4.2}$$

The processing time for the job in position k is given by the sum $\sum_{j \in JOBS} DUR_j \cdot rank_{jk}$ (where DUR_j denotes the duration given in the table in the previous section). Only for the job j in position k does the variable $rank_{jk}$ have the value 1. By multiplying with the duration of job j, we are able to retrieve the duration of the job at position k. Using this technique we can write the constraints of our problem. 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 (given as REL_j) of the job assigned to this position. This gives the constraints (7.4.3):

$$\forall k \in JOBS : start_k \geq \sum_{j \in JOBS} REL_j \cdot rank_{jk} \tag{7.4.3}$$

Another constraint used by all models specifies 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 constraints (7.4.4).

$$\forall k \in \{1, \ldots, NJ - 1\} : start_{k+1} \geq start_k + \sum_{j \in JOBS} DUR_j \cdot rank_{jk} \tag{7.4.4}$$

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 k). The line (7.4.5) models this objective. The complete model is then given by the following:

$$\text{minimize } start_{NJ} + \sum_{j \in JOBS} DUR_j \cdot rank_{j,NJ} \tag{7.4.5}$$

$$\forall k \in JOBS : \sum_{j \in JOBS} rank_{jk} = 1 \tag{7.4.6}$$

$$\forall j \in JOBS : \sum_{k \in JOBS} rank_{jk} = 1 \tag{7.4.7}$$

$$\forall k \in JOBS : start_k \geq \sum_{j \in JOBS} REL_j \cdot rank_{jk} \tag{7.4.8}$$

$$\forall k \in \{1, \ldots, NJ - 1\} : start_{k+1} \geq start_k + \sum_{j \in JOBS} DUR_j \cdot rank_{jk} \tag{7.4.9}$$

$$\forall k \in JOBS : start_k \geq 0 \tag{7.4.10}$$

$$\forall j, k \in JOBS : rank_{jk} \in \{0, 1\} \tag{7.4.11}$$

Objective 2: For minimizing the average processing time, we introduce additional variables $comp_k$ (representing the completion times of the job in position k) to simplify the notation. We add the following two sets of constraints to the problem to obtain these completion times:

$$\forall k \in JOBS : comp_k = start_k + \sum_{j \in JOBS} DUR_j \cdot rank_{jk} \tag{7.4.12}$$

$$\forall k \in JOBS : comp_k \geq 0 \tag{7.4.13}$$

The new objective (7.4.14) consists of minimizing the average processing time, or equivalently, minimizing the sum of the job completion times.

$$\text{minimize} \sum_{k \in JOBS} comp_k \tag{7.4.14}$$

The complete model for the second objective consists of lines (7.4.14) and (7.4.6) to (7.4.13).

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 0. We thus obtain the constraints (7.4.15) and (7.4.16).

$$\forall k \in JOBS : late_k = \max(0, comp_k - \sum_{j \in JOBS} DUE_j \cdot rank_{jk}) \tag{7.4.15}$$

$$\forall k \in JOBS : late_k \geq 0 \tag{7.4.16}$$

The new objective function (7.4.17) minimizes the total tardiness of all jobs:

$$\text{minimize} \sum_{k \in JOBS} late_k \tag{7.4.17}$$

The new model is given by the constraints (7.4.17), (7.4.6) to (7.4.13), and (7.4.15) to (7.4.16).

7.4.2 Implementation

The Mosel implementation below solves the same problem three times, each time with a different objective, and prints the resulting solutions. To simplify the representation, we use the completion time variables already in the formulation of the first objective. For the calculation of the tardiness of jobs, this implementation uses the fact that by default all variables are non-negative. That means that if the constraint that calculates the difference between the completion time and the due date results in a negative lower bound for $late_k$, then the Xpress-Optimizer will set this variable to 0.

```
model "B-4 Sequencing"
 uses "mmxprs"

 forward procedure print_sol(obj:integer)

 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,JOBS) of mpvar  ! =1 if job j at position k
  start: array(JOBS) of mpvar      ! Start time of job at position k
  comp: array(JOBS) of mpvar       ! Completion time of job at position k
  late: array(JOBS) of mpvar       ! Tardiness of job at position k
  finish: mpvar                    ! Completion time of the entire schedule
 end-declarations

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

! One job per position and one position per job
  forall(k in JOBS) sum(j in JOBS) rank(j,k) = 1
  forall(j in JOBS) sum(k in JOBS) rank(j,k) = 1

! Sequence of jobs
  forall(k in 1..NJ-1)
   start(k+1) >= start(k) + sum(j in JOBS) DUR(j)*rank(j,k)

! Start times
  forall(k in JOBS) start(k) >= sum(j in JOBS) REL(j)*rank(j,k)

! Completion times
```

```
  forall(k in JOBS) comp(k) = start(k) + sum(j in JOBS) DUR(j)*rank(j,k)

 forall(j,k in JOBS) rank(j,k) is_binary

! Objective function 1: minimize latest completion time
 forall(k in JOBS) finish >= comp(k)
 minimize(finish)
 print_sol(1)

! Objective function 2: minimize average completion time
 minimize(sum(k in JOBS) comp(k))
 print_sol(2)

! Objective function 3: minimize total tardiness
 forall(k in JOBS) late(k) >= comp(k) - sum(j in JOBS) DUE(j)*rank(j,k)
 minimize(sum(k in JOBS) late(k))
 print_sol(3)

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

! Solution printing
 procedure print_sol(obj:integer)
  writeln("Objective ", obj, ": ", getobjval,
          if(obj>1, "  completion time: " + getsol(finish), "") )
  write("\t")
  forall(k in JOBS) write(strfmt(getsol(sum(j in JOBS) j*rank(j,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))
  write("\nDue\t")
  forall(k in JOBS) write(strfmt(getsol(sum(j in JOBS) DUE(j)*rank(j,k)),4))
  if(obj=3) then
   write("\nLate\t")
   forall(k in JOBS) write(strfmt(getsol(late(k)),4))
  end-if
  writeln
 end-procedure

end-model
```

Similarly to the implementation of the first problem in this chapter, we define a procedure for printing out the solution that is called after every optimization run. In this example, we want to introduce some variations in the way the solutions are printed depending on the optimization criterion, we therefore pass the number of the latter as **parameter** to the subroutine. For selecting the information that is to be printed we use two different versions of the `if` statement: the inline `if` and `if-then` that includes a block of statements.

The reader may have been wondering why we did not use the probably more obvious pair *start* – *end* for naming the variables in this example: `end` is a keyword of the Mosel language (see Section 5.2.3), 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.

7.4.3 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 4 → 5 → 2 → 6 → 7 → 3 → 1. If we compare the completion times with the due dates we see that the 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 → 1 → 4 → 6 → 2 → 7 → 3 where jobs 4 and 7 finish one time unit late and job 3 is late by 16, it terminates at time 31 instead of being ready at its due date 15. This schedule has an average completion time of 15.71.

7.5 Paint production

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 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 *CLEAN* where $CLEAN_{ij}$ denotes the cleaning time between batch i and batch j.

Table 7.7: 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.

7.5.1 Model formulation

We may try to model this problem as an assignment problem like the assignment of personnel to work-posts (Chapter 14) and the problem of flight connections (Chapter 11). Let $JOBS = \{1, \ldots, NJ\}$ 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 $succ_{ij}$ that take the value 1 if and only if batch j succeeds batch i. This means that we need to decide which successor is assigned to every batch i. We thus find the classical formulation of an assignment problem.

$$\text{minimize} \sum_{i \in JOBS} \sum_{j \in JOBS, j \neq i} (DUR_i + CLEAN_{ij}) \cdot succ_{ij} \tag{7.5.1}$$

$$\forall i \in JOBS : \sum_{j \in JOBS, j \neq i} succ_{ij} = 1 \tag{7.5.2}$$

$$\forall j \in JOBS : \sum_{i \in JOBS, i \neq j} succ_{ij} = 1 \tag{7.5.3}$$

$$\forall i, j \in JOBS, i \neq j : succ_{ij} \in \{0, 1\} \tag{7.5.4}$$

Note that we do not use any variables $succ_{ij}$ with $i = j$ since a batch may not appear more than once in the production cycle. It would be possible to include these variables and penalize them through a large value in the diagonal of *CLEAN*. The constraints (7.5.2) and (7.5.3) guarantee that every batch has a single successor and a single predecessor.

The objective function (7.5.1) sums up the processing time of i and the cleaning time between i and j for every pair of batches (i, j). Due to the constraints (7.5.2) and (7.5.3), for every batch i these durations will only be counted for the batch j that immediately succeeds i (that is, the j for which $succ_{ij} = 1$ holds).

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 $1 \rightarrow 3 \rightarrow 2 \rightarrow 1$ and $4 \rightarrow 5 \rightarrow 4$. In Linear Programming, the constraints that force a set of assignment variables to form a single cycle are tricky. A first possibility is to add the constraints (7.5.5) to our model.

$$\forall S \subseteq \{2, \ldots, NJ\} : \sum_{(i,j) \in S} succ_{ij} \leq |S| - 1 \tag{7.5.5}$$

If a solution contains a sub-cycle with a subset S of batches, the sum of the $succ_{ij}$ at 1 in S has the same value as the number of batches in S. For example, there are three batches (1, 2, and 3) in the first sub-cycle of the previous solution. By imposing a smaller cardinality, the constraints (7.5.5) force the sequence of batches to 'enter' and 'leave' the set S. These constraints concern all subsets S that do not contain the first batch (or any other fixed batch), because otherwise even the single cycle we wish to generate would be excluded. This formulation generates an exponential number of constraints (2^{NJ-1}). We therefore choose a weaker formulation with a real variable y_j per batch and $NJ \cdot (NJ - 1)$ constraints (7.5.7).

$$\forall j \in JOBS : y_j \geq 0 \tag{7.5.6}$$

$$\forall i \in JOBS, \forall j = 2, \ldots, NJ, i \neq j : y_j \geq y_i + 1 - NJ \cdot (1 - succ_{ij}) \tag{7.5.7}$$

To understand these constraints, suppose the solution of the assignment problem consists of several sub-cycles and we choose a sub-cycle that does not contain the batch 1, such as $4 \rightarrow 5 \rightarrow 4$. The constraints (7.5.7) with $succ_{ij} = 1$ for this sub-cycle are:

$$y_5 \geq y_4 + 1$$
$$y_4 \geq y_5 + 1$$

By combining these constraints, we obtain $y_4 - 1 \geq y_5 \geq y_4 + 1$ or $-1 \geq 1$, a contradiction. A solution that is divided into sub-cycles is therefore not feasible for the assignment problem with the additional constraints (7.5.7). If, on the contrary, the cycle is not subdivided, then values for y_j exist that fulfill the constraints. This is for instance the case if y_j denotes the **rank** of the batch j in the cycle, choosing batch 1 as the starting position (with $y_1 = 1$). For the variables $succ_{ij}$ at 1, the constraints result in $y_j \geq y_i + 1$ and are therefore satisfied. For the 0-valued variables $succ_{ij}$ they evaluate to $y_i - y_j \leq NJ - 1$ and are also satisfied since the y_j take values between 1 and NJ, and hence their difference does not exceed $NJ - 1$.

To summarize, the mathematical model for our paint problem is the assignment problem in lines (7.5.1) to (7.5.4) with the additional constraints in (7.5.6) and (7.5.7).

7.5.2 Implementation

The transformation of the mathematical model into a Mosel program is fairly straightforward.

```
model "B-5 Paint production"
 uses "mmxprs"

 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

  succ: array(JOBS,JOBS) of mpvar     ! =1 if batch i is followed by batch j,
                                      ! =0 otherwise
  y: array(JOBS) of mpvar             ! Variables for excluding subtours
 end-declarations

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

! Objective: minimize the duration of a production cycle
 CycleTime:= sum(i,j in JOBS | i<>j) (DUR(i)+CLEAN(i,j))*succ(i,j)

! One successor and one predecessor per batch
 forall(i in JOBS) sum(j in JOBS | i<>j) succ(i,j) = 1
 forall(j in JOBS) sum(i in JOBS | i<>j) succ(i,j) = 1

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

 forall(i,j in JOBS | i<>j) succ(i,j) is_binary

! Solve the problem
 minimize(CycleTime)

! Solution printing
 writeln("Minimum cycle time: ", getobjval)
 writeln("Sequence of batches:\nBatch Duration Cleaning")
 first:=1
 repeat
  second:= integer(sum(j in JOBS | first<>j) j*getsol(succ(first,j)) )
  writeln("  ",first, strfmt(DUR(first),8), strfmt(CLEAN(first,second),9))
  first:=second
 until (second=1)

end-model
```

The listing above again includes the solution printing and it makes use of some new features: we use a `repeat-until` loop for enumerating the batches in the order that they have been scheduled. For every

batch (`first`) we calculate its successor (`second`) that then becomes on its turn the one for which we search a successor. The loop stops when we are back at the starting point (batch 1). The calculation of the successor is based on the solution values of the successor variables. This sum is of type `real`, but has an integer value, and we need to transform it to the type `integer`. The function `integer` that is employed here truncates the value that it is applied to (in many cases it is preferable to use the function `round` instead, which rounds to the nearest integer).

7.5.3 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. This problem is highly combinatorial: for NJ batches, there are $(NJ-1)!$ possible cycles. In this simple example, it would be possible to enumerate by hand the 24 possibilities but already for 10 batches there are 9! = 362,880 different sequences and for 20 batches, $19! \approx 1.2 \cdot 10^{17}$ sequences.

7.6 Assembly line balancing

An electronics factory produces an amplifier on an assembly line with four workstations. An amplifier is assembled in twelve operations between which there are certain precedence constraints. The following table indicates the duration of every task (in minutes) and the list of its immediate predecessors (the abbreviation PCB used in this table stands for 'printed circuit board').

The production manager would like to distribute the tasks among the four workstations, subject to the precedence constraints, in order to balance the line to obtain the shortest possible cycle time, that is, the total time required for assembling an amplifier. Every task needs to be assigned to a single workstation that has to process it without interruption. Every workstation deals with a single operation at a time. We talk about **cycle time** because the operations on every workstation will be repeated for every amplifier. When an amplifier is finished, the amplifiers at stations 1 to 3 advance by one station, and the assembly of a new amplifier is started at the first workstation.

Table 7.8: List of tasks and predecessors

Task	Description	Duration	Predecessors
1	Preparing the box	3	–
2	PCB with power module	6	1
3	PCB with pre-amplifier	7	1
4	Filter of the amplifier	6	2
5	Push-pull circuit	4	2
6	Connecting the PCBs	8	2,3
7	Integrated circuit of the pre-amplifier	9	3
8	Adjusting the connections	11	6
9	Heat sink of the push-pull	2	4,5,8
10	Protective grid	13	8,11
11	Electrostatic protection	4	7
12	Putting on the cover	3	9,10

7.6.1 Model formulation

Let $TASKS$ be the set of tasks, DUR_i the duration of task i, $MACH$ the set of workstations (numbered in the order of the production flow). The precedence relations can be represented as a directed graph $G = (TASKS, ARCS)$, where $ARCS$ denotes the set of arcs. An arc (i, j) from task i to j symbolizes that i is the immediate predecessor of j.

To assign the tasks, we define binary variables $process_{im}$ with $process_{im} = 1$ if and only if task i is assigned to workstation m. The constraints (7.6.1) are required to assure that every task goes to a single workstation.

$$\forall i \in TASKS : \sum_{m \in MACH} process_{im} = 1 \tag{7.6.1}$$

An assignment is valid only if it fulfills the precedence constraints, which means that for every arc (i, j) the workstation processing i must have a number that is smaller or equal to the workstation number for

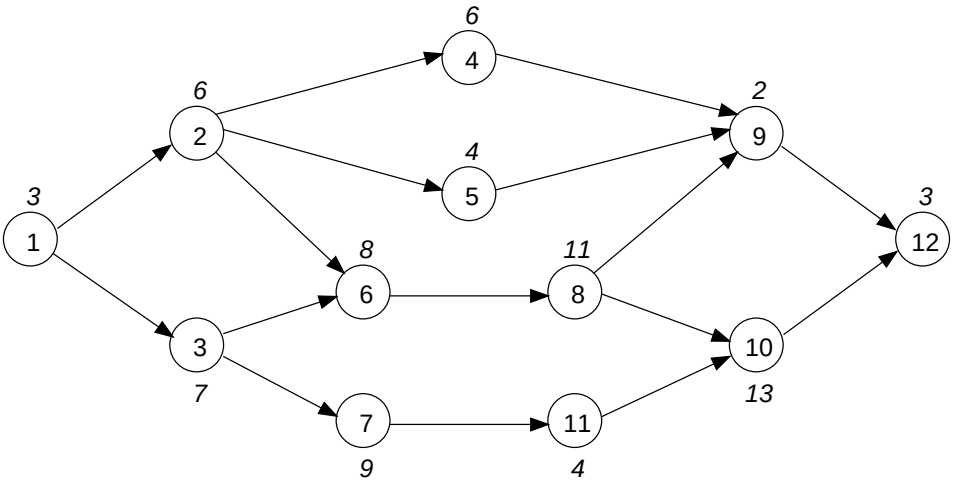

Figure 7.8: Graph of tasks with durations

j. This relation is established with the constraints (7.6.2). Note how the numbers of the workstations for i and j are calculated with sums of the assignment variables: due to constraints (7.6.1) only the index m for which $process_{im} = 1$ will be counted.

$$\forall (i,j) \in ARCS : \sum_{m \in MACH} k \cdot process_{im} \leq \sum_{m \in MACH} k \cdot process_{jm} \tag{7.6.2}$$

We now introduce a real variable $cycle \geq 0$ that represents the cycle time. The constraints (7.6.3) indicate that this variable is an upper bound on the workload assigned to every workstation.

$$\forall m \in MACH : \sum_{i \in TASKS} DUR_i \cdot process_{im} \leq cycle \tag{7.6.3}$$

We thus obtain the following mathematical model, where all variables with the exception of *cycle* are binaries. The objective function (7.6.4) consists of minimizing *cycle*.

$$\text{minimize } cycle \tag{7.6.4}$$

$$\forall i \in TASKS : \sum_{m \in MACH} process_{im} = 1 \tag{7.6.5}$$

$$\forall (i,j) \in ARCS : \sum_{m \in MACH} k \cdot process_{im} \leq \sum_{m \in MACH} k \cdot process_{jm} \tag{7.6.6}$$

$$\forall m \in MACH : \sum_{i \in TASKS} DUR_i \cdot process_{im} \leq cycle \tag{7.6.7}$$

$$\forall i \in TASKS, m \in MACH : process_{im} \in \{0,1\} \tag{7.6.8}$$

$$cycle \geq 0 \tag{7.6.9}$$

In this problem formulation, we assume the the production line already exists with its *NM* workstations and that we wish to balance the workload. For (re-)designing production lines, it would possible to work with a given market demand (number of amplifiers per day) and a given cycle time to minimize the number of workstations that form the new production line. To be able to generate solutions for this problem, the maximum duration of tasks obviously must not exceed the given cycle time.

To solve this new version of the problem, we may take the preceding model, delete the objective function and replace the variable *cycle* by a constant (the desired cycle time). We then try to solve the system (7.6.5) to (7.6.9), initializing the number of machines *NM* with some lower bound, such as the total duration of all tasks divided by the cycle time and rounding the resulting value to the next larger integer. If this system has no solution, we increment the value of *NM* and restart the solution algorithm. This process converges since a trivial solution exists with one task per workstation.

7.6.2 Implementation

The following Mosel program implements the mathematical model defined by lines (7.6.4) – (7.6.9) in the previous section. It poses a frequently recurring problem, namely the encoding of the graph *G*. One

possibility consists of defining a binary matrix B (called an **adjacency matrix**), with $B_{ij} = 1$ if and only if the arc (i, j) is in G. The second method that is used here lists only the existing arcs and therefore has a lower memory consumption for sparse graphs like G: the graph is represented by a table ARC with a line per arc and two columns. The arc represented by the line a has the source ARC_{a1} and the sink ARC_{a2}. The first data line contains the number 1 and 2 for the arc (1,2), the second 1 and 3 for (1,3) and so on. Since we initialize the arc numbers dynamically with the data read from file (the range of arc numbers RA is not fixed in the program), the index-tuples for the entries of the array ARC need to be given in the data file `b6linebal.dat`.

```
model "B-6 Assembly line balancing"
 uses "mmxprs"

 declarations
  MACH=1..4                            ! Set of workstations
  TASKS=1..12                          ! Set of tasks

  DUR: array(TASKS) of integer         ! Durations of tasks
  ARC: array(RA:range, 1..2) of integer ! Precedence relations between tasks

  process: array(TASKS,MACH) of mpvar   ! 1 if task on machine, 0 otherwise
  cycle: mpvar                         ! Duration of a production cycle
 end-declarations

 initializations from 'b6linebal.dat'
  DUR ARC
 end-initializations

! One workstation per task
 forall(i in TASKS) sum(m in MACH) process(i,m) = 1

! Sequence of tasks
 forall(a in RA) sum(m in MACH) m*process(ARC(a,1),m) <=
                 sum(m in MACH) m*process(ARC(a,2),m)

! Cycle time
 forall(m in MACH) sum(i in TASKS) DUR(i)*process(i,m) <= cycle

 forall(i in TASKS, m in MACH) process(i,m) is_binary

! Minimize the duration of a production cycle
 minimize(cycle)
end-model
```

7.6.3 Results

The mixed integer solver finds a minimum cycle time of 20 minutes. The assignment of tasks to workstations and the resulting workload per workstation are given in the following table. It is easy to verify that the precedence constraints hold.

Table 7.9: Assignment with minimum cycle time

Workstation number	**1**	**2**	**3**	**4**
Assigned tasks	1,2,4,5	3,7,11	6,8	9,10,12
Workload	19	20	19	18

7.7 References and further material

With the exception of the stadium construction all problems in this chapter are very difficult combinatorial problems, so-called **NP-hard** problems (the exact definition of this notion is given in [GJ79]). There is a large variety of scheduling. The works of Carlier and Chretienne [CC88], Lopez and Roubellat [LR99], Błazewicz et al. [BESW93], and French [Fre82] are good entry points for this very active domain.

The stadium construction problem (Section 7.1) is a project scheduling problem that is now easily solved and for which a number of software tools are available, such as Project by Microsoft and PSN by Scitor-Le Bihan. To answer question 1, there are simple solution methods based on graphs. The graph of a project may be coded in two different ways. The first consists of representing tasks by arcs and the nodes

correspond to the stages of the project. This graph, called AOA (**activities on arcs**) is used by the **PERT method** [Eva64]. In the representation that we have used the nodes correspond to the tasks, this is the AON (**activities on nodes**) encoding that is used by the **Method of Potentials** [Pri94a] [GM90].

Historically, the PERT method stems from the USA and the Method of Potentials from France. Besides the encoding of the graph, the two methods are largely equivalent, to the point that certain software provides both methods. Nevertheless, the AOA representation becomes quite impractical outside the context of project scheduling, for instance for workshop planning. In this case, the AON notation is used systematically.

The question 2 concerns the PERT-cost problem, or scheduling with project crashing, in which the objective is to minimize the total cost if the tasks have flexible durations and the costs depend on these durations. Depending on the assumptions, several model formulations are possible: for example, an additional cost that is proportional to the number of days saved per task, or a cost inversely proportional to the duration of a task. The reader may find other practical cases in Wasil [WA88].

The **flow shop** problem (Section 7.2) is a classical scheduling problem. The mathematical formulation we have used is due to Wagner [Wag59]. Other solution methods are described by French [Fre82], pages 132-135, and Pinedo [Pin95a], pages 99-101. Curiously, these works forget to mention the constraints (7.2.6) – (7.2.7) that are absolutely necessary for obtaining correct results.

The flow-shop problem is NP-hard in the general case, and Mathematical Programming only allows solving problems limited to about thirty operations. For large cases, simple heuristics like NEH (from the name of the authors Nawaz, Enscore and Ham [NEH83]) or metaheuristics like tabu search [WH89] are used. The two-machine case is easy and may be solved by an algorithm by Johnson that is described in the two books just quoted.

The problem in Section 7.3 is a classical scheduling problem called **job shop**. This problem is NP-hard in the general case. Modeling as a MIP is often inefficient for large problems. Nevertheless, the generic model formulation given here is used by Applegate and Cook with cutting plane techniques [AC91] where the authors obtain interesting results for difficult instances. But the problem is usually solved by list-based heuristics or with exact methods based on a formulation with disjunctive graphs [CCE+93], [Fre82], [Pin95b]. Note that the generic model formulation is also valid for the general flow shop problem (whilst the model of Section 7.2 only applies to the permutation flow shop).

Different solution algorithms are available for one-machine problems (Section 7.4). The aim of this exercise was to give a common model formulation for the three objectives. Other models using variables relative to the position of jobs may be found in [LQ92] [Sev98].

The paint production problem (Section 7.5) is a representative of a family of scheduling problems where the preparation before every task depends on the chosen order (**sequence-dependent setup times**). We have in fact used one of the models for the famous **traveling salesman problem** (TSP). In this problem, the jobs (batches) become cities and the cleaning times the cost of traveling from one city to the next. The objective is to find the minimum cost for the journey of a sales representative that starts from his home in city 1, visits every city once, and finally returns to his starting point.

Here we have considered the asymmetric case, that is, the cleaning time between batch *i* and batch *j* is in general different from the one between *j* and *i*. In Chapter 11, we study a symmetric TSP, with distances between cities as the crow flies. It is solved with a different technique, by progressively adding constraints of type (5). Since the TSP is NP-hard, the models presented often take too long to be solved for more than twenty cities. Beyond this limit, specialized tree search methods [HK70] need to be used. Demonstration versions of such methods are available [Ros85] [EM92], and a Pascal code is given by Syslo [SDK83]. For cases with hundreds of cities one needs to give up on optimality and use heuristic techniques, some of which like tabu search (*e.g.* [GTdW93]) or simulated annealing ([KGV83]) are in practice quite efficient. Such heuristics are explained with some detail in [Pri94a].

The problem of assembly line balancing (Section 7.6) is NP-hard for the two given versions (minimizing the cycle time or the number of workstations). Without precedence constraints, the first version is equivalent to a scheduling problem with parallel machines (see the wagon loading problem in Chapter 9) whilst the second is reduced to a bin-packing problem (the problem of saving files to disks in Chapter 9). Problems of about twenty tasks can be dealt with in Mathematical Programming, beyond this limit heuristics may be used like the one by Dar-El [DE73], described together with another heuristic in the book on production planning by Buffa and Sarin, pages 660-671 [BS87]. Scholl has recently published a book dedicated to assembly line balancing problems [Sch99].

Chapter 8
Planning problems

The term **planning** is often used with three different meanings in the context of production management: **long-term planning** that is used for taking **strategic** decisions, **mid-term planning** that corresponds to the **tactical** level, and **short-term planning** for **operational** decisions. Long-term planning concerns the direction a company is taking over several years, like the depot location problem in Chapter 10. Short-term planning takes detailed production decisions for the shop floor, with a time horizon of a few days. Most importantly it comprises scheduling problems such as those described in Chapter 7. The present chapter is dedicated to tactical planning, currently referred to as **production planning**.

This decision level often covers a time horizon of one or several months, working for instance with time slices of the length of a week. It takes a macroscopic view of the production system: the detailed constraints for workshops and machines are ignored, the operations are dealt with in an **aggregated** fashion. The objective is to determine the best quantities to produce in every time period whilst, in a majority of cases, minimizing the cost incurred through production and storage.

Section 8.1 presents a simple case of single product planning, namely bicycles. The following section (8.2) deals with the production of several products (different types of glasses). In Section 8.3 we plan the production of toy lorries with all their components. This is a typical case of Material Requirement Planning (MRP). The two following problems have more complex cost functions: the production of electronic components in Section 8.4 includes costs for changing the level of production, and for the production of fiberglass in Section 8.5 the cost depends on the time period. The problem of Section 8.6 stands apart: it concerns the assignment of product batches to machines of different capacities and speeds.

8.1 Planning the production of bicycles

A company produces bicycles for children. The sales forecast in thousand of units for the coming year are given in the following table. The company has a capacity of 30,000 bicycles per month. It is possible to augment the production by up to 50% through overtime working, but this increases the production cost for a bicycle from the usual € 32 to € 40.

Table 8.1: Sales forecasts for the coming year in thousand units

Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
30	15	15	25	33	40	45	45	26	14	25	30

Currently there are 2,000 bicycles in stock. The storage costs have been calculated as € 5 per unit held in stock at the end of a month. We assume that the storage capacity at the company is virtually unlimited (in practice this means that the real capacity, that is quite obviously limited, does not impose any limits in our case). We are at the first of January. Which quantities need to be produced and stored in the course of the next twelve months in order to satisfy the forecast demand and minimize the total cost?

8.1.1 Model formulation

The variables that we need to determine are the numbers of bicycles $pnorm_t$ and $pover_t$ to be produced respectively in normal working hours and in overtime hours during month t, and the number of bicycles $store_t$ held in stock at the end of the month. Let $MONTHS = \{1, \ldots, 12\}$ be the set of time periods. The objective is to minimize the total cost, that is, the sum of cost of production (in normal and additional hours) and the storage cost. We write *CNORM* and *COVER* for the production cost of a bicycle in normal

and overtime hours, and *CSTOCK* the monthly storage cost per bicycle. With these conventions we obtain the objective function (8.1.1).

$$\text{minimize} \sum_{t \in MONTHS} (CNORM \cdot pnorm_t + COVER \cdot pover_t + CSTOCK \cdot store_t) \tag{8.1.1}$$

In every month t, the available quantity of bicycles is given as the sum of bicycles in stock at the end of month $t-1$ and the bicycles produced in normal and overtime hours during this month. This sum must equal the number of bicycles sold in the course of month t, plus the number of bicycles held in stock at the end of month t. This key relation is called the **inventory balance equation**, and forms an important component of mathematical models for production planning problems.

Let us call the initial stock level *ISTOCK* and DEM_t the forecasted demand for month t. The balance constraint for the first month is then given by relation (8.1.2)

$$pnorm_1 + pover_1 + ISTOCK = DEM_1 + store_1 \tag{8.1.2}$$

The constraints (8.1.3) establish this relation for all following months.

$$\forall t \in MONTHS, t \neq 1 : pnorm_t + pover_t + store_{t-1} = DEM_t + store_t \tag{8.1.3}$$

The production capacity in normal working hours is limited by the given capacity *CAP*, and the additional production in overtime hours is limited to 50% of this capacity; hence the constraints (8.1.4) and (8.1.5).

$$\forall t \in MONTHS : pnorm_t \leq CAP \tag{8.1.4}$$
$$\forall t \in MONTHS : pover_t \leq 0.5 * CAP \tag{8.1.5}$$

And finally, we note that all variables are non-negative (constraints (8.1.6)).

$$\forall t \in MONTHS : pnorm_t \geq 0, pover_t \geq 0, store_t \geq 0 \tag{8.1.6}$$

8.1.2 Implementation

The mathematical model may be translated into the following Mosel program.

```
model "C-1 Bicycle production"
 uses "mmxprs"

 declarations
  TIMES = 1..12                  ! Range of time periods

  DEM: array(TIMES) of integer   ! Demand per months
  CNORM,COVER: integer           ! Prod. cost in normal / overtime hours
  CSTOCK: integer                ! Storage cost per bicycle
  CAP: integer                   ! Monthly capacity in normal working hours
  ISTOCK: integer                ! Initial stock

  pnorm:array(TIMES) of mpvar    ! No. of bicycles produced in normal hours
  pover:array(TIMES) of mpvar    ! No. of bicycles produced in overtime hours
  store:array(TIMES) of mpvar    ! No. of bicycles stored per month
 end-declarations

 initializations from 'c1bike.dat'
  DEM CNORM COVER CSTOCK CAP ISTOCK
 end-initializations

! Objective: minimize production cost
 Cost:= sum(t in TIMES) (CNORM*pnorm(t) + COVER*pover(t) + CSTOCK*store(t))

! Satisfy the demand for every period
 forall(t in TIMES)
  pnorm(t) + pover(t) + if(t>1, store(t-1), ISTOCK) = DEM(t) + store(t)

! Capacity limits on normal and overtime working hours per month
 forall(t in TIMES) do
  pnorm(t) <= CAP
  pover(t) <= 0.5*CAP
 end-do
```

```
! Solve the problem
 minimize(Cost)

end-model
```

In this implementation we use the inline `if` function to condense the balance constraints for the first and all following periods into a single constraint. For $t = 1$ the expression evaluates to

```
 pnorm(1) + pover(1) + ISTOCK = DEM(1) + store(1)
```

while for the rest of the periods we have

```
 pnorm(t) + pover(t) + store(t-1) = DEM(t) + store(t)
```

8.1.3 Results

The minimum cost for production and storage is € 11,247,000. The following table list the production plan for the coming year (in thousands) — its first line recalls the forecast monthly demand. In January to April and September to December the demands can be satisfied with production in normal hours. In April, 3000 bicycles need to be stored to satisfy the higher demand in May. In June to August overtime work is required to produce the required quantities.

Table 8.2: Quantities to produce and store in thousand units

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
Demand	30	15	15	25	33	40	45	45	26	14	25	30
Normal	28	15	15	28	30	30	30	30	26	14	25	30
Additional	–	–	–	–	–	10	15	15	–	–	–	–
Store	–	–	–	3	–	–	–	–	–	–	–	–

Note that in practice one might not like a solution like this: the inventory at the month end is almost always 0, so if there were a surge in demand (*e.g.* unusually good weather) the company would be in trouble.

8.2 Production of drinking glasses

The main activity of a company in northern France is the production of drinking glasses. It currently sells six different types (V1 to V6), that are produced in batches of 1000 glasses, and wishes to plan its production for the next 12 weeks. The batches may be incomplete (fewer than 1000 glasses). The demand in thousands for the 12 coming weeks and for every glass type is given in the following table.

Table 8.3: Demands for the planning period (batches of 1000 glasses)

Week	1	2	3	4	5	6	7	8	9	10	11	12
V1	20	22	18	35	17	19	23	20	29	30	28	32
V2	17	19	23	20	11	10	12	34	21	23	30	12
V3	18	35	17	10	9	21	23	15	10	0	13	17
V4	31	45	24	38	41	20	19	37	28	12	30	37
V5	23	20	23	15	10	22	18	30	28	7	15	10
V6	22	18	20	19	18	35	0	28	12	30	21	23

For every glass type the initial stock is known, as well as the required final stock level (in thousands). Per batch of every glass type, the production and storage costs in € are given, together with the required working time for workers and machines (in hours), and the required storage space (measured in numbers of trays).

The number of working hours of the personnel is limited to 390 hours per week, and the machines have a weekly capacity of 850 hours. Storage space for up to 1000 trays is available. Which quantities of the different glass types need to be produced in every period to minimize the total cost of production and storage?

Table 8.4: Data for the six glass types

	Production cost	Storage cost	Initial stock	Final stock	Time_{worker}	$\text{Time}_{machine}$	Storage space
V1	100	25	50	10	3	2	4
V2	80	28	20	10	3	1	5
V3	110	25	0	10	3	4	5
V4	90	27	15	10	2	8	6
V5	200	10	0	10	4	11	4
V6	140	20	10	10	4	9	9

8.2.1 Model formulation

This problem is a generalization of the bicycle problem in Section 8.1: we now have several products, each using the same resources.

To express the model in a simple way, we need to consider the time periods one after the other. Let *PRODS* be the set of products (glass types) and $WEEKS = \{1, \ldots, NT\}$ the set of time periods. We write DEM_{pt} for the demand for product p in time period t. These demands are given above in Table 8.3. We also have $CPROD_p$ and $CSTOCK_p$ the production and storage cost for glass type p. This cost is identical for all time periods, but it would be easy to model a different cost per time period by adding an index for the time period.

$TIMEW_p$ and $TIMEM_p$ denote the worker and machine times respectively required per unit of product p, and correspondingly, $SPACE_p$ the storage area. The initial stock $ISTOCK_p$ is given, as is the desired final stock level $FSTOCK_p$ per product. All these data are listed in Table 8.4 above. We write *CAPW* and *CAPM* for the capacities of workers and machines respectively, and *CAPS* for the capacity of the storage area.

To solve this problem, we need variables $produce_{pt}$ to represent the production of glass type p in time period t. The variables corresponding to the stock level of every product p at the end of period t are called $store_{pt}$. By convention, the initial stock level $ISTOCK_p$ may be considered as the stock level at the end of time period 0 and we use the notation $store_{p0}$ to simplify the formulation of the stock balance constraints (8.2.1):

$$\forall p \in PRODS, t \in WEEKS : store_{pt} = store_{p,t-1} + produce_{pt} - DEM_{pt} \tag{8.2.1}$$

These stock balance constraints state that the quantity $store_{pt}$ of product that is held in stock at the end of a time period t equals the stock level $store_{p,t-1}$ at the end of the preceding period plus the production $produce_{pt}$ of the time period t minus the demand DEM_{pt} of this time period.

The company wishes to have a certain amount of product in stock at the end of the planning period. These constraints on the final stock levels are expressed by the constraints (8.2.2):

$$\forall p \in PRODS : store_{p,NT} \geq FSTOCK_p \tag{8.2.2}$$

We now formulate the various capacity constraints for every time period. The following constraints guarantee that the capacity limits on manpower (8.2.3), machine time (8.2.4), and storage space (8.2.5) are kept:

$$\forall t \in WEEKS : \sum_{p \in PRODS} TIMEW_p \cdot produce_{pt} \leq CAPW \tag{8.2.3}$$

$$\forall t \in WEEKS : \sum_{p \in PRODS} TIMEM_p \cdot produce_{pt} \leq CAPM \tag{8.2.4}$$

$$\forall t \in WEEKS : \sum_{p \in PRODS} SPACE_p \cdot produce_{pt} \leq CAPS \tag{8.2.5}$$

We may now formulate the cost function that is to be minimized (8.2.6). This function is the sum of production and storage costs for all products and time periods.

$$\text{minimize} \sum_{p \in PRODS} \sum_{t \in WEEKS} \left(CPROD_p \cdot produce_{pt} + CSTORE_p \cdot store_{pt} \right) \tag{8.2.6}$$

We obtain the complete mathematical model by combining lines (8.2.1) to (8.2.6), to which we need to add the non-negativity constraints for the production variables (8.2.13) and for the stored quantities

(8.2.14).

$$\text{minimize} \sum_{p \in PRODS} \sum_{t \in WEEKS} \left(CPROD_p \cdot produce_{pt} + CSTORE_p \cdot store_{pt} \right) \tag{8.2.7}$$

$$\forall p \in PRODS, t \in WEEKS : store_{pt} = store_{p,t-1} + produce_{pt} - DEM_{pt} \tag{8.2.8}$$

$$\forall p \in PRODS : store_{p,NT} \geq FSTOCK_p \tag{8.2.9}$$

$$\forall t \in WEEKS : \sum_{p \in PRODS} TIMEW_p \cdot produce_{pt} \leq CAPW \tag{8.2.10}$$

$$\forall t \in WEEKS : \sum_{p \in PRODS} TIMEM_p \cdot produce_{pt} \leq CAPM \tag{8.2.11}$$

$$\forall t \in WEEKS : \sum_{p \in PRODS} SPACE_p \cdot produce_{pt} \leq CAPS \tag{8.2.12}$$

$$\forall p \in PRODS, t \in WEEKS : produce_{p,t} \geq 0 \tag{8.2.13}$$

$$\forall p \in PRODS, t \in WEEKS : store_{p,t} \geq 0 \tag{8.2.14}$$

8.2.2 Implementation

The mathematical model may be implemented with Mosel as follows.

```
model "C-2 Glass production"
 uses "mmxprs"

 declarations
  NT = 12                              ! Number of weeks in planning period
  WEEKS = 1..NT
  PRODS = 1.. 6                        ! Set of products

  CAPW,CAPM: integer                   ! Capacity of workers and machines
  CAPS: integer                        ! Storage capacity
  DEM: array(PRODS,WEEKS) of integer   ! Demand per product and per week
  CPROD: array(PRODS) of integer       ! Production cost per product
  CSTOCK: array(PRODS) of integer      ! Storage cost per product
  ISTOCK: array(PRODS) of integer      ! Initial stock levels
  FSTOCK: array(PRODS) of integer      ! Min. final stock levels
  TIMEW,TIMEM: array(PRODS) of integer ! Worker and machine time per unit
  SPACE: array(PRODS) of integer       ! Storage space required by products

  produce: array(PRODS,WEEKS) of mpvar ! Production of products per week
  store: array(PRODS,WEEKS) of mpvar   ! Amount stored at end of week
 end-declarations

 initializations from 'c2glass.dat'
  CAPW CAPM CAPS DEM CSTOCK CPROD ISTOCK FSTOCK TIMEW TIMEM SPACE
 end-initializations

! Objective: sum of production and storage costs
 Cost:=
  sum(p in PRODS, t in WEEKS) (CPROD(p)*produce(p,t) + CSTOCK(p)*store(p,t))

! Stock balances
 forall(p in PRODS, t in WEEKS)
  store(p,t) = if(t>1, store(p,t-1), ISTOCK(p)) + produce(p,t) - DEM(p,t)

! Final stock levels
 forall(p in PRODS) store(p,NT) >= FSTOCK(p)

! Capacity constraints
 forall(t in WEEKS) do
  sum(p in PRODS) TIMEW(p)*produce(p,t) <= CAPW     ! Workers
  sum(p in PRODS) TIMEM(p)*produce(p,t) <= CAPM     ! Machines
  sum(p in PRODS) SPACE(p)*store(p,t)   <= CAPS     ! Storage
 end-do

! Solve the problem
 minimize(Cost)

end-model
```

In the model formulation above as before we use the `if` function in the formulation of the stock balance constraints. It is also possible to employ the notation $store_{p0}$ for the initial stock levels as in the mathematical model. Since `store` is an array of variables, we still need to use the array `ISTOCK` for reading in the data from file and then fix the storage variables for time period 0 to these values (additional set of bound constraints):

```
 declarations
  store: array(PRODS,0..NT) of mpvar     ! Amount stored at end of week
 end-declarations

! Stock balances
 forall(p in PRODS, t in WEEKS)
  store(p,t) = store(p,t-1) + produce(p,t) - DEM(p,t)

! Fix the initial stock levels
 forall(p in PRODS) store(p,0) = ISTOCK(p)
```

In this model, it is easy to change the objective function, for instance to take into account only the production costs (first part of the objective) or the storage costs (second part of the objective)

Remark: in the implementation of this model, it may be tempting to define names like *prod* for the production variables or *PROD* for the set of products, but both these names cannot be used because `prod` is a keyword of the Mosel language (see the complete list of reserved words in Section 5.2.3).

8.2.3 Results

The solution of this problem gives us a total cost of € 185,899.

Table 8.5 displays the quantities of the different glass types to produce in every week of the planning period and the quantities held in stock at the end of every week. Some production and storage quantities are fractional but every unit corresponds to 1000 glasses. We may therefore round the results to three digits after the decimal point to obtain a result with integral numbers of glasses without modifying the total cost too much.

Taking for example the quantities to be produced in the first period: 8,760 glasses of type 1, 18,000 glasses of type 3, 16,000 glasses of type 4, 47,680 glasses of type 5, 12,000 glasses of type 6 and none of type 2 (the demand for this type is covered by the initial stock). When looking at the constraints in more detail, we find that the available manpower is fully used most of the time (in the first week, 351 hours are worked, in all other weeks the limit of 390 hours is reached) and the machines are used to their full capacity in certain time periods (weeks 1-3 and 5), but the available storage space is in excess of the actual needs.

Table 8.5: Quantities to produce and store of every glass type (in thousands)

	Week	1	2	3	4	5	6	7	8	9	10	11	12
1	Prod.	8.76	5.48	0.56	30.2	27.36	8.64	23	20	29	30	28	42
	Store	*38.76*	*22.24*	*4.8*	-	*10.36*	–	–	–	–	–	–	*10*
2	Prod.	0	16	23	20	11	10	12	34	21	23	30	22
	Store	*3*	–	–	–	–	–	–	–	–	–	–	*10*
3	Prod.	18	35	17	10	9	21	23	15	10	0	13	27
	Store	–	–	–	–	–	–	–	–	–	–	–	*10*
4	Prod.	16	45	24	38	41	20	19	37	28	12	30	47
	Store	–	–	–	–	–	–	–	–	–	–	–	*10*
5	Prod.	47.68	14.64	35.08	14.35	23.48	22.77	43.75	0	26.5	2.75	0	0
	Store	*24.68*	*19.32*	*31.4*	*30.75*	*44.23*	*45*	*70.75*	*40.75*	*39.25*	*35*	*20*	*10*
6	Prod.	12	18	20	19	18	35	0.75	27.25	12	49	29.25	5.75
	Store	–	–	–	–	–	–	*0.75*	–	–	*19*	*27.25*	*10*

As opposed to the bicycle problem in Section 8.1, in this problem final stock levels have been given. The specification of final stock levels is quite a tricky issue in production planning models. If one does not specify levels, then models will typically run stocks down to zero at the end of the planning horizon, a situation which will usually be unacceptable in practice. If the final stock levels are different from the initial stock levels, as in this example, then care must be taken in interpreting average costs, as some benefit has accrued if stock levels are being run down, and vice versa if stock is being increased.

8.3 Material Requirement Planning

The company Minorette produces two types of large toy lorries for children: blue removal vans and red tank lorries. Each of these lorries is assembled from thirteen items. See Figure 8.1 for the breakdown of components (also called a **Gozinto graph** or **Parts explosion**) and the following Table 8.6 for the prices of the components.

Table 8.6: Prices of components

Wheel	Steel bar	Bumper	Chassis	Cabin	Door window
€0.30	€1	€0.20	€0.80	€2.75	€0.10
Windscreen	**Blue container**	**Red tank**	**Blue motor**	**Red motor**	**Headlight**
€0.29	€2.60	€3	€1.65	€1.65	€0.15

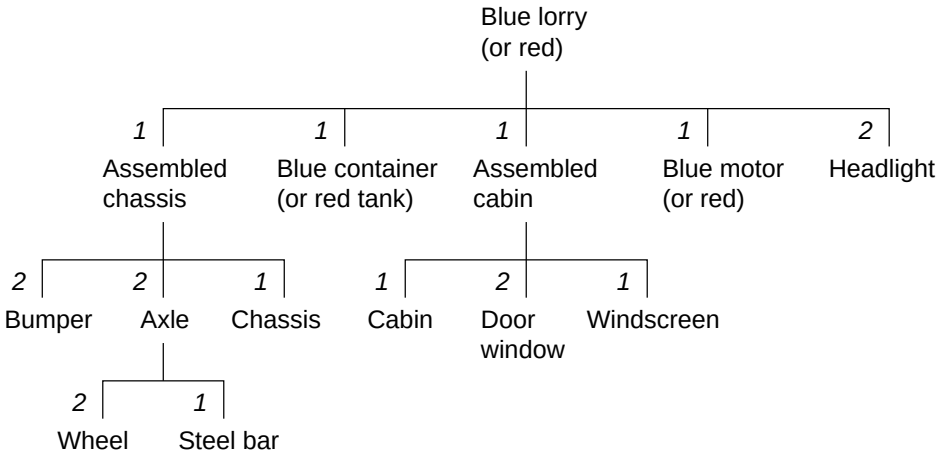

Figure 8.1: Breakdown of components (Gozinto graph)

The subsets (axles, chassis, blue or red cabin) may be assembled by the company using the components, or subcontracted. The following table lists the costs for assembling and for subcontracting these subsets, together with the capacities of the company. The assembly costs do not take into account the buying prices of their components.

Table 8.7: Subcontracting and assembly costs, assembly capacities

	Axle	Assembled chassis	Assembled cabin	Blue lorry	Red lorry
Subcontracting	€12.75	€30	€3	–	–
Assembly	€6.80	€3.55	€3.20	€2.20	€2.60
Capacity	600	4000	3000	4000	5000

For the next month, Minorette has the same demand forecast of 3000 pieces for the two types of lorries. At present, the company has no stock. Which quantities of the different items should Minorette buy or subcontract to satisfy the demand while minimizing the production cost?

8.3.1 Model formulation

Let *ITEMS* be the complete set of all products with the subsets *FINAL* of final products, *ASMBL* of products resulting from assembly, and *PREPROD* of products used in the assembly of others. We write REQ_{qp} for the requirement of component p in the assembly of q. For instance, for an axle we need two wheels and one steel bar, that is $REQ_{axle,wheel} = 2$ and $REQ_{axle,steelbar} = 1$.

Let $CBUY_p$ be the price paid when buying item p. Since subcontracting a product means buying it from its producer, we consider that this cost is the price for buying a preproduct that is not assembled and is the cost of subcontracting for the assembled pieces.

We write $CPROD_p$ for the cost of assembling a product p. The variables that we need to determine are the quantities to produce $produce_p$ of the assembled products p and those to buy (or subcontract) buy_p

of every preproduct p. This gives the following mathematical model:

$$\text{minimize} \sum_{p \in PREPROD} CBUY_p \cdot buy_p + \sum_{p \in ASMBL} CPROD_p \cdot produce_p \tag{8.3.1}$$

$$\forall p \in FINAL : produce_p \geq DEM_p \tag{8.3.2}$$

$$\forall p \in PREPROD : \tag{8.3.3}$$

$$p \in ASMBL : buy_p + produce_p \geq \sum_{q \in ASMBL} REQ_{qp} \cdot produce_q$$

$$p \notin ASMBL : buy_p \geq \sum_{q \in ASMBL} REQ_{qp} \cdot produce_q$$

$$\forall p \in ASMBL : produce_p \leq CAP_p \tag{8.3.4}$$

$$\forall p \in ASMBL : produce_p \in \mathbb{N} \tag{8.3.5}$$

$$\forall p \in PREPROD : buy_p \in \mathbb{N} \tag{8.3.6}$$

The objective function (8.3.1) minimizes the production cost, that is, the buying price or cost for subcontracting and the assembly cost. The constraints (8.3.2) indicate that the produced quantities of the final products (blue and red lorries) must be greater than or equal to the demand DEM_p. The constraints (8.3.3) guarantee that the total quantity of item p that is bought or produced is sufficient to produce all articles q that contain p.

The constraints (8.3.4) establish the limits on the assembly capacities CAP_p for every assembled product p. The constraints (8.3.5) and (8.3.6) are the integrality constraints for the variables.

Note that an alternative formulation to the model displayed above consists of defining buying and assembly costs for all items: preproducts that cannot be assembled receive the value 'infinity' as their assembly cost, and this value is also assigned as the buying price to the final products. With these costs, we may define variables $produce_p$ and buy_p for all products and in the objective function (1) and the constraints (3) simply sum over all variables — the 'infinity' cost values force the corresponding variables to take the value 0. Whilst easier to write down, this second formulation introduces large coefficient values into the model that typically lead to numerical instabilities and should therefore be avoided.

8.3.2 Implementation

The following Mosel program implements the mathematical model of lines (8.3.1) to (8.3.6). All data and index sets are defined in the data file and initialized dynamically when executing the program.

```
model "C-3 Toy production"
 uses "mmxprs"

 declarations
  ITEMS: set of string                ! Set of all products
  FINAL: set of string                ! Set of final products
  ASMBL: set of string                ! Set of assembled products
  PREPROD: set of string              ! Set of preproducts

  CAP: array(ASMBL) of integer        ! Capacity of assembly lines
  DEM: array(FINAL) of integer        ! Demand of lorries
  CPROD: array(ASMBL) of real         ! Assembly costs
  CBUY: array(ITEMS) of real          ! Purchase costs
  REQ: array(ASMBL,PREPROD) of integer ! Items req. for assembling a product
 end-declarations

 initializations from 'c3toy.dat'
  DEM CBUY REQ
  [CPROD, CAP] as 'ASSEMBLY'
 end-initializations

 finalize(ASMBL); finalize(PREPROD); finalize(FINAL); finalize(ITEMS)

 declarations
  produce: array(ASMBL) of mpvar      ! Quantity of items produced
  buy: array(PREPROD) of mpvar        ! Quantity of items bought
 end-declarations

! Objective: total costs
 Cost:=sum(p in PREPROD) CBUY(p)*buy(p) + sum(p in ASMBL) CPROD(p)*produce(p)
```

```
! Satisfy demands
 forall(p in FINAL) produce(p) >= DEM(p)

! Assembly balance
 forall(p in PREPROD) buy(p) + if(p in ASMBL, produce(p), 0) >=
                       sum(q in ASMBL) REQ(q,p)*produce(q)

! Limits on assembly capacity
 forall(p in ASMBL) produce(p) <= CAP(p)

 forall(p in PREPROD) buy(p) is_integer
 forall(p in ASMBL) produce(p) is_integer

! Solve the problem
 minimize(Cost)

end-model
```

As opposed to most other models presented in this book, in this implementation we do not fix the index sets directly in the Mosel program but initialize all data dynamically from file. We only define the variables once the data has been read in and the index sets have been finalized — that is, the arrays of variables are created with fixed sizes. If we defined the variables in the same `declarations` block as the data arrays, this array would be created as a **dynamic array**, just like the data arrays. But unlike the data arrays, in a dynamic array of variables every entry needs to be created explicitly, using the function `create`. It is therefore also possible to implement the Mosel program as shown below, though it should be noted that the above version with fixed-size arrays is slightly more efficient:

```
 declarations
  ITEMS: set of string                ! Set of all products
  FINAL: set of string                ! Set of final products
  ASMBL: set of string                ! Set of assembled products
  PREPROD: set of string              ! Set of pre-products

  CAP: array(ASMBL) of integer        ! Capacity of assembly lines
  DEM: array(FINAL) of integer        ! Demand of lorries
  CPROD: array(ASMBL) of real         ! Assembly costs
  CBUY: array(ITEMS) of real          ! Purchase costs
  REQ: array(ASMBL,PREPROD) of integer ! Items req. for assembling a product

  produce: array(ASMBL) of mpvar      ! Quantity of items produced
  buy: array(PREPROD) of mpvar        ! Quantity of items bought
 end-declarations

 initializations from 'c3toy.dat'
  DEM CBUY REQ
  [CPROD, CAP] as 'ASSEMBLY'
 end-initializations

 forall(p in ASMBL) create(produce(p))
 forall(p in PREPROD) create(buy(p))
```

In this example, the data of the arrays `CPROD` and `CAP` are contained in a single record `ASSEMBLY` and these two arrays are therefore read jointly using the keyword `as` to indicate the label of the record.

8.3.3 Results

The minimum total cost for satisfying the demand of toy lorries is € 238,365. The next two tables summarize the quantities of products that are bought, subcontracted, or assembled at the factory.

Table 8.8: Quantities of preproducts bought

Wheel	Steel bar	Bumper	Chassis	Cabin	Door window
1200	600	600	300	0	0
Windscreen	**Blue container**	**Red tank**	**Blue motor**	**Red motor**	**Headlight**
0	3000	3000	3000	3000	12000

Table 8.9: Production and subcontracting of assembled products

	Axle	Assembled chassis	Assembled cabin	Blue lorry	Red lorry
Produce	600	300	0	3000	3000
Subcontract	0	5700	6000	–	–

8.4 Planning the production of electronic components

To augment its competitiveness a small business wishes to improve the production of its best selling products. One of its main activities is the production of cards with microchips and electronic badges. The company also produces the components for these cards and badges. Good planning of the production of these components therefore constitutes a decisive success factor for the company. The demand for components is internal in this case and hence easy to anticipate.

For the next six months the production of four products with references X43-M1, X43-M2, Y54-N1, Y54-N2 is to be planned. The production of these components is sensitive to variations of the level of production, and every change leads to a non-negligible cost through controls and readjustments. The company therefore wishes to minimize the cost associated with these changes whilst also taking into account the production and storage costs.

The demand data per time period, the production and storage costs, and the initial and desired final stock levels for every product are listed in the following table. When the production level changes, readjustments of the machines and controls have to be carried out for the current month. The cost incurred is proportional to the increase or reduction of the production compared to the preceding month. The cost for an increase of the production is € 1 per unit but only € 0.50 for a decrease of the production level.

Table 8.10: Data for the four products

	Demands						Cost		Stock levels	
Month	**1**	**2**	**3**	**4**	**5**	**6**	**Production**	**Storage**	**Initial**	**Final**
X43-M1	1500	3000	2000	4000	2000	2500	20	0.4	10	50
X43-M2	1300	800	800	1000	1100	900	25	0.5	0	10
Y54-N1	2200	1500	2900	1800	1200	2100	10	0.3	0	10
Y54-N2	1400	1600	1500	1000	1100	1200	15	0.3	0	10

What is the production plan that minimizes the sum of costs incurred through changes of the production level, production and storage costs?

8.4.1 Model formulation

The mathematical model developed for the problem in Section 8.2 has many similarities to this model. We denote the variables and constants in the same way. Let *PRODS* be the set of components, $MONTHS = \{1, \ldots, NT\}$ the set of planning periods, DEM_{pt} the demand for product p in time period t, $CPROD_p$ and $CSTOCK_p$ respectively the cost of producing and storing one unit of product p, and $ISTOCK_p$ and $FSTOCK_p$ the initial and final stock levels of product p.

The variables $produce_{pt}$ and $store_{pt}$ represent respectively the quantity produced and the amount of product p in stock at the end of period t. With the convention $store_{p0} = ISTOCK_p$ the stock balance constraints are given by (8.4.1) and the final stock levels are guaranteed by constraints (8.4.2).

$$\forall p \in PRODS, t \in MONTHS : store_{pt} = store_{p,t-1} + produce_{pt} - DEM_{pt} \tag{8.4.1}$$

$$\forall p \in PRODS : store_{p,NT} \geq FSTOCK_p \tag{8.4.2}$$

To measure the changes to the level of production, we need to additional sets of variables, $reduce_t$ and add_t that represent the reduction or increase of the production in time period t. A change the the level of production is simply the difference between the total quantity produced in period t and the total quantity produced in period $t-1$. If this difference is positive, we have an increase in the production level, and a reduction otherwise. The change is expressed through the value of $add_t - reduce_t$.

Since the level of production cannot increase and reduce at the same time, one of the two variables is automatically at 0; whilst the other represents either the increase or the reduction of the production. This leads to the equation (8.4.3).

$$\forall t \in \{2, \ldots, NT\} : \sum_{p \in PRODS} produce_{pt} - \sum_{p \in PRODS} produce_{p,t-1} = add_t - reduce_t \tag{8.4.3}$$

The objective function is like the one of problem 8.2 with, in addition, the cost of changes to the production level. We write *CADD* and *CRED* for the cost of increasing and reducing the production respectively. These costs are proportional to the changes. They form the second part of the objective function (8.4.4).

$$\text{minimize} \quad \sum_{p \in PRODS} \sum_{t \in MONTHS} \left(CPROD_p \cdot produce_{pt} + CSTORE_p \cdot store_{pt} \right) + \sum_{t=2}^{NT} \left(CRED \cdot reduce_t + CADD \cdot add_t \right) \tag{8.4.4}$$

Note that there are no variables $reduce_1$ or add_1 because there is no change at the beginning of the planning period.

The complete linear program contains the lines (8.4.4) to (8.4.3), to which we need to add the non-negativity constraints (8.4.9) and (8.4.10) for all variables:

$$\text{minimize} \sum_{p \in PRODS} \sum_{t \in MONTHS} \left(CPROD_p \cdot produce_{pt} + CSTORE_p \cdot store_{pt} \right) + \sum_{t=2}^{NT} \left(CRED \cdot reduce_t + CADD \cdot add_t \right) \tag{8.4.5}$$

$$\forall p \in PRODS, t \in MONTHS : store_{pt} = store_{p,t-1} + produce_{pt} - DEM_{pt} \tag{8.4.6}$$

$$\forall p \in PRODS : store_{p,NT} \geq FSTOCK_p \tag{8.4.7}$$

$$\forall t \in \{2, \ldots, NT\} : \sum_{p \in PRODS} produce_{pt} - \sum_{p \in PRODS} produce_{p,t-1} = add_t - reduce_t \tag{8.4.8}$$

$$\forall t \in \{2, \ldots, NT\} : reduce_t \geq 0, add_t \geq 0 \tag{8.4.9}$$

$$\forall p \in PRODS, t \in MONTHS : produce_{p,t} \geq 0, store_{p,t} \geq 0 \tag{8.4.10}$$

8.4.2 Implementation

As we have noted, the model of this problem is close to the one of the problem in Section 8.2. A similar likeness can be observed between the two implementations with Mosel.

```
model "C-4 Electronic components"
 uses "mmxprs"

 declarations
  NT = 6                                  ! Number of time periods (months)
  MONTHS = 1..NT
  PRODS = 1..4                            ! Set of components

  DEM: array(PRODS,MONTHS) of integer     ! Demand of components per month
  CPROD: array(PRODS) of integer          ! Production costs
  CSTOCK: array(PRODS) of real            ! Storage costs
  CADD,CRED: real                         ! Cost of additional/reduced prod.
  ISTOCK,FSTOCK: array(PRODS) of integer ! Initial and final stock levels

  produce: array(PRODS,MONTHS) of mpvar  ! Quantities produced per month
  store: array(PRODS,MONTHS) of mpvar    ! Stock levels at end of every month
  reduce: array(MONTHS) of mpvar         ! Reduction of production per month
  add: array(MONTHS) of mpvar            ! Increase of production per month
 end-declarations

 initializations from 'c4compo.dat'
  DEM CPROD CSTOCK CADD CRED ISTOCK FSTOCK
 end-initializations

! Objective: total cost of production, storage, and change of prod. level
 Cost:= sum(p in PRODS, t in MONTHS) (CPROD(p)*produce(p,t) +
                                      CSTOCK(p)*store(p,t)) +
        sum(t in 2..NT) (CRED*reduce(t) + CADD*add(t))

! Stock balance constraints (satisfy demands)
 forall(p in PRODS, t in MONTHS)
  store(p,t) = if(t>1, store(p,t-1), ISTOCK(p)) + produce(p,t) - DEM(p,t)

! Changes to the level of production
```

```
 forall(t in 2..NT)
  sum(p in PRODS) (produce(p,t) - produce(p,t-1)) = add(t) - reduce(t)

! Guarantee final stock levels
forall(p in PRODS) store(p,NT) >= FSTOCK(p)

! Solve the problem
 minimize(Cost)

end-model
```

8.4.3 Results

When solving the problem we obtain a total cost of €683,929. The production level for the first four periods is 7060 units in total, in period 5 it decreases to 6100 units and remains at this level for the last period. The following table lists the corresponding production plan (there are several possible plans).

Table 8.11: Optimal production plan

	1	2	3	4	5	6
X43-M1	1490	3000	2000	4000	2000	2550
X43-M2	1300	800	800	1000	1100	910
Y54-N1	2150	1500	3640	1060	1200	2130
Y54-N2	2120	1760	620	1000	1800	510
Total	7060	7060	7060	7060	6100	6100

The stock levels are relatively low. With the exception of the last period where a final stock is required, products X43-M1 and X43-M2 are never held in stock. The product Y54-N1 is held in stock in periods 3 and 6 (740 and 30 units), and the product Y54-N2 is held in stock in the periods 1, 2, 5, and 6 with 720, 880, 700, and 10 units respectively.

8.5 Planning the production of fiberglass

A company produces fiberglass by the cubic meter and wishes to plan its production for the next six weeks. The production capacity is limited, and this limit takes a different value in every time period. The weekly demand is known for the entire planning period. The production and storage costs also take different values depending on the time period. All data are listed in the following table.

Table 8.12: Data per week

Week	Production capacity (m^3)	Demand (m^3)	Production cost (€/m^3)	Storage cost (€/m^3)
1	140	100	5	0.2
2	100	120	8	0.3
3	110	100	6	0.2
4	100	90	6	0.25
5	120	120	7	0.3
6	100	110	6	0.4

Which is the production plan that minimizes the total cost of production and storage?

8.5.1 Model formulation

This problem is not much different from the previous ones and it would be easy to solve it using one of the formulations presented earlier in this chapter. To introduce some variation, we are going to use a **transshipment flow formulation**. Other models of this family will be discussed in the context of transport problems in Chapter 10. To start, we draw a network that represents the problem (Figure 8.2). This network consists of six nodes for the production in every time period, and six more nodes for the demand in every time period. In the general case, we obtain a network of $2 \cdot NT$ nodes if NT is the number of time periods.

The production nodes form the upper half of the graph. The have been assigned the odd indices 1 to 11 corresponding to time periods 1 to 6. Every node n has a weight CAP_n that corresponds to the production

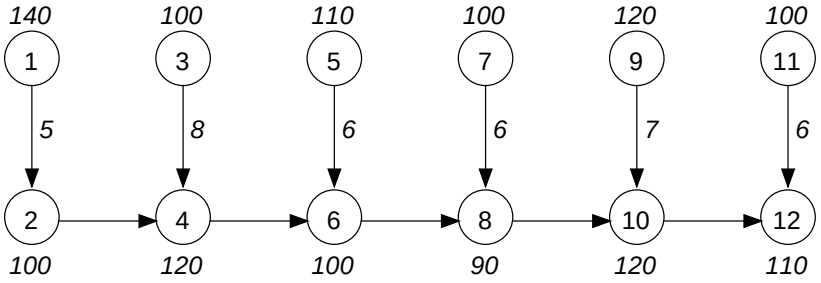

Figure 8.2: Representation of the network

capacity of the period. The demand nodes are in the lower half of the graph, with the even indices 2 to 12 equally corresponding to time periods 1 to 6. Every demand node n has a weight DEM_n that corresponds to the demand of the time period. An arc with the cost of the production in a period goes from every production node to the demand node of the same period. A second set of arcs weighted with the storage costs connects every demand node to the demand node of the following time period.

A production plan corresponds to a flow in this network. Let $flow_{mn}$ denote the flow on the arc (m, n). The flow on a vertical arc represents the quantity (in cubic meters) of fiberglass produced in the corresponding time period. The flow on a horizontal arc represents the quantity of product carried over to the next time period (stock level). The aim is to compute the minimum cost flow that satisfies the demands and does not exceed the production capacities. The resulting model is a transshipment problem because there are no capacities on the arcs. With $COST_{mn}$ the cost of an arc (m, n), we obtain the following objective function (total cost of the flow):

$$\text{minimize} \sum_{(m,n)\in ARCS} COST_{mn} \cdot flow_{mn} \tag{8.5.1}$$

For every time period, the amount carried over to the next period (stock) equals the stock at the beginning of the period plus the production of the period minus the demand of the period. This equation results in the constraint (8.5.2) for the first period since there is no initial stock. In other words, the quantity stored at the end of the period (flow on arc (2,4)) equals the amount produced in period 1 (flow on arc (1,2)) minus the demand DEM_2.

$$flow_{24} = flow_{12} - DEM_2 \tag{8.5.2}$$

The general stock balance equation (8.5.3) applies to all other periods but the last. For every even index n, the flow between $n-2$ and n represents the production in period $n/2$, whilst the flow between $n-1$ and n represents the stock carried over from the preceding period. The flow between n and $n+2$ represents the stock at the end of the current period. $LAST$ denotes the last node in the graph (= node with the largest sequence number).

$$\forall n = 4, \ldots, LAST - 2, n \text{ even} : flow_{n,n+2} = flow_{n-2,n} + flow_{n-1,n} - DEM_n \tag{8.5.3}$$

For the last period, we do not wish to create any final stock, so we obtain the constraint (8.5.4).

$$flow_{LAST-2,LAST} + flow_{LAST-1,LAST} - DEM_{LAST} = 0 \tag{8.5.4}$$

We also need to fulfill the constraints on the production capacity (8.5.5).

$$\forall n = 1, \ldots, LAST - 1, n \text{ odd} : flow_{n,n+1} \leq CAP_n \tag{8.5.5}$$

We obtain the following mathematical model. It is not required to state that the flow variables $flow_{mn}$ are integer since this property is automatically given in the optimal solution to the linear problem (this results from LP theory that is not covered in this book).

$$\text{minimize} \sum_{(m,n)\in ARCS} COST_{mn} \cdot flow_{mn} \tag{8.5.6}$$

$$flow_{24} = flow_{12} - DEM_2 \tag{8.5.7}$$

$$\forall n = 4, \ldots, LAST - 2, n \text{ even} : flow_{n,n+2} = flow_{n-2,n} + flow_{n-1,n} - DEM_n \tag{8.5.8}$$

$$flow_{LAST-2,LAST} + flow_{LAST-1,LAST} - DEM_{LAST} = 0 \tag{8.5.9}$$

$$\forall n = 1, \ldots, LAST - 1, n \text{ odd} : flow_{n,n+1} \leq CAP_n \tag{8.5.10}$$

$$\forall (m, n) \in ARCS : flow_{mn} \geq 0 \tag{8.5.11}$$

8.5.2 Implementation

The mathematical model leads to the following Mosel program. The demand and capacity data (= node **weights**) have been grouped into a single array *WEIGHT*, indexed by the node number. The graph with the cost of the arcs is encoded as a two-dimensional array *ARC*, an entry ARC_{mn} of which is defined if the arc (m, n) is in the network. The value of the entry corresponds to the cost of the arc. The array *ARC* is given in sparse format in the data file `c5fiber.dat`.

```
model "C-5 Fiberglass"
 uses "mmxprs"

 declarations
  NODES: range                          ! Production and demand nodes
                                        ! odd numbers: production capacities
                                        ! even numbers: demands

  ARC: array(NODES,NODES) of real       ! Cost of flow on arcs
  WEIGHT: array(NODES) of integer       ! Node weights (capacities/demand)

  flow: array(NODES,NODES) of mpvar     ! Flow on arcs
 end-declarations

 initializations from 'c5fiber.dat'
  ARC WEIGHT
 end-initializations

 forall(m,n in NODES | exists(ARC(m,n))) create(flow(m,n))

! Objective: total cost of production and storage
 Cost:= sum(m,n in NODES | exists(ARC(m,n))) ARC(m,n)*flow(m,n)

! Satisfy demands (flow balance constraints)
 forall(n in NODES | isodd(n)=FALSE)
  if(n<getlast(NODES), flow(n,n+2), 0) =
   if(n>2, flow(n-2,n), 0) + flow(n-1,n) - WEIGHT(n)

! Production capacities
 forall(n in NODES | isodd(n)) flow(n,n+1) <= WEIGHT(n)

! Solve the problem
 minimize(Cost)

end-model
```

In this Mosel program, the variables are defined as a dynamic array and therefore need to be created once their index range is known, that is, after the data defining the arcs of the network has been read from file.

We remind the reader that the use of the function `exists` allows Mosel to enumerate only the defined entries of a sparse array, provided that the index sets are named and are the same as those used for the declaration of the array.

The implementation above introduces two new functions: `isodd` indicates whether the integer value it is applied to is even or odd. `getlast` returns the last entry of an array (that is, the entry with the highest index value).

8.5.3 Results

The optimal solution calculated by the Mosel program is a total cost of €3,988 with the quantities of fiberglass to be produced and stored in every period shown in Table 8.13.

Table 8.13: Production and storage per week

Week	**1**	**2**	**3**	**4**	**5**	**6**
Production	140	80	110	100	110	100
Stock	40	0	10	20	10	0

8.6 Assignment of production batches to machines

Having determined a set of ten batches to be produced in the next period, a production manager is looking for the best assignment of these batches to the different machines in his workshop. The five available machines in the workshop may each process any of these batches, but since they are all models from different years of manufacture the speed with which they process the batches changes from one machine to another. Furthermore, due to maintenance periods and readjustments, every machine may only work a certain number of hours in the planning period. Table 8.14 lists the processing times of the production batches on all machines and the capacities of the machines.

Table 8.14: Processing durations and capacities (in hours)

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

The production cost of a batch depends on the machine that processes it. The hourly usage cost for every machine depends on its technology, its age, its consumption of consumables (such as electricity, machine oil) and the personnel required to operate it. These differences are amplified by the variation in the processing durations of a batch depending on the machine. Table 8.15 lists the production costs in thousand €. On which machine should each batch be executed if the production manager wishes to minimize the total cost of production?

Table 8.15: Production cost depending on the assignment (in k€)

	Production batches									
Machine	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	17	21	22	18	24	15	20	18	19	18
2	23	16	21	16	17	16	19	25	18	21
3	16	20	16	25	24	16	17	19	19	18
4	19	19	22	22	20	16	19	17	21	19
5	18	19	15	15	21	25	16	16	23	15

8.6.1 Model formulation

In its generic form, this problem is a **generalized assignment problem**. Let *MACH* be the set of machines and *PRODS* the set of batches to produce. We write DUR_{mp} for the processing duration of batch p on machine m and $COST_{mp}$ for the cost incurred if batch p is assigned to machine m. CAP_m denotes the maximum capacity (in hours) of the machine m. We introduce binary variables use_{mp} that take the value 1 if and only if the batch p is assigned to the machine m. The objective function (8.6.1) that is to be minimized is the total production cost.

$$\text{minimize} \sum_{m \in MACH} \sum_{p \in PRODS} COST_{mp} \cdot use_{mp} \tag{8.6.1}$$

As a first set of constraints, we need to ensure that every batch is assigned to a single machine. This is done with constraints (8.6.2). This type of constraint is typical for classical assignment problems, just like the problem of flight connections in Chapter 11.

$$\forall p \in PRODS : \sum_{m \in MACH} use_{mp} = 1 \tag{8.6.2}$$

Every machine may only work within its capacity limits, and this constraint is established by relations (8.6.3). In these constraints we sum up the durations of the batches assigned to the same machine and specify that this sum does not exceed the maximum total capacity of the machine. This type of constraint is a **knapsack constraint** (see Chapter 9).

$$\forall m \in MACH : \sum_{p \in PRODS} DUR_{mp} \cdot use_{mp} \leq CAP_m \tag{8.6.3}$$

This gives the following complete model:

$$\text{minimize} \sum_{m \in MACH} \sum_{p \in PRODS} COST_{mp} \cdot use_{mp} \tag{8.6.4}$$

$$\forall p \in PRODS : \sum_{m \in MACH} use_{mp} = 1 \tag{8.6.5}$$

$$\forall m \in MACH : \sum_{p \in PRODS} DUR_{mp} \cdot use_{mp} \leq CAP_m \tag{8.6.6}$$

$$\forall m \in MACH, \forall p \in PRODS : use_{mp} \in \{0, 1\} \tag{8.6.7}$$

8.6.2 Implementation

The conversion of such a compact mathematical model into a Mosel program is straightforward.

```
model "C-6 Machine assignment"
 uses "mmxprs"

 declarations
  MACH = 1..5                        ! Set of machines
  PRODS = 1..10                      ! Set of production batches

  CAP: array(MACH) of integer        ! Machine capacities
  DUR: array(MACH,PRODS) of integer  ! Production durations
  COST: array(MACH,PRODS) of integer ! Production costs

  use: array(MACH,PRODS) of mpvar    ! 1 if machine assigned to batch,
                                     ! 0 otherwise
 end-declarations

 initializations from 'c6assign.dat'
  DUR CAP COST
 end-initializations

! Objective: total production cost
 Cost:= sum(m in MACH, p in PRODS) COST(m,p)*use(m,p)

! Assign a single machine to every batch
 forall(p in PRODS) sum(m in MACH) use(m,p) = 1

! Limits on machine capacities
 forall(m in MACH) sum(p in PRODS) DUR(m,p)*use(m,p) <= CAP(m)

 forall(m in MACH, p in PRODS) use(m,p) is_binary

! Solve the problem
 minimize(Cost)

end-model
```

8.6.3 Results

After solving this problem, we obtain a total cost of €173k. The following table (there are several solutions with the same objective value) shows the assignment:

Table 8.16: Optimal assignment

Machine	Assigned batches	Total duration
1	1, 9	17
2	2, 5	18
3	3, 6, 8	25
4	7, 10	13
5	4	13

8.7 References and further material

There is plenty of literature about production planning, for instance research articles and monographs, such as Hax and Candea [HC84] and the one by Buffa and Taubert [BT79]. The recent book by Voß and Woodruff [VW02] on Supply Chain Management provides implementations of several typical problems with Mosel and other modeling languages.

Historically, planning problems have first been treated by taking into account the stock management almost exclusively. They were to answer the two questions: **how much to order** and **when to order**, to maintain the chosen stock level. The first model was the **Economic Order Quantity** model. It determines the quantity to produce according to a simple mathematical formula due to Wilson, that takes into account the cost of the order, the storage cost, and the demand of the period under consideration. The answer to the second question is based on the economic order quantity, the demand and the duration of the period. But this type of model is valid, among others, only if the demand is deterministic (known for the period) and does not change over time, if the parameters used for the calculations are also constant and do not depend on the quantity ordered, and if the resupply takes place instantaneously. These drawbacks have lead to **multi-period models** in which the time horizon is discretized over a sequence of time periods. By indexing the quantities with the time periods, these models are suitable for Linear Programming.

The simplest multi-period models have cost that are **proportional** to the quantities (this are usually production and storage costs). In other words, there are no **fix costs** like adjustments before starting the production of a batch. Constraints on limited resources (personnel, machine time etc.) are frequent. The problem of planning the production of bicycles in Section 8.1 is a one-product case that falls under this category. The production of fiberglass in Section 8.5 has costs that depend on the time period and is well modeled by flows. The glass production problem of Section 8.2 is a typical multi-product case. All these basic models work with a fixed personnel level (**fixed workforce models**). The first model of this type that appears in the literature is due to Bowman [Bow56].

Let us cite some extensions of multi-product models with proportional costs. Researchers early on studied **variable workforce models**, see for instance [HH60]. Later appear costs for additional hours, recruitment, licensing and training. The problem of electronic components in Section 8.4 also forms an interesting extension with cost caused by **changes of the production level**. A different level of generalization is reached by **multi-stage production planning** systems [JM74], [Can77].

In production planning, **lot sizing** problems form a class of their own that has not been mentioned in this chapter. These very hard problems are characterized by **setup costs** that apply every time the production of a product is restarted. These costs that are due to the division of a lot (batch) into several sub-lots are added to the usual production costs. The only easy case considers a single product and an unlimited production capacity. It is solved approximately with the heuristic of Silver and Meal [SPP98], and to optimality by a dynamic programming method of Wagner and Whitin [WW58]. Among the difficult extensions, let us cite the uncapacitated multi-product case [Kao79], and the cases with limited production capacities with the heuristics by Walker [Wal76] and the column generation method of Lasdon and Terjung [LT71].

The 70s have seen the birth of two major concepts that are 'en vogue' in production planning: the **MRP method** and **Just-in-Time**. MRP (**Material Requirement Planning**) was invented by Orlicky at IBM [Orl75]. This computerized stock management system has the advantage of taking into account the subsets, components and raw material required by a product. It produces a **master production schedule** that answers the two questions (how much and when to order). The use of Linear Programming for MRP is more recent; the problem of the production of toy lorries in Section 8.3 gives an example for this.

Just-in-Time (**JIT**) is more of a philosophy or a state of mind that aims at eliminating any waste of time or resource use. JIT reduces the stock levels and increases the frequency of renewal of the latter by trying to produce the desired product with the required quantity at the right moment. Establishing this way of functioning is not easy but often leads to major improvements. The 'philosophical' aspect is attractive and fairly common sense.

A problem that is rarely mentioned are the conflicts between different levels of production planning and scheduling. The aggregation of operations is required in planning to reduce the complexity but, most of all, because it is useless to go into too much detail with a planning horizon of several months for which the orders are not known precisely. Due to this aggregation, planning may unfortunately result in a production plan that proves to be infeasible for scheduling when the detailed constraints of the workshops are integrated into the model. Some recent methods allow the planner to avoid these problems [DPL94].

The generalized assignment problem of Section 8.6 is not really a production planning problem. We have placed it into this chapter because it appears when the batches calculated by the planning have to be

assigned to machines or workshops with different capacities and processing times. Instances of large size have to be treated with specialized techniques, like the tree-based method by Fisher et al. that uses the Lagrangian relaxation of the linear program [FJVW86].

Chapter 9
Loading and cutting problems

Loading or **packing** problems consist of placing objects of different sizes into containers of known capacities. Depending on the context, the objects and containers may be of a different nature: programs to store in memory, files to save onto a disk, boxes to be put into containers or onto shelves, *etc.* The term **packing** usually refers to non-fragmentable objects that may cohabitate in a single container (such as boxes in a lorry), whilst **loading** applies to fluid products (liquids and gas in tanks, powder in silos), that are fragmentable but may not be mixed in a single container.

In the closely related **cutting stock** problem the aim is to cut a set of objects from larger aggregates: cable segments from a bobbin, orders of glass, sheet metal or photographic film cut from large sheets, pieces cut from metal blocks, *etc.* A collective name for loading and cutting stock problems is **placement** problems.

In the simplest versions of these problems, the sizes and capacities only have a single dimension (weight of a box, size of a file in bytes, length of a cable, *etc.*) and the containers all have the same capacity. Obviously, problems exist with several types of containers and objects with two, three, or more dimensions. Frequently, these are geometric dimensions (sheets, boxes), or various physical properties (weight, volume).

Common objectives are to fill as much as possible of a container knowing that it is impossible to take everything, or to place objects into a minimum number of containers, or to balance the load. When cutting material, one aims to minimize the **trim loss** or to produce large offcuts that may be used for cutting further objects.

The first problem, presented in Section 9.1, is a loading problem in which the load of wagons has to be balanced. The problem of Section 9.2 concerns loading a barge. The problem of loading tanks in Section 9.3 consists of assigning petrol products to a set of tanks (some of which are partially filled) so as to maximize the number of tanks that is left empty. The subject of Section 9.4 is the problem of saving files onto disks. The two last problems deal with cutting rectangular sheets from large metal sheets (Section 9.5) and metal bars for desks (Section 9.6). These two are typical examples of **pattern selection** problems, that is, problems where the containers may only be cut according to a limited number of patterns.

9.1 Wagon load balancing

Three railway wagons with a carrying capacity of 100 quintals (1 quintal = 100 kg) have been reserved to transport sixteen boxes. The weight of the boxes in quintals is given in the following table. How shall the boxes be assigned to the wagons in order to keep to the limits on the maximum carrying capacity and to minimize the heaviest wagon load?

Table 9.1: Weight of boxes

Box	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Weight	34	6	8	17	16	5	13	21
Box	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
Weight	25	31	14	13	33	9	25	25

Before implementing a Mathematical Programming solution, one may wish to try to see whether it is possible to solve this problem instance with the following simple heuristic: until all boxes are distributed to the wagons we choose in turn the heaviest unassigned box and put it onto the wagon with the least load.

9.1.1 Model formulation

This **packing** problem is similar to **bin packing** problems (see also the tank loading problem in Section 9.3 and the backup of files in Section 9.4). Here the number of containers (wagons) is fixed, and the objective is to minimize the heaviest load. In bin packing problems, the capacity of the containers (bins) is fixed, but the objective is to minimize the number of bins used.

Let *BOXES* be the set of boxes, *WAGONS* the set of wagons, $WEIGHT_b$ the weight of box b and *WMAX* the maximum carrying load of a wagon. The assignment of the boxes to the wagons can be defined through binary variables $load_{bw}$ that take the value 1 if and only if box b is assigned to wagon w.

In this problem we wish to minimize the maximum load of the wagons. Such an objective is sometimes referred to as **minimax** objective. Similarly, there are **maximin** problems in which the objective is to maximize a minimum, like the assignment of personnel to workposts in Chapter 11. Minimax or maximin optimization problems are also called **bottleneck problems**. The procedure to represent a bottleneck criterion in a linear model is always the same:

- We define a non-negative variable *maxweight* to represent the maximum weight over all the wagon loads.
- Constraints are established to set *maxweight* as the upper bound on every wagon load.
- The objective function consists of minimizing *maxweight*.

By proceeding this way, in the optimal solution the minimization will force *maxweight* to take the value that corresponds to the weight of the heaviest wagon load. We derive the following 0-1 problem. The constraints (9.1.2) ensure that every box is assigned to a single wagon. The constraints (9.1.3) establish *maxweight* as the upper bound on the wagon loads. The objective (9.1.1) is to minimize *maxweight*. All variables are binary (9.1.4) with the exception of *maxweight* which is simply a non-negative real.

$$\text{minimize } maxweight \tag{9.1.1}$$

$$\forall b \in BOXES: \sum_{w \in WAGONS} load_{bw} = 1 \tag{9.1.2}$$

$$\forall b \in BOXES: \sum_{w \in WAGONS} WEIGHT_b \cdot load_{bw} \leq maxweight \tag{9.1.3}$$

$$\forall b \in BOXES, \forall w \in WAGONS: load_{bw} \in \{0, 1\} \tag{9.1.4}$$

$$maxweight \geq 0 \tag{9.1.5}$$

For two containers (wagons), it would be possible to define a much simpler mathematical model. The problem then turns into the task of partitioning the boxes into two subsets with weights as close as possible. In the best case, we obtain subsets of the weight $TOTALW / 2$ (with $TOTALW = \sum_{b \in BOXES} WEIGHT_b$ the total weight of all boxes). We may therefore reduce the problem to choosing boxes to place onto the first wagon in such a way as to obtain a load as close as possible (from below) to $TOTALW / 2$. For this purpose, we need to define binary variables $load_b$ with $load_b = 1$ if box b goes onto wagon 1, and $load_b = 0$ (or $1 - load_b = 1$) if the box is loaded onto wagon 2. The boxes that are not loaded onto wagon 1 go onto wagon 2. Thus, if the first wagon receives a load of weight $TOTALW / 2 - \Delta$, the second wagon has the load $TOTALW / 2 + \Delta$.

$$\text{maximize } \sum_{b \in BOXES} WEIGHT_b \cdot load_b \tag{9.1.6}$$

$$\sum_{b \in BOXES} WEIGHT_b \cdot load_b \leq TOTALW / 2 \tag{9.1.7}$$

$$\forall b \in BOXES: load_b \in \{0, 1\} \tag{9.1.8}$$

This slightly simpler problem where we try to fill as much as possible of a single container (here the first wagon with a capacity limited by $TOTALW / 2$) is called a **knapsack problem** (see also the barge loading problem in Section 9.2).

9.1.2 Implementation

The mathematical model of lines (9.1.1) – (9.1.4) is easily translated into a Mosel program. The role of the lower bound on the maximum weight variable that has been added to the model formulation is explained in the discussion of the results in the next section.

Before defining and solving the MP problem, we test whether we can find heuristically a distribution of loads to the wagons that fits the given capacity limit *WMAX*. The solution heuristic requires the boxes to be given in decreasing order of their weight. We therefore implement a sorting heuristic using a Shell sort method:

- First sort, by straight insertion, small groups of numbers.
- Next, combine several small groups and sort them (possibly repeat this step).
- Finally, sort the whole list of numbers.

The spacings between the numbers of groups sorted during each pass through the data are called the **increments**. A good choice is the sequence that can be generated by the recurrence $i_1 = 1$, $i_{k+1} = 3 \cdot i_k + 1$, $k = 1, 2, \ldots$

Our implementation of the sorting algorithm assumes that the indices of the array to sort have the values $1, \ldots, N$.

```
model "D-1 Wagon load balancing"
 uses "mmxprs"

 forward function solve_heur:real
 forward procedure shell_sort(A:array(range) of integer,
                              I:array(range) of integer)

 declarations
  BOXES = 1..16                     ! Set of boxes
  WAGONS = 1..3                     ! Set of wagons

  WEIGHT: array(BOXES) of integer   ! Box weights
  WMAX: integer                     ! Weight limit per wagon

  load: array(BOXES,WAGONS) of mpvar ! 1 if box loaded on wagon, 0 otherwise
  maxweight: mpvar                  ! Weight of the heaviest wagon load
 end-declarations

 initializations from 'd1wagon.dat'
  WEIGHT WMAX
 end-initializations

! Solve the problem heuristically and terminate the program if the
! heuristic solution is good enough
 if solveheur<=WMAX then
  writeln("Heuristic solution fits capacity limits")
  exit(0)
 end-if

! Every box into one wagon
 forall(b in BOXES) sum(w in WAGONS) load(b,w) = 1

! Limit the weight loaded into every wagon
 forall(w in WAGONS) sum(b in BOXES) WEIGHT(b)*load(b,w) <= maxweight

! Bounds on maximum weight
 maxweight <= WMAX
 maxweight >= ceil((sum(b in BOXES) WEIGHT(b))/3)

 forall(b in BOXES,w in WAGONS) load(b,w) is_binary

! Minimize the heaviest load
 minimize(maxweight)

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

! Heuristic solution: one at a time place the heaviest unassigned box
! onto the wagon with the least load
 function solve_heur:real
  declarations
   ORDERW: array(BOXES) of integer      ! Box indices
   Load: array(WAGONS,range) of integer ! Boxes loaded onto the wagons
   CurWeight: array(WAGONS) of integer  ! Current weight of wagon loads
   CurNum: array(WAGONS) of integer     ! Current number of boxes per wagon
  end-declarations
```

```
 ! Copy the box indices into array ORDERW and sort them in decreasing
 ! order of box weights
  forall(b in BOXES) ORDERW(b):=b
  shellsort(WEIGHT,ORDERW)

 ! Distribute the loads to the wagons using the LPT heuristic
  forall(b in BOXES) do
   v:=1                                ! Find wagon with the smallest load
   forall(w in WAGONS) v:=if(CurWeight(v)<CurWeight(w), v, w)
   CurNum(v)+=1                        ! Increase the counter of boxes on v
   Load(v,CurNum(v)):=ORDERW(b)        ! Add box to the wagon
   CurWeight(v)+=WEIGHT(ORDERW(b))     ! Update current weight of the wagon
  end-do

  returned:= max(w in WAGONS) CurWeight(w)  ! Return the solution value
 end-function

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

! Sort an array in decreasing order using a Shell sort method
! (The array to be sorted - first argument - remains unchanged,
! we reorder the array of indices in the second argument.)
 procedure shell_sort(A:array(range) of integer, I:array(range) of integer)
  N:=getsize(I)
  inc:=1                               ! Determine the starting increment
  repeat
   inc:=3*inc+1
  until (inc>N)

  repeat                               ! Loop over the partial sorts
   inc:=inc div 3
   forall(i in inc+1..N) do            ! Outer loop of straight insertion
    v:=I(i)
    j:=i
    while (A(I(j-inc))<A(v)) do        ! Inner loop of straight insertion
     I(j):=I(j-inc)
     j -= inc
     if j<=inc then break; end-if
    end-do
    I(j):= v
   end-do
  until (inc<=1)
 end-procedure

end-model
```

In this Mosel program for the first time we use a **function** (all subroutines defined so far were procedures, that is, they do not have a return value). What changes besides the keyword `function` instead of `procedure` is that the type of the return value (here: `real`) is declared in the subroutine prototype and the actual value is assigned to `returned` in the body of the function.

In the implementation of the Shell sort algorithm we also encounter some new features:

- The function `getsize` returns the size (= number of defined elements) of an array or set.
- The main sorting loop uses the three different types of loops available in Mosel: `repeat-until`, `forall`, and `while`. Like the `forall` loop, the `while` loop has an inline version that applies to a single statement and the version `while-do` to loop over a block of statements.
- The `break` statement can be used to interrupt one or several loops. In our case it stops the inner `while` loop. Since we are jumping out of a single loop, we could just as well write `break 1`. If we wrote `break 3`, the `break` would make the algorithm jump 3 loop levels higher, that is outside of the `repeat-until` loop.

Another feature introduced by this implementation is the Mosel function `exit` to terminate the execution of a model: if the heuristic solves the problem, there is no need to define the mathematical model and start the optimization algorithm.

9.1.3 Results

The heuristic returns a maximum load of 101 quintals (the two other wagons are loaded with 97 quintals each). This means that the heuristic solution does not satisfy the capacity limit of the wagons (100 quin-

tals) and we need to use an optimization algorithm to find out whether it is possible to transport all the boxes.

If we try running the mathematical model above without the lower bound constraint on the variable *maxweight*, the MIP search is not finished after several hundred thousand nodes (or several minutes of running time on a Pentium III PC) because the formulation is very weak (there are, for instance, many equivalent solutions, simply obtained through permuting the numbering of boxes): an optimal MIP solution is found quickly[1] but it takes a long time to prove its optimality.

A lower bound on the maximum weight is given by the ideal case that all wagons take the same load, that is $\sum_{b \in BOXES} WEIGHT_b / 3$. Since all box weights are integers and boxes cannot be fragmented, we can improve this bound by rounding it to the next larger integer (expressed through **ceil**). We have ceil $\left(\sum_{b \in BOXES} WEIGHT_b / 3\right)$ = ceil $\left(295 / 3\right)$ = ceil(98.33333) = 99. With this bound, the optimal LP solution has the value 99. Due to this lower bound, the MIP search stops as soon as an integer feasible solution with the value 99 is found. Since the boxes cannot be divided, it was not obvious that we would be able to find an integer solution within the carrying capacity of 100 quintals per wagon.

Table 9.2: Wagon loads

Wagon	Total weight	Boxes
1	99	2, 3, 7, 9, 12, 14, 16
2	98	4, 5, 6, 8, 11, 15
3	98	1, 10, 13

An alternative to the additional lower bound constraint on the maximum weight is to adapt the **cutoff** value used by the optimizer: whenever an integer feasible solution is found during the Branch and Bound search, the optimizer uses this value as the new, upper bound on the objective function. Since in this problem we can only have integer-valued solutions, we can deduce a value (close to) 1 better than this bound, to force the optimizer to look for the next integer solution. We add the following line to our Mosel program before calling the optimization algorithm:

```
setparam("XPRS_MIPADDCUTOFF",-0.99999)
```

Once the MIP search has found a solution with value 99, the optimizer sets the new upper bound to $99 - 0.99999 = 98.00001$. The optimal solution of the root LP (without the lower bound constraint on *maxweight*) is 98.33333 which is larger than this bound, and the search therefore terminates.

Table 9.2 displays one of many possible assignments of boxes to wagons with the objective value 99.

9.2 Barge loading

A shipper on the river Rhine owns a barge of carrying capacity 1500 m^3. Over time he has specialized in the transport of wheat. He has seven regular customers who load and unload practically at the same places. The shipper knows his costs of transport from long experience and according to his personal preferences has concluded agreements with his clients for the price charged to them for the transport of their wheat. The following table summarizes the information about the seven clients. Every client wishes to tranport a certain number of lots, deciding himself the size of his lots in m^3. The table lists the price charged by the shipper for every transported lot. The last column of the table contains the cost incurred by the shipper per transported m^3. This cost differs depending on the distance covered.

Table 9.3: Lots to transport

Client	Available quantity (no. of lots)	Lot size (in m^3)	Price per lot (in €)	Transport cost (in €/m^3)
1	12	10	1000	80
2	31	8	600	70
3	20	6	600	85
4	25	9	800	80
5	50	15	1200	73
6	40	10	800	70
7	60	12	1100	80

[1] The reader is reminded that it is possible to visualize the output of the optimizer by adding the line `setparam("XPRS_VERBOSE",true)` before the optimization statement. In Xpress-IVE, the tree search can be visualized directly.

The objective of the shipper is to maximize his profit from transporting the wheat with lots that may be divided.

Question 1: As a first step, assuming that every client has an unlimited quantity of wheat, which clients' wheat should be transported?

Question 2: If in addition the actual availability of wheat lots from the customers is taken into account, which strategy should the shipper adopt?

Question 3: What happens if the lots cannot be divided?

9.2.1 Model formulation

The models for all three questions are given in this subsection. In each case, the objective is to maximize the shipper's profit. Let CAP be the total carrying capacity of the barge. We write $CLIENTS$ for the set of clients, $AVAIL_c$ for the number of lots available from a client c, $SIZE_c$ the corresponding lot size, $PRICE_c$ the price charged to this customer, and $COST_c$ the transport cost per m^3 incurred by the shipper. With the given data, we calculate the shipper's profit $PROF_c$ per transported lot of client c. This profit is obtained easily via equation (9.2.1): from the price charged to a customer we need to deduct the transport cost for a lot stemming from this customer.

$$\forall c \in CLIENTS : PROF_c = PRICE_c - COST_c \cdot SIZE_c \tag{9.2.1}$$

The shipper's profit per transported lot of every client calculated through this equation is summarized in the following table.

Table 9.4: Profit per lot

Client	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Profit/lot (in €)	200	40	90	80	105	100	140
Profit/m^3 (in €)	20	5	15	8.8889	7	10	11.6667

We introduce the decision variables $load_c$ whose value is the number of lots transported for client c. Since the lots may be divided, these variables may take fractional values. The model only has a single constraint, namely to limit the volume of wheat transported by the barge (constraints (9.2.3)). Using the profit data calculated above, we may now easily express the objective function (9.2.2).

$$\text{maximize} \sum_{c \in CLIENTS} PROF_c \cdot load_c \tag{9.2.2}$$

$$\sum_{c \in CLIENTS} SIZE_c \cdot load_c \leq CAP \tag{9.2.3}$$

$$\forall c \in CLIENTS : load_c \geq 0 \tag{9.2.4}$$

If we have to take into account the number of lots available from each client, we simply have to add the bound constraints (9.2.5) that state that the transported number of lots $load_c$ is within the limit of availability.

$$\forall c \in CLIENTS : load_c \leq AVAIL_c \tag{9.2.5}$$

In the last case, the lots may not be fragmented, that is, the variables $load_c$ may only take integer values. We therefore have to add the following constraints (9.2.6) to the model.

$$\forall c \in CLIENTS : load_c \in \mathbb{N} \tag{9.2.6}$$

9.2.2 Implementation

To simplify the statement of the objective function, in the following Mosel implementation of the mathematical model the profit per client is calculated separately, as is done in the previous section. The resulting model is very simple because it contains only a single constraint. The additional constraints are progressively added to the problem definition. After every optimization run we call the procedure `print_sol` to display the results.

```
model "D-2 Ship loading"
 uses "mmxprs"

 forward procedure print_sol(num:integer)
```

```
 declarations
  CLIENTS = 1..7                    ! Set of clients

  AVAIL: array(CLIENTS) of integer  ! Number of lots per client
  SIZE: array(CLIENTS) of integer   ! Lot sizes
  PRICE: array(CLIENTS) of integer  ! Prices charged to clients
  COST: array(CLIENTS) of integer   ! Cost per client
  PROF: array(CLIENTS) of integer   ! Profit per client
  CAP: integer                      ! Capacity of the ship

  load: array(CLIENTS) of mpvar     ! Lots taken from clients
 end-declarations

 initializations from 'd2ship.dat'
  AVAIL SIZE PRICE COST CAP
 end-initializations

 forall(c in CLIENTS) PROF(c):= PRICE(c) - COST(c)*SIZE(c)

 Profit:= sum(c in CLIENTS) PROF(c)*load(c)

! Limit on the capacity of the ship
 sum(c in CLIENTS) SIZE(c)*load(c) <= CAP

! Problem 1: unlimited availability of lots at clients
 maximize(Profit)
 print_sol(1)

! Problem 2: limits on availability of lots at clients
 forall(c in CLIENTS) load(c) <= AVAIL(c)

 maximize(Profit)
 print_sol(2)

! Problem 3: lots must be integer
 forall(c in CLIENTS) load(c) is_integer

 maximize(Profit)
 print_sol(3)

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

! Solution printing
 procedure print_sol(num:integer)
  writeln("Problem ", num, ": profit: ", getobjval)
  forall(c in CLIENTS)
   write( if(getsol(load(c))>0 , " " + c + ":" + getsol(load(c)), ""))
  writeln
 end-procedure

end-model
```

9.2.3 Results

We examine the results obtained for the three questions in turn. In the first case, the optimizer returns a total profit of €30,000 and the only load taken are 150 lots from client 1. The other clients have to look for a different transport opportunity. This answer is quite obvious: the barge should be filled with the wheat from the client that provides the highest profit for the space it occupies (ratio $PROF_c$ / $SIZE_c$, values displayed in the second line of Table 9.4).

But in reality, the available quantities of wheat are limited. When we solve this second, more constrained problem, the value of the objective function decreases to €17,844.44. The transported quantities belong to five clients as shown in the table below. In this case, we are also able to deduce the solution through logic. It suffices to load the hold starting with the client who provides the best profit-per-volume ratio ($PROF_c$ / $SIZE_c$). We continue to load the hold with the wheat from the second best client (in the order of decreasing profit-per-volume ratio) and so on, until the hold is full. The process stops when the hold is full, so the last lot is likely to be incomplete.

In our example, the order of clients according to the decreasing profit-per-volume ratio is 1, 3, 7, 6, 4, 5, and 2. All wheat of clients 1,3,7, and 6 is taken, filling 1360 m^3 of the hold. The remaining 140 m^3 are filled with wheat from client 4, that is 15.5556 of his lots.

Table 9.5: Transported lots

Client	1	2	3	4	5	6	7
Question 1	150	0	0	0	0	0	0
Question 2	12	0	20	15.5556	0	40	60
Question 3	12	0	20	15	1	39	60

If the lots are forced to take integer values only (question 3), the solution is similar to the previous one, with a few small changes. The transported quantities for clients 1, 3, and 7 remain the same. Only 15 lots of client 4 and 39 of client 6 are taken and in addition 1 lot of client 5. The value of the objective function correspondingly decreases to € 17,805. Note that in the general case, there may be large differences between the lots chosen in the cases of fragmentable or integer lots. This type of problem is called a **knapsack problem**.

9.3 Tank loading

Five tanker ships have arrived at a chemical factory. They are carrying loads of liquid products that must not be mixed: 1200 tonnes of Benzol, 700 tonnes of Butanol, 1000 tonnes of Propanol, 450 tonnes of Styrene, and 1200 tonnes of THF. Nine tanks of different capacities are available on site. Some of them are already partially filled with a liquid. The following table lists the characteristics of the tanks (in tonnes). Into which tanks should the ships be unloaded (question 1) to maximize the capacity of the tanks that remain unused, or (question 2) to maximize the number of tanks that remain free?

Table 9.6: Characteristics of tanks

Tank	1	2	3	4	5	6	7	8	9
Capacity	500	400	400	600	600	900	800	800	800
Current product	–	Benzol	–	–	–	–	THF	–	–
Quantity	0	100	0	0	0	0	300	0	0

9.3.1 Model formulation

Let *TANKS* be the set of tanks and *LIQ* the set of liquid products. We write ARR_l for the quantity of liquid l that is about to arrive at the factory. CAP_t is the capacity of tank t, which initially contains a quantity $QINIT_t$ of liquid type $TINIT_t$ (this type is only defined if $QINIT_t > 0$). All quantities are in tonnes. The model formulation relies on the following property: it is possible to fill the tanks optimally by giving priority to the tanks that already contain a liquid l for unloading the corresponding liquid before it is filled into any empty tanks.

To understand why this correct, assume that a quantity x of liquid l is filled into an empty tank a, and another tank b already contains liquid l with a remaining capacity of $y > 0$. If $x > y$, we do not make any changes to the empty tanks (concerning their number or total capacity) if we first fill b before filling $x - y$ into a. If $x \leq y$, we save tank a by filling x into tank b. We therefore obtain a solution that is at least as good by giving priority to b until this tank is full or all of liquid l has been unloaded.

After the prioritized filling of the partially filled tanks, our problem remains to fill the empty tanks with quantities $REST_l$ defined by the relations (9.3.1): these are the quantities of product that remain once the capacities of the partially filled tanks are exhausted. As for the rest of the data, we suppose that the values $REST_l$ are larger than zero, otherwise it suffices to remove from the problem the tanks that are completely filled and the liquids that have been entirely unloaded.

$$\forall l \in LIQ : REST_l = ARR_l - \sum_{\substack{t \in TANKS \\ TINIT_t = l}} (CAP_t - QINIT_t) \tag{9.3.1}$$

The model can then be formulated in a straightforward manner using binary variables $load_{lt}$ that are defined only for tanks t that are empty after the prioritized filling (9.3.5). A variable $load_{lt}$ is 1 if the liquid l is unloaded into the tank t. The constraints (9.3.4) guarantee that every empty tank is filled with at most one liquid. The constraints (9.3.3) ensure that the set of tanks that are filled with the liquid l have a sufficiently large total capacity. The objective function (9.3.2) minimizes the total capacity of the tanks that are used and so answers the first question, namely maximize the total capacity of the tanks

remaining empty. To maximize the number of unused tanks, it suffices to remove the coefficients CAP_t from the sum in (9.3.2).

$$\text{minimize} \sum_{l \in LIQ} \sum_{\substack{t \in TANKS \\ QINIT_t = 0}} CAP_t \cdot load_{lt} \tag{9.3.2}$$

$$\forall l \in LIQ : \sum_{\substack{t \in TANKS \\ QINIT_t = 0}} CAP_t \cdot load_{lt} \geq REST_l \tag{9.3.3}$$

$$\forall t \in TANKS, QINIT_t = 0 : \sum_{l \in LIQ} load_{lt} \leq 1 \tag{9.3.4}$$

$$\forall l \in LIQ, t \in TANKS, QINIT_t = 0 : load_{lt} \in \{0, 1\} \tag{9.3.5}$$

9.3.2 Implementation

The following Mosel program implements the mathematical model developed in the previous section. Since the $load_{lt}$ variables are only created for the tanks that are entirely empty at the beginning, there is no need to test the condition $QINIT_t = 0$ in the sums over these variables.

```
model "D-3 Tank loading"
 uses "mmxprs"

 forward procedure print_sol

 declarations
  TANKS: range                        ! Set of tanks
  LIQ: set of string                  ! Set of liquids

  CAP: array(TANKS) of integer        ! Tank capacities
  TINIT: array(TANKS) of string       ! Initial tank contents type
  QINIT: array(TANKS) of integer      ! Quantity of initial contents
  ARR: array(LIQ) of integer          ! Arriving quantities of chemicals
  REST: array(LIQ) of integer         ! Rest after filling part. filled tanks

  load: array(LIQ,TANKS) of mpvar     ! 1 if liquid loaded into tank,
                                      ! 0 otherwise
 end-declarations

 initializations from 'd3tanks.dat'
  CAP ARR
  [TINIT, QINIT] as 'FILLED'
 end-initializations

 finalize(LIQ)

 forall(t in TANKS | QINIT(t)=0, l in LIQ) do
  create(load(l,t))
  load(l,t) is_binary
 end-do

! Complete the initially partially filled tanks and calculate the remaining
! quantities of liquids
 forall(l in LIQ)
  REST(l):= ARR(l) - sum(t in TANKS | TINIT(t)=l) (CAP(t)-QINIT(t))

! Objective 1: total tank capacity used
 TankUse:= sum(l in LIQ, t in TANKS) CAP(t)*load(l,t)

! Objective 2: number of tanks used
 TankNum:= sum(l in LIQ, t in TANKS) load(l,t)

! Do not mix different liquids
 forall(t in TANKS) sum(l in LIQ) load(l,t) <= 1

! Load the empty tanks within their capacity limits
 forall(l in LIQ) sum(t in TANKS) CAP(t)*load(l,t) >= REST(l)

! Solve the problem with objective 1
 minimize(TankUse)
```

```
 print_sol

! Solve the problem with objective 2
 minimize(TankNum)
 print_sol

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

! Solution printing
 procedure print_sol
  writeln("Used capacity: ", getsol(TankUse) +
                             sum(t in TANKS | QINIT(t)>0) CAP(t),
          " Capacity of empty tanks: ", sum(t in TANKS) CAP(t) -
                                        getsol(TankUse) -
                                        sum(t in TANKS | QINIT(t)>0) CAP(t))
  writeln("Number of tanks used: ", getsol(TankNum) +
                                    sum(t in TANKS | QINIT(t)>0) 1)
  forall(t in TANKS)
   if(QINIT(t)=0) then
    write(t, ": ")
    forall(l in LIQ) write( if(getsol(load(l,t))>0 , l, ""))
    writeln
   else
    writeln(t, ": ", TINIT(t))
   end-if
 end-procedure

end-model
```

9.3.3 Results

When evaluating the results, we need to take into account that our model only works with the tanks that are completely empty initially: the capacities (or for objective 2 the number) of the tanks that are partially filled in the beginning need to be added to the objective values calculated by the program.

For the first objective, we obtain a total capacity used of 5200 tonnes, tank 4 with a capacity of 600 tonnes remains empty (there are several equivalent solutions). Table 9.7 shows the distribution of chemicals to the other tanks and the unused capacity possibly remaining if these tanks are not completely filled.

Table 9.7: Optimal tank filling

Product	Tanks	Remaining capacity
Benzol	2, 6	0
Butanol	9	100
Propanol	3, 5	0
Styrene	1	50
THF	7, 8	100

In the graphical representation of the solution in Figure 9.1 the unused tank capacities are represented by grey-shaded areas and the initial fill heights are indicated with dashed lines.

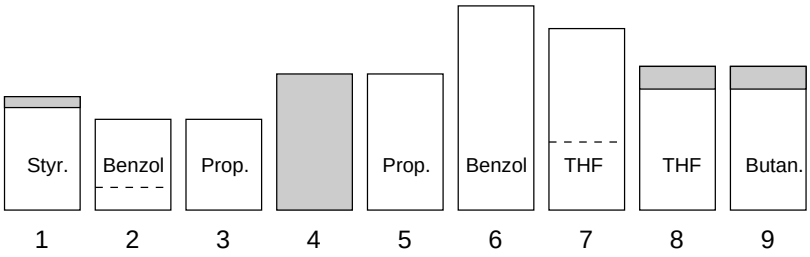

Figure 9.1: Optimal tank filling

In total, 8 tanks are used in the optimal solution for the first objective. After solving the problem with the second objective, we find that 8 is indeed the smallest number of tanks that may be used for unloading all liquids.

9.4 Backing up files

Before leaving on holiday, you wish to backup your most important files onto floppy disks. You have got empty disks of 1.44Mb capacity. The sixteen files you would like to save have the following sizes: 46kb, 55kb, 62kb, 87kb, 108kb, 114kb, 137kb, 164kb, 253kb, 364kb, 372kb, 388kb, 406kb, 432kb, 461kb, and 851kb.

Assuming that you do not have any program at hand to compress the files and that you have got a sufficient number of floppy disks to save everything, how should the files be distributed in order to minimize the number of floppy disks used?

9.4.1 Model formulation

Let *FILES* be the set of files to backup and $DISKS = \{1, \ldots, ND\}$ the set of floppy disks. Let *CAP* be the capacity of a disk and $SIZE_f$ the size of file f in kbyte. We use variables $save_{fd}$ that take the value 1 if file f is saved to disk d, and 0 in all other cases. We also define a variable use_d for every disk d that takes the value 1 if any files are saved to this disk and 0 otherwise.

The objective is to minimize the number of disks that are used, which corresponds to minimizing the sum of variables use_d, subject to the constraints:

a) A file must be saved onto a single disk. This constraint corresponds to the relation (9.4.1).

$$\forall f \in FILES : \sum_{d \in DISKS} save_{fd} = 1 \tag{9.4.1}$$

b) The capacity of disks is limited, which translates into the constraints (9.4.2).

$$\forall d \in DISKS : \sum_{f \in FILES} SIZE_f \cdot save_{fd} \leq CAP \cdot use_d \tag{9.4.2}$$

The variable use_d on the right side of this constraint links the variables $save_{fd}$ and use_d: if $save_{fd}$ has the value 1, then the file f is saved onto the disk d and hence, the disk d is used and variable use_d needs to take the value 1. The relation (9.4.2) forces use_d to take the value 1 if at least one variable $save_{fd}$ ($f \in FILES$) has the value 1. We obtain the following MIP model:

$$\text{minimize} \sum_{d \in DISKS} use_d \tag{9.4.3}$$

$$\forall f \in FILES : \sum_{d \in DISKS} save_{fd} = 1 \tag{9.4.4}$$

$$\forall d \in DISKS : \sum_{f \in FILES} SIZE_f \cdot save_{fd} \leq CAP \cdot use_d \tag{9.4.5}$$

$$\forall d \in DISKS, f \in FILES : save_{fd} \in \{0, 1\} \tag{9.4.6}$$

$$\forall d \in DISKS : use_d \in \{0, 1\} \tag{9.4.7}$$

It is possible to model this problem with fewer variables: we keep the Boolean variables $save_{fd}$ from the preceding model (but not the variables use_d) and introduce an additional variable *diskuse* that corresponds to the number of disks that are used. We obtain a second model formulation as follows:

$$\text{minimize } diskuse \tag{9.4.8}$$

$$\forall f \in FILES : diskuse \geq \sum_{d \in DISKS} d \cdot save_{fd} \tag{9.4.9}$$

$$\forall f \in FILES : \sum_{d \in DISKS} save_{fd} = 1 \tag{9.4.10}$$

$$\forall d \in DISKS : \sum_{f \in FILES} SIZE_f \cdot save_{fd} \leq CAP \tag{9.4.11}$$

$$\forall d \in DISKS, f \in FILES : save_{fd} \in \{0, 1\} \tag{9.4.12}$$

$$diskuse \geq 0 \tag{9.4.13}$$

The objective function is very simple in this case (9.4.8). We suppose that the disks are filled starting with the disk numbered $d = 1$, then disk number 2, and so on. Due to the constraints (9.4.1) or (9.4.10), the

disk that contains the file f has the index k that may be calculated through the relation (9.4.14).

$$k = \sum_{d \in DISKS} d \cdot save_{fd} \tag{9.4.14}$$

The value of *diskuse* must be at least as large as the highest index number of the disks that are used, hence the constraints (9.4.9). The constraints (9.4.10), that indicate that a file must be stored onto a single disk, are identical to the constraints (9.4.1) of the previous model. The capacity limits are established through the constraints (9.4.11).

Finally, all variables $save_{fd}$ must be binaries and *diskuse* non-negative (it is not necessary to define this variable explicitly as integer because this constraints will automatically be satisfied in the optimal solution). In this model, the minimization will reduce the value of *diskuse* as much as possible and backup the files in such a way that the index value of the largest disk becomes as small as possible. This model has *ND* Boolean variables fewer than the previous one.

9.4.2 Implementation

The following Mosel program implements the second mathematical model (lines (9.4.8) to (9.4.13)). Note that based on the given file sizes we calculate heuristically an upper bound on the number of disks that we may need for backing up all the files: we divide the sum of all file sizes by the capacity of a floppy disk, round this value to the next larger integer. The resulting value may be used as an upper bound in cases like the given data set where most of the files that need to be saved are very small compared to the capacity of a disk. The only bound value for the number of disks that is save in the general case is the number of files that need to be saved. (Consider, for instance, the case of 5 files of size 0.75Mb: $5 \cdot 0.75 = 3.75$, dividing this by 1.44 and rounding the result to the next larger integer we obtain 3, but we actually need 5 disks since only a single file fits onto every disk.)

```
model "D-4 Bin packing"
 uses "mmxprs"

 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
 end-declarations

 initializations from '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

 declarations
  save: array(FILES,DISKS) of mpvar  ! 1 if file saved on disk, 0 otherwise
  diskuse: mpvar                     ! Number of disks used
 end-declarations

! Limit the number of disks used
 forall(f in FILES) diskuse >= sum(d in DISKS) d*save(f,d)

! Every file onto a single disk
 forall(f in FILES) sum(d in DISKS) save(f,d) = 1

! Capacity limit of disks
 forall(d in DISKS) sum(f in FILES) SIZE(f)*save(f,d) <= CAP

 forall(d in DISKS,f in FILES) save(f,d) is_binary

! Minimize the total number of disks used
 minimize(diskuse)

end-model
```

9.4.3 Results

In the optimal solution, three disks are used. The files may be distributed to the disks as shown in the following table (there are several possible solutions).

Table 9.8: 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

In problems like this, where the objective function is very weak and we have essentially a feasibility problem, it may speed the Branch and Bound search to break the symmetry of the problem. The files have been numbered in an arbitrary way, as have the disks, so there is no loss of generality if we fix one assignment, for example assigning the biggest file to the first disk. Such **symmetry breaking** devices can be quite useful in combinatorial problems, as they sometimes prevent the LP relaxation from taking fractional combinations of solutions that would otherwise be integer feasible.

9.5 Cutting sheet metal

A sheet metal workshop cuts pieces of sheet metal from large rectangular sheets of 48 decimeters $\times$ 96 decimeters (dm). It has received an order for 8 rectangular pieces of 36 dm $\times$ 50 dm, 13 sheets of 24 dm $\times$ 36 dm, 5 sheets of 20 dm $\times$ 60 dm, and 15 sheets of 18 dm $\times$ 30 dm. Theses pieces of sheet metal need to be cut from the available large pieces. How can this order by satisfied by using the least number of large sheets?

9.5.1 Model formulation

In this type of highly combinatorial problem, one exploits the fact that only a relatively small number of combinations of rectangles can be cut from the large sheets. These cutting patterns can easily be enumerated; Figure 9.2 represents the sixteen subsets that can be found. The sets shown are maximal subsets, that is, no further small rectangle of the order may be added to them. The Figure only displays one of several possible arrangements of every pattern.

Let *SIZES* be the set of rectangles of different sizes of the order, and *PATTERNS* the set of cutting patterns represented in the graphic. For every size, the demand DEM_s is given. Every pattern has an associated cost $COST_p$ ($COST_p$ = 1 if we simply wish to minimize the number of large sheets used). The composition of the cutting patterns is given by the matrix CUT_{sp} defined by Table 9.9.

Table 9.9: Summary of cutting patterns

Pattern	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
36 x 50	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
24 x 36	2	1	0	2	1	0	3	2	1	0	5	4	3	2	1	0
20 x 60	0	0	0	2	2	2	1	1	1	1	0	0	0	0	0	0
18 x 30	0	1	3	0	1	3	0	2	3	5	0	1	3	5	6	8

Once having finished the tedious work of enumerating the cutting patterns, writing the mathematical model is easy. We need to calculate the number of sheets to cut with every pattern in order to produce all ordered rectangular pieces whilst minimizing the total number of sheets used.

We introduce integer variables use_p (9.5.3) that denote the number of times a cutting pattern p is used. The constraints (9.5.2) indicate that the number of rectangles of every type needs to satisfy the demand. The objective function (9.5.1) is the total cost (or simply the number) of large sheets used in cutting.

$$\text{minimize} \sum_{p \in PATTERNS} COST_p \cdot use_p \tag{9.5.1}$$

$$\forall s \in SIZES : \sum_{p \in PATTERNS} CUT_{sp} \cdot use_p \geq DEM_s \tag{9.5.2}$$

$$\forall p \in PATTERNS : use_p \in \mathbb{N} \tag{9.5.3}$$

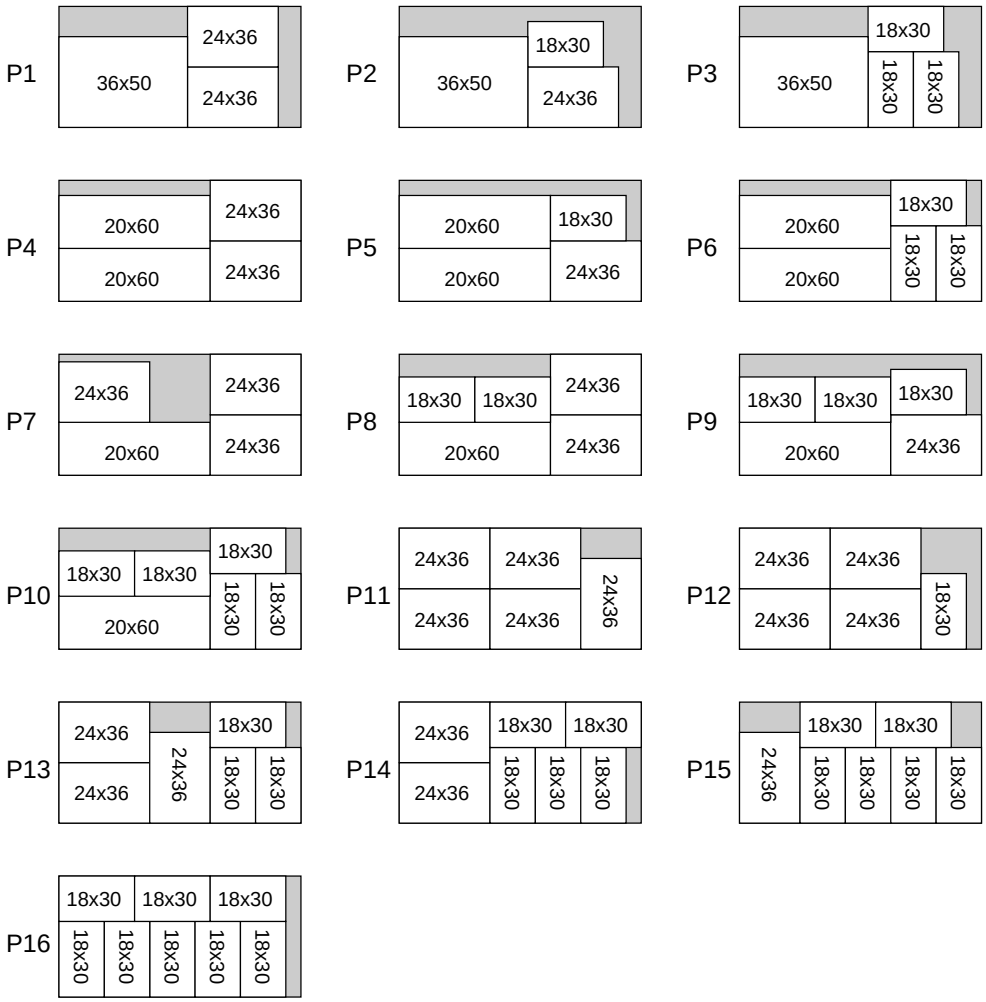

Figure 9.2: Graphical representation of the 16 cutting patterns

9.5.2 Implementation

The following Mosel program implements the mathematical model described in the previous section.

```
model "D-5 Sheet metal cutting"
 uses "mmxprs"

 declarations
  PATTERNS = 1..16                        ! Set of cutting patterns
  SIZES = 1..4                            ! Set of sheet sizes

  DEM: array(SIZES) of integer            ! Demands for the different sizes
  CUT: array(SIZES,PATTERNS) of integer   ! Cutting patterns

  use: array(PATTERNS) of mpvar           ! Use of cutting patterns
 end-declarations

 initializations from 'd5cutsh.dat'
  DEM CUT
 end-initializations

! Objective: total number of sheets used
 Sheets:= sum(p in PATTERNS) use(p)

! Satisfy demands
 forall(s in SIZES) sum(p in PATTERNS) CUT(s,p)*use(p) >= DEM(s)

 forall(p in PATTERNS) use(p) is_integer

! Solve the problem
 minimize(Sheets)

end-model
```

9.5.3 Results

The solver finds an optimal integer solution that uses 11 large sheets out of which the pattern 1 is cut 6 times, patterns 2 and 3 once, pattern 6 twice and pattern 10 once (note that there are several equivalent solutions). From this distribution come exactly the numbers of small sheets required to satisfy the order. We have thus minimized the total number of large sheets and produced exactly the required quantity, which is not necessarily the case in larger examples.

The mathematical model is very compact in its generic form after the cutting patterns have been enumerated. In practical applications it may be too difficult to carry out this task by hand and some specialized program will have to be employed to generate these patterns in order to prevent the omission of admissible combinations.

9.6 Cutting steel bars for desk legs

The company SchoolDesk produces desks of different sizes for kindergartens, primary and secondary schools, and colleges. The legs of the desks all have the same diameter, with different lengths: 40 cm for the smallest ones, 60 cm for medium height, and 70 cm for the largest ones. These legs are cut from steel bars of 1.5 or 2 meters. The company has received an order for 108 small, 125 medium and 100 large desks. How should this order be produced if the company wishes to minimize the trim loss?

9.6.1 Model formulation

This problem is similar to the previous problem of cutting sheet metal. The model formulation exploits the fact that only a small number of cutting patterns exist for the steel bars. The shorter bars of 1.5 m may, for instance, be cut into two pieces of 70 cm which leaves an offcut of 10 cm that cannot be used for any other type of leg. The same bar could alternatively be cut into one piece of 60cm and one of 70 cm, a loss of 20 cm. The following table summarizes the different possibilities.

Let *SIZES* be the set of leg/desk heights and DEM_s the demand for desks of a size s. Since every desk has four legs, we need to multiply the demand by 4 to obtain the required number of desk legs. The sets

Table 9.10: Possible cutting patterns for every bar type

	Pattern number	Leg types 40cm	60cm	70cm	Loss (in cm)
Bar type 1	1	0	0	2	10
(1.5m)	2	0	1	1	20
	3	2	0	1	0
	4	0	2	0	30
	5	2	1	0	10
	6	3	0	0	30
Bar type 2	7	0	1	2	0
(2m)	8	0	2	1	10
	9	1	0	2	20
	10	3	0	1	10
	11	0	3	0	20
	12	5	0	0	0

$PAT1$ and $PAT2$ of cutting patterns for bars of type 1 and 2 respectively are combined to form the set of all patterns $PATTERNS = PAT1 \cup PAT2$. We write LEN_b for the length of bar type b.

The objective is to minimize the trim loss, that is, the difference between the total length of the original steel bars used for cutting and the total length of the desk legs ordered. With variables use_p denoting the number of times a cutting pattern p is used, the first two terms of the objective function (9.6.1) represent the total length of the bars that are used and the last term the total length of the ordered desk legs.

$$\text{minimize} \sum_{p \in PAT1} LEN_1 \cdot use_p + \sum_{p \in PAT2} LEN_2 \cdot use_p - \sum_{s \in SIZES} 4 \cdot DEM_s \cdot s \tag{9.6.1}$$

$$\forall s \in SIZES : \sum_{p \in PATTERNS} CUT_{ps} \cdot use_p \geq 4 \cdot DEM_s \tag{9.6.2}$$

$$\forall p \in PATTERNS : use_p \in \mathbb{N} \tag{9.6.3}$$

In the objective function, the term corresponding to the length of the ordered desk legs may be omitted because it is a constant value that has no influence on the minimization process. The constraints (9.6.2), in which CUT_{ps} indicates the number of legs of length s contained in the cutting pattern p, guarantee that the demands are satisfied. The constraints (9.6.3) force the variables use_p to take non-negative integer values only.

9.6.2 Implementation

A translation of the mathematical model to a Mosel is given by the following program. Note that the set operator `+` is used to build the union of two sets (`PAT1` and `PAT2`).

```
model "D-6 Cutting steel bars"
 uses "mmxprs"

 declarations
  PAT1 = 1..6; PAT2 = 7..12              ! Sets of cutting patterns
  PATTERNS = PAT1 + PAT2                 ! Set of all cutting patterns
  SIZES: set of integer                  ! Desk heights

  DEM: array(SIZES) of integer           ! Demands for the different heights
  CUT: array(PATTERNS,SIZES) of integer  ! Cutting patterns
  LEN: array(range) of integer           ! Lengths of original steel bars

  use: array(PATTERNS) of mpvar          ! Use of cutting patterns
 end-declarations

 initializations from 'd6cutbar.dat'
  DEM CUT LEN
 end-initializations

! Objective: total loss
 Loss:= sum(p in PAT1) LEN(1)*use(p) + sum(p in PAT2) LEN(2)*use(p) -
         sum(s in SIZES) 4*DEM(s)*s

! Satisfy demands
```

```
 forall(s in SIZES) sum(p in PATTERNS) CUT(p,s)*use(p) >= 4*DEM(s)

 forall(p in PATTERNS) use(p) is_integer

! Solve the problem
 minimize(Loss)

end-model
```

In this model we not only use ranges of integers as index sets (`PAT1`, `PAT2`, `PATTERNS`, and the unnamed index set of `LEN`) as in most other examples but also a set of integers, `SIZES`. The form `set of integer` is more general than a `range` set that always contains consecutive integer numbers. In the present case, we want `SIZES` to contain only the three different leg heights (and not in addition all the integer values between the smallest and the largest height). This is a nice way of using data directly as index values.

9.6.3 Results

The minimal loss calculated by the optimization is 2020cm. The ordered desk legs are cut from 2 bars of 1.5m (using cutting patterns 1 and 3 once) and 385 bars of 2m (using 195 times pattern 7, 7 times pattern 8, 97 times pattern 11, and 86 times pattern 12). With this combination of cutting patterns, exactly the quantity of legs is produced that is needed to satisfy the order of desks.

9.7 References and further material

All problems in this chapter are NP-hard and Mathematical Programming is only able to deal with problems of moderate size. The wagon loading problem in Section 9.1 is also known as the problem of scheduling n nonpreemptive tasks on m identical machines, also called the m processor problem. Minimizing the maximum load of the wagons corresponds to minimizing the total duration of the schedule. Using a formulation as a linear program it is hardly possible to deal with more than three machines and thirty tasks. For two machines, a dynamic programming method (a kind of recursive optimization) of complexity $O(nB)$ (with B the total duration of all items) solves problems with up to one hundred objects [MT90].

Certain specialized tree search methods are able to cope with up to a hundred tasks [HW95]. When the optimal methods take too long, it is possible to find good solutions with the LPT (Longest Processing Time) heuristic that at every iteration places the unassigned object with the longest processing time onto the machine with the least load. The ratio of the LPT solution compared to the optimal solution is only $\frac{4}{3} - \frac{1}{3m}$ [Gra69]. As we have seen in Section 9.1 the LPT heuristic can easily be implemented with Mosel: the heuristic solution is close to the optimum, but not sufficiently good for the given capacity limits (which demonstrates the usefulness of optimization!). Sorting algorithms like the Shell sort used in our implementation of the LPT heuristic are described with some detail in [PFTVed].

The barge loading problems in Section 9.2 are knapsack problems; they are recognizable by their single capacity constraint. The knapsack problem with fractional variables is easy: the container must simply be filled in the order of decreasing cost per size unit. Integer knapsack problems are NP-hard, but cases of considerable size can be solved with dynamic programming or tree search methods [SDK83], [MT79].

The tank loading problem is described in a book by Christofides et al. [CMT79a]. There a tree search method is described for large instances (35 liquids, 70 tanks), and also algorithms for the dynamic case that take into account sequences of loading and unloading operations.

The problem of backing up files onto disks in Section 9.4 is a so-called bin-packing problem in which one searches to distribute n objects i of weight W_i into a minimum number of boxes among m available, every one with the same capacity. This problem is NP-hard. It is usually solved heuristically. The interested reader may be recommended the work by Coffman et al. [CGJ96] that contains a summary of over a hundred references, and the one by Martello and Toth [MT90].

Cutting stock problems like those in Sections 9.5 and 9.6 are highly combinatorial and usually contain large numbers of variables. The constraints are quite varied, like the one that cuts must go from border to border (guillotine cuts).

Among the optimal methods, there are tree-based methods, see for instance [HZ96]. Another exact method, called column generation, was introduced by Gilmore and Gomory ([GG61] and [GG63]) to solve one-dimensional problems. This method consists of firstly solving a problem that contains a very reduced subset of the columns of the complete mathematical program, the latter possibly being too large to be

generated in its entirety. The promising columns are then added progressively. When the method works well, the optimum is found after generating only a small fraction of the columns of the complete model.

For large-sized instances one needs to use metaheuristics like tabu search [LMV99] or genetic algorithms [Jak96]. Patterns that are equivalent in terms of trim loss may have different cutting costs. See Chu for methods that minimize the cost [CA99]. The formulation in 9.5 is a set covering problem like the placement of mobile phone transmitters in Chapter 12. This formulation is relatively efficient and works with up to a hundred patterns, but this limit is small compared to the enormous number of patterns that arise if the rectangles that need to be cut are small compared to the original sheets. Sweeney et al. provide a comprehensive bibliography with over 400 references for loading and cutting problems [SRP92].

Chapter 10
Ground transport

As with air transport in Chapter 11, transport by road and rail is rich in optimization problems. The main difference between the two types of networks is the high density of the network and the multiplicity of players in ground transport. The opening of frontiers and strong competition between transport providers has made the use of optimization methods a vital means of reducing transport costs and thus being able to stand out from competitors.

Section 10.1 presents a vehicle rental problem in which the cars have to be returned to the agencies at the least cost in order to establish the ideal fleet strengths. In Section 10.2 a problem of distributing among different modes of transport is described: a given quantity of goods has to be transported between two points in a network which provides different modes of transport with a known cost and limited capacities. Section 10.3 deals with a classical problem at the strategic level, the choice of depot locations that minimizes the cost of opening depots and of delivering to clients. In Section 10.4 we solve a problem optimizing the routes for the delivery of heating oil. Section 10.5 describes a combined (intermodal) transport problem that differs from the problem in 10.2 through costs for changing the mode. The problem of planning a fleet of vans terminates this chapter.

10.1 Car rental

A small car rental company has a fleet of 94 vehicles distributed among its 10 agencies. The location of every agency is given by its geographical coordinates X and Y in a grid based on kilometers. We assume that the road distance between agencies is approximately 1.3 times the Euclidean distance (as the crow flies). The following table indicates the coordinates of all agencies, the number of cars required the next morning, and the stock of cars in the evening preceding this day.

Table 10.1: Description of the vehical rental agencies

Agency	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
X coordinate	0	20	18	30	35	33	5	5	11	2
Y coordinate	0	20	10	12	0	25	27	10	0	15
Required cars	10	6	8	11	9	7	15	7	9	12
Cars present	8	13	4	8	12	2	14	11	15	7

Supposing the cost for transporting a car is € 0.50 per km, determine the movements of cars that allow the company to re-establish the required numbers of cars at all agencies, minimizing the total cost incurred for transport.

10.1.1 Model formulation

For every agency a in the given set $AGENTS$, we write X_a and Y_a for its geographic coordinates. REQ_a is the number of cars required at an agency a and $STOCK_a$ the present number of cars for each agency. The difference between these two values indicates whether there is an excess number (positive value) or a need for additional vehicles (negative value). The problem consists of finding the minimum cost flow of vehicles from the set $EXCESS$ of agencies with an excess of cars to the set $NEED$ of agencies that have a deficit of cars. A flow that re-establishes the required numbers of cars necessarily exists because the sum of excesses is equal to the sum of deficits. As a first step, we define variables $move_{ab}$ to represent the flow between two agencies:

$$\forall a \in EXCESS, b \in NEED : move_{ab} \in \mathbb{N} \tag{10.1.1}$$

Every agency with an excess number of cars needs to get rid of these (10.1.2), and every agency in need of cars has to complete its fleet by receiving the missing number (10.1.3).

$$\forall a \in EXCESS: \sum_{b \in NEED} move_{ab} = STOCK_a - REQ_a \tag{10.1.2}$$

$$\forall b \in NEED: \sum_{a \in EXCESS} move_{ab} = STOCK_b - REQ_b \tag{10.1.3}$$

The objective function to be minimized is the total cost of transporting cars (10.1.4), where *COST* denotes the transport cost per car per kilometer and $DIST_{ab}$ is the distance between two agencies *a* and *b*.

$$\text{minimize} \sum_{a \in EXCESS} \sum_{b \in NEED} COST \cdot DIST_{ab} \cdot move_{ab} \tag{10.1.4}$$

This minimum cost flow problem is a **transportation problem** recognizable by a set of sources with availabilities and a set of sinks (destinations) with demands. For minimum cost flow problems, the simplex algorithm always finds an integer solution when solving the LP. The constraints (10.1.1) may therefore be replaced by simple non-negativity constraints.

10.1.2 Implementation

The following Mosel program implements the mathematical model above. After reading in the data, we test whether the number of cars in stock and the required number of cars are the same and stop the program if this is not the case. Otherwise, the two subsets of agencies with an excess or in need of cars are calculated. These sets are used subsequently in the declaration of the array of decision variables and the distance matrix.

```
model "E-1 Car rental"
 uses "mmxprs"

 declarations
  AGENTS = 1..10                    ! Car rental agencies

  REQ: array(AGENTS) of integer     ! Required number of cars
  STOCK: array(AGENTS) of integer   ! Number of cars in stock
  X,Y: array(AGENTS) of integer     ! Coordinates of rental agencies
  COST: real                        ! Cost per km of moving a car
  NEED: set of integer              ! Agencies needing more cars
  EXCESS: set of integer            ! Agencies with too many cars
 end-declarations

 initializations from 'e1carrent.dat'
  REQ STOCK X Y COST
 end-initializations

 if sum(a in AGENTS) (STOCK(a)-REQ(a)) <> 0 then
  writeln("Problem is infeasible")
  exit(0)
 end-if

! Calculate sets of agencies with excess or deficit of cars
 forall(a in AGENTS)
  if STOCK(a) - REQ(a) < 0 then
   NEED += {a}
  elif STOCK(a) - REQ(a) > 0 then
   EXCESS += {a}
  end-if

 finalize(NEED); finalize(EXCESS)

 declarations
  DIST: array(EXCESS,NEED) of real   ! Distance between agencies
  move: array(EXCESS,NEED) of mpvar  ! Cars exchanged between agencies
 end-declarations

! Calculate distances between agencies
 forall(a in EXCESS,b in NEED)
  DIST(a,b):= 1.3*sqrt((X(a)-X(b))^2 + (Y(a)-Y(b))^2)
```

```
! Objective: total transport cost
 Cost:= sum(a in EXCESS,b in NEED) COST*DIST(a,b)*move(a,b)

! Agencies with excess availability
 forall(a in EXCESS) sum(b in NEED) move(a,b) = STOCK(a) - REQ(a)

! Agencies in need of cars
 forall(b in NEED) sum(a in EXCESS) move(a,b) = REQ(b) - STOCK(b)

 forall(a in EXCESS,b in NEED) move(a,b) is_integer

! Solve the problem
 minimize(Cost)

end-model
```

This implementation introduces the exponential operator `^`. Note that instead of using the predefined Mosel function `sqrt(r)` we could also write `r^0.5` for the square root $\sqrt{r}$.

10.1.3 Results

The optimizer calculates a total transport cost of € 152.64. The following table displays the required movements of cars between agencies to obtain the desired distribution of the fleet with this minimum cost.

Table 10.2: Optimal plan for transporting vehicles

→	1	3	4	6	7	10	Excess
2	0	1	0	5	1	0	7
5	0	0	3	0	0	0	3
8	0	0	0	0	0	4	4
9	2	3	0	0	0	1	6
Need	2	4	3	5	1	5	

10.2 Choosing the mode of transport

A company in the south-west of France needs to transport 180 tonnes of chemical products stored in depots D1 to D4 to the three recycling centers C1, C2, and C3. The depots D1 to D4 contain respectively 50, 40, 35, and 65 tonnes, that is 190 tonnes in total. Two modes of transport are available: road and rail. Depot D1 only delivers to centers C1 and C2 and that by road at a cost of € 12k/t and € 11k/t. Depot D2 only delivers to C2, by rail or road at € 12k/t and € 14k/t respectively. Depot D3 delivers to center C2 by road (€ 9k/t) and to C3 by rail or road for € 4k/t and € 5k/t respectively. The depot D4 delivers to center C2 by rail or road at a cost of € 11k/t and € 14k/t, and to C3 by rail or road at € 10k/t and € 14k/t respectively.

Its contract with the railway company for the transport of chemical products requires the company to transport at least 10 tonnes and at most 50 tonnes for any single delivery. Besides the standard security regulations, there are no specific limitations that apply to road transport. How should the company transport the 180 tonnes of chemicals to minimize the total cost of transport?

10.2.1 Model formulation

We are going to model this problem as a **minimum cost flow problem** with a fixed total throughput. We first construct a graph G = (*NODES, ARCS*). To start, we put into the set of nodes *NODES* a layer of nodes for the depots and a second one for the recycling centers (follow the construction with the help of Figure 10.1). The set of arcs *ARCS* contains the possible connections between depots and recycling centers. A transport plan corresponds to a flow in G, that is a flow $flow_{ij}$ on every arc (i, j). An arc (i, j) is characterized by a minimum flow $MINCAP_{ij}$ (0 except for rail transport), a capacity or maximum flow $MAXCAP_{ij}$ (infinity except for rail transport), and a transport cost $COST_{ij}$ per tonne.

The two modes of transport from a depot to a center require two different arcs. Such a graph, with at most p arcs in the same sense between two nodes is called a **p-graph**. Such a graph cannot be coded as a (two-dimensional) matrix: for instance the element $COST_{ij}$ of a cost matrix can only define a single cost. To obtain a graph with at most one arc between any pair of nodes, it is sufficient to create a fictitious

node per mode of transport, for every connection between a depot i and a center j. For instance, there is one connection by road and one by rail between depot D2 (node 3) and the center C1 (node 12). To avoid generating a 2-graph with two arcs (3,12), we create a node 6 for the rail transport and node 7 for road transport. The railway connection becomes the path (3,6,12), the road connection (3,7,12). The capacities and costs are only established for the arcs (3,6) and (3,7).

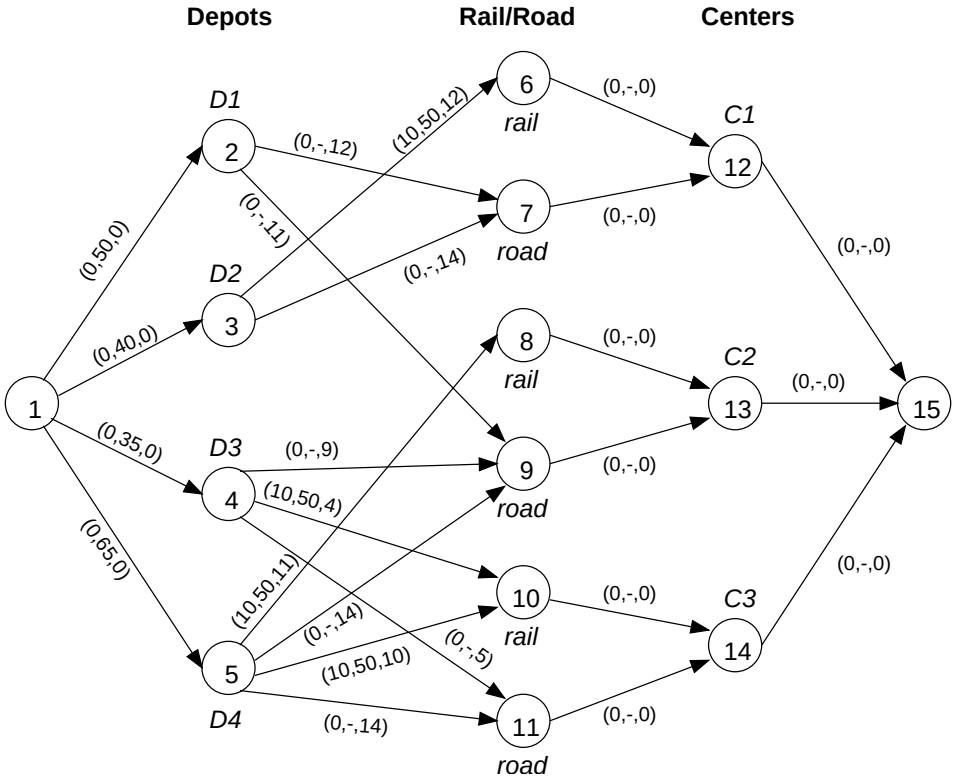

Figure 10.1: Graph of the network

The graph does not take into account the stock levels at the depots. To include these, we create a **source** node (fictitious node 1) that is connected to every depot d by an arc $(1, d)$ with a capacity $MAXCAP_{1d}$ that corresponds to the stock level at d. Therefore, the flow leaving the depot d cannot exceed this value. To facilitate the formulation of the mathematical model we also create a **sink** node (fictitious node 15) to which every center is connected. The resulting graph is represented in Figure 10.1, with the triple $(MINCAP_{ij}, MAXCAP_{ij}, COST_{ij})$ for every arc (i, j). A '–' stands for infinite capacity.

The mathematical model contains the flow conservation constraints (10.2.2), also called **Kirchhoff's law**: the sum of the incoming flows at every node (except for source and sink) equals the sum of outgoing flows. The flow on every arc has at least the minimum value $MINCAP_{ij}$ (constraints (10.2.3)), without exceeding the maximum capacity $MAXCAP_{ij}$ (constraints (10.2.4)). The constraint (10.2.5) imposes a total flow quantity $MINQ = 180$ tonnes. It stipulates that the total flow leaving the source (node 1) equals $MINQ$. It would equally be possible to establish the equivalent constraint that the total flow into the sink is equal to $MINQ$, since the flow is conserved through the network. In this constraint we could replace the equality sign by $\geq$, since the total quantity transported will be forced to its lower bound when the cost, and hence the sum of flows, is minimized.

We are left with the explanation of the objective function in line (10.2.1). Since $COST_{ij}$ is the cost per tonne, the cost of a quantity $flow_{ij}$ transported via the arc (i, j) is $COST_{ij} \cdot flow_{ij}$. And hence the total cost of transport to be minimized is the sum of the flow quantities in the entire set of arcs.

Finally, due to the way we have defined the graph, we obtain a fairly compact mathematical model. Note that the non-negativity constraints are implicitly given through the constraints (10.2.3).

$$\text{minimize} \sum_{(i,j)\in ARCS} COST_{ij} \cdot flow_{ij} \tag{10.2.1}$$

$$\forall i \in NODES,\ i \neq SOURCE, SINK : \sum_{(j,i)\in ARCS} flow_{ji} = \sum_{(i,j)\in ARCS} flow_{ij} \tag{10.2.2}$$

$$\forall (i, j) \in ARCS : flow_{ij} \geq MINCAP_{ij} \tag{10.2.3}$$

$$\forall (i,j) \in ARCS : flow_{ij} \leq MAXCAP_{ij} \tag{10.2.4}$$

$$\sum_{(SOURCE,i)\in ARCS} flow_{SOURCE,i} = MINQ \tag{10.2.5}$$

An alternative to this graph-based formulation of the problem would be to represent the quantity of product transported from a depot d to a client c using mode m as a decision variable $transport_{dcm}$ for every admissible triple (d, c, m). The total transported quantity is then given by the sum of all defined variables $transport_{dcm}$, and the objective function is the sum of $COST_{dcm} \cdot transport_{dcm}$ for all admissible triples (d, c, m). The minimum and maximum capacities of certain modes of transport are given as bound constraints on the corresponding variables $transport_{dcm}$, similarly to constraints (10.2.3) and (10.2.4) above. For our specific problem, this formulation may be easier than the graph-based formulation. However, for the implementation and further discussion of this problem we use the generic minimum cost flow model given by lines (10.2.1) and (10.2.5).

10.2.2 Implementation

This problem raises a classical problem, namely the coding of a graph. It is possible to represent a graph through an $N \times N$ matrix (where N is the total number of nodes), and define the flow variables for the pairs of nodes (i, j) that are connected by an arc. However, for sparse graphs like the present one, it is often preferable (and more efficient) to represent the graph as a list of arcs. This representation is used in the following Mosel implementation, as opposed to the representation in the mathematical model above. An arc $a = (i, j)$ is referred to by its counter a and not via the node pair (i, j). The list of arcs is given in the form of a two-dimensional array A with $A_{a1} = i$ and $A_{a2} = j$. The flow variables are defined once the data has been read (and hence, the set of arcs is known). Note that the nodes in this implementation are not numbered but labeled 'SOURCE', 'D2', 'C4' *etc.* to ease the interpretation of the results.

```
model "E-2 Minimum cost flow"
 uses "mmxprs"

 declarations
  NODES: set of string                  ! Set of nodes
  MINQ : integer                        ! Total quantity to transport
  A: array(ARCS:range,1..2) of string   ! Arcs
  COST: array(ARCS) of integer          ! Transport cost on arcs
  MINCAP,MAXCAP: array(ARCS) of integer ! Minimum and maximum arc capacities
 end-declarations

 initializations from 'e2minflow.dat'
  A MINQ MINCAP MAXCAP COST
 end-initializations

 finalize(ARCS)

! Calculate the set of nodes
 NODES:= union(a in ARCS) {A(a,1),A(a,2)}

 declarations
  flow: array(ARCS) of mpvar            ! Flow on arcs
 end-declarations

! Objective: total transport cost
 Cost:= sum(a in ARCS) COST(a)*flow(a)

! Flow balance: inflow equals outflow
 forall(n in NODES | n<>"SOURCE" and n<>"SINK")
  sum(a in ARCS | A(a,2)=n) flow(a) = sum(a in ARCS | A(a,1)=n) flow(a)

! Min and max flow capacities
 forall(a in ARCS | MAXCAP(a) > 0) do
  flow(a) >= MINCAP(a)
  flow(a) <= MAXCAP(a)
 end-do

! Minimum quantity to transport
 sum(a in ARCS | A(a,1)="SOURCE" ) flow(a) >= MINQ

! Solve the problem
 minimize(Cost)
```

```
end-model
```

This model introduces another aggregate operator of Mosel: the set of nodes `NODES` is constructed as the `union` of all nodes connected by arcs in the set `ARCS`.

10.2.3 Results

The minimum cost is € 1,715k. Figure 10.2 displays the solution: arcs that are used for transporting goods are labeled with the transported quantities, unused arcs and nodes are printed with dotted lines. For example, the entire stock of 50 tonnes at depot D1 is transported by road to the recycling center C2.

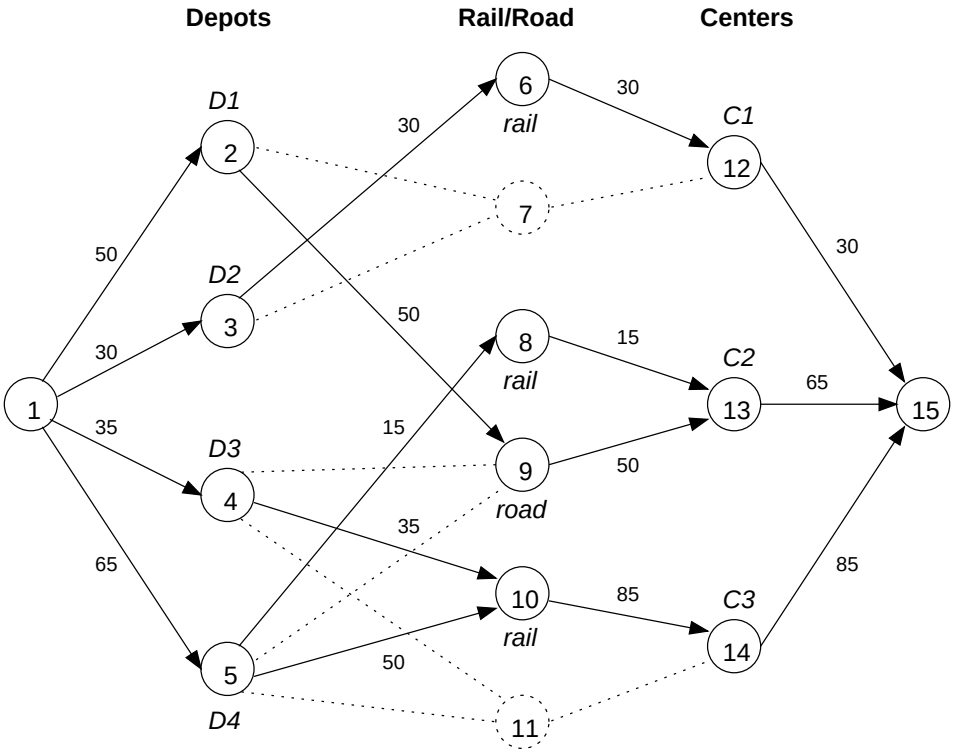

Figure 10.2: Optimal transport plan

It may be noted that such a problem with minimum flows on its arcs may well have no solution. This is the case, for instance, if *MINQ* = 10 because the four arcs of rail transport leaving the depots each require at least a minimum flow of 10 tonnes.

10.2.4 Extension

This model may be used to solve any type of minimum cost flow problem. For example, it would be possible to add demands for the centers. For a center *c*, it is sufficient to place the demand as a minimum value for the flow on the arc (*c, sink*) to guarantee the satisfaction of the demand. However, a necessary condition for the existence of a solution needs to be fulfilled: the sum of product availabilities at the depots must be greater than or equal to the sum of the demands at the centers.

10.3 Depot location

A large company wishes to open new depots to deliver to its sales centers. Every new set-up of a depot has a fixed cost. Goods are delivered from a depot to the sales centers close to the site. Every delivery has a cost that depends on the distance covered. The two sorts of cost are quite different: set-up costs are capital costs which may usually be written off over several years, and transport costs are operating costs. A detailed discussion of how to combine these two costs is beyond the scope of this book — we assume here that they have been put on some comparable basis, perhaps by taking the costs over a year.

There are 12 sites available for the construction of new depots and 12 sales centers need to receive deliveries from these depots.

The following Table 10.3 gives the costs (in thousand €) of satisfying the entire demand of each customer (sales center) from a depot (not the unit costs). So, for instance, the cost per unit of supplying customer 9 (who has a total demand of 30 tonnes according to Table 10.5) from depot 1 is €60000/30t, *i.e.* €2000/t. Certain deliveries that are impossible are marked with the infinity symbol ∞.

Table 10.3: Delivery costs for satisfying entire demand of customers

	Customer											
Depot	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
1	100	80	50	50	60	100	120	90	60	70	65	110
2	120	90	60	70	65	110	140	110	80	80	75	130
3	140	110	80	80	75	130	160	125	100	100	80	150
4	160	125	100	100	80	150	190	150	130	∞	∞	∞
5	190	150	130	∞	∞	∞	200	180	150	∞	∞	∞
6	200	180	150	∞	∞	∞	100	80	50	50	60	100
7	100	80	50	50	60	100	120	90	60	70	65	110
8	120	90	60	70	65	110	140	110	80	80	75	130
9	140	110	80	80	75	130	160	125	100	100	80	150
10	160	125	100	100	80	150	190	150	130	∞	∞	∞
11	190	150	130	∞	∞	∞	200	180	150	∞	∞	∞
12	200	180	150	∞	∞	∞	100	80	50	50	60	100

In addition, for every depot we have the following information: the fixed cost for constructing the depot that needs to be included into the objective function and its capacity limit, all listed in Table 10.4.

Table 10.4: Fix costs and capacity limits of the depot locations

Depot	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Cost (k€)	3500	9000	10000	4000	3000	9000	9000	3000	4000	10000	9000	3500
Capacity (t)	300	250	100	180	275	300	200	220	270	250	230	180

The quantities demanded by the sales centers (customers), are summarized in the following table.

Table 10.5: Demand data

Customer	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Demand (t)	120	80	75	100	110	100	90	60	30	150	95	120

In every case, the demand of a customer needs to be satisfied but a sales center may be delivered to from several depots. Which depots should be opened to minimize the total cost of construction and of delivery, whilst satisfying all demands?

10.3.1 Model formulation

To formulate the mathematical model, we write DEM_c for the demand by customer (sales center) c, and CAP_d for the maximum capacity of depot d. The fixed cost of constructing depot d is given by $CFIX_d$, the cost of delivery from depot d to customer c as $COST_{cd}$. Furthermore, let *DEPOTS* be the set of all possible depot locations and *CUST* the set of customers to be delivered to from these depots.

To solve the problem, we need to know which depots will be opened. So we define binary variables $build_d$ that take the value 1 if site d is chosen, and otherwise 0. In addition, we also need to know which depot(s) deliver(s) goods to a customer. We introduce variables $fflow_{dc}$ for the **fraction** of the demand of customer c that is satisfied from depot d. These variables take their values in the interval [0, 1]; we therefore have the constraints (10.3.1).

$$\forall d \in DEPOTS, c \in CUST : fflow_{dc} \leq 1 \tag{10.3.1}$$

The demand of every customer needs to be entirely satisfied:

$$\forall c \in CUST : \sum_{d \in DEPOTS} fflow_{dc} = 1 \tag{10.3.2}$$

We now have to model the fact that the total amount leaving depot d must be no more than its capacity CAP_d, but that the flow is zero if the depot is not built. The constraints (10.3.3) capture this.

$$\forall d \in DEPOTS : \sum_{c \in CUST} DEM_c \cdot fflow_{dc} \leq CAP_d \cdot build_d \tag{10.3.3}$$

Let us now see why. Since $fflow_{dc}$ is the fraction of customer c's satisfied demand, $fflow_{dc} \cdot DEM_c$ is the amount going from d to c. The total outflow from d must be no more than CAP_d if d is built ($build_d = 1$), and must be 0 if $build_d = 0$.

The total cost to be minimized consists of the costs for constructing the depots and the delivery costs. These are the two sums comprising the objective function (10.3.4). The complete mathematical model is given by the following.

$$\text{minimize} \sum_{d \in DEPOTS} CFIX_d \cdot build_d + \sum_{d \in DEPOTS} \sum_{c \in CUST} COST_{dc} \cdot fflow_{dc} \tag{10.3.4}$$

$$\forall d \in DEPOTS, c \in CUST : fflow_{dc} \leq 1 \tag{10.3.5}$$

$$\forall c \in CUST : \sum_{d \in DEPOTS} fflow_{dc} = 1 \tag{10.3.6}$$

$$\forall d \in DEPOTS : \sum_{c \in CUST} DEM_c \cdot fflow_{dc} \leq CAP_d \cdot build_d \tag{10.3.7}$$

$$\forall d \in DEPOTS, c \in CUST : fflow_{dc} \geq 0 \tag{10.3.8}$$

$$\forall d \in DEPOTS : build_d \in \mathbb{N} \tag{10.3.9}$$

10.3.2 Implementation

The following Mosel program is a straightforward translation of the mathematical model.

```
model "E-3 Depot location"
 uses "mmxprs"

 declarations
  DEPOTS = 1..12                       ! Set of depots
  CUST = 1..12                         ! Set of customers

  COST: array(DEPOTS,CUST) of integer  ! Delivery cost
  CFIX: array(DEPOTS) of integer       ! Fix cost of depot construction
  CAP: array(DEPOTS) of integer        ! Depot capacity
  DEM: array(CUST) of integer          ! Demand by customers

  fflow: array(DEPOTS,CUST) of mpvar   ! Perc. of demand supplied from depot
  build: array(DEPOTS) of mpvar        ! 1 if depot built, 0 otherwise
 end-declarations

 initializations from 'e3depot.dat'
  COST CFIX CAP DEM
 end-initializations

! Objective: total cost
 TotCost:= sum(d in DEPOTS, c in CUST) COST(d,c)*fflow(d,c) +
           sum(d in DEPOTS) CFIX(d)*build(d)

! Satisfy demands
 forall(c in CUST) sum(d in DEPOTS) fflow(d,c) = 1

! Capacity limits at depots
 forall(d in DEPOTS) sum(c in CUST) DEM(c)*fflow(d,c) <= CAP(d)*build(d)

 forall(d in DEPOTS) build(d) is_binary
 forall(d in DEPOTS, c in CUST) fflow(d,c) <= 1

! Solve the problem
 minimize(TotCost)

end-model
```

We could add an additional set of constraints stating the relation 'if there is any delivery from depot d, then this depot must be built' (and its inverse 'if a depot is not built, then there is no delivery from this

depot'). This relation is implied by the constraints (10.3.3) but the additional (disaggregated) constraints provide a tighter formulation. That is, if these constraints are added to the model they draw the solution value of the LP relaxation closer to the MIP solution. The additional constraints may be added directly to the model, but since the model is fully stated through the version printed above, we could turn them into **model cuts** as shown below. By defining these constraints as model cuts we leave the choice to the optimizer whether to use these additional constraints or not. Note that any constraint that is to become a model cut needs to be named and declared globally in the Mosel program as shown in the following program extract (linear constraints in Mosel have the type `linctr`).

```
declarations
 modcut: array(DEPOTS,CUST) of linctr
end-declarations

forall(d in DEPOTS, c in CUST) do
 modcut(d,c):= fflow(d,c) <= build(d)
 setmodcut(modcut(d,c))
end-do
```

10.3.3 Results

The optimization algorithm calculates a minimum total cost of € 18,103k. The five depots 1, 5, 8, 9, and 12 are built. The following table details how the customers are delivered to from these depots. All depots built, except depot 5 are used to their maximum capacity.

Table 10.6: Delivery plan

	Customer											
Depot	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
1	–	5	75	100	–	–	–	–	–	–	–	120
5	120	40	–	–	–	–	–	–	–	–	–	–
8	–	35	–	–	–	100	–	–	–	85	–	–
9	–	–	–	–	110	–	–	–	–	65	95	–
12	–	–	–	–	–	–	90	60	30	–	–	–

10.4 Heating oil delivery

A transporter has to deliver heating oil from the refinery at Donges to a certain number of clients in the west of France. His clients are located at Brain-sur-l'Authion, Craquefou, Guérande, la Haie Fouassière, Mésanger and les Ponts-de-Cé. The following table lists the demands in liters for the different sites.

Table 10.7: Demands by clients (in liters)

Brain-sur-l'Authion	**Craquefou**	**Guérande**	**Haie Fouassière**	**Mésanger**	**Ponts-de-Cé**
14000	3000	6000	16000	15000	5000

The next table contains the distance matrix between the clients and the refinery.

Table 10.8: Distance matrix (in km)

	Donges	**Brain-sur-l'Authion**	**Craquefou**	**Guérande**	**Haie Fouassière**	**Mésanger**	**Ponts-de-Cé**
Donges	0	148	55	32	70	140	73
Brain-s.-l'Authion	148	0	93	180	99	12	72
Craquefou	55	93	0	85	20	83	28
Guérande	32	180	85	0	100	174	99
Haie Fouassière	70	99	20	100	0	85	49
Mésanger	140	12	83	174	85	0	73
Ponts-de-Cé	73	72	28	99	49	73	0

The transport company uses tankers with a capacity of 39000 liters for the deliveries. Determine the tours for delivering to all clients that minimize the total number of kilometers driven.

10.4.1 Model formulation

This problem may be seen as a generalization of the famous Traveling Salesman Problem (TSP), an example of which we see in Section 11.5. In this case there are several 'salesmen', who each have to have a tour starting and finishing at Donges. We have to determine not only the cities in the tours, but also how many tours there are to be, and the order the cities in each tour are visited.

We introduce variables $prec_{ij}$ that take the value 1 if town i immediately precedes town j in a tour, and 0 otherwise. Let $SITES = \{1, \ldots, NS\}$ be the number of sites. Site 1 is the refinery, so that we have the subset $CLIENTS = \{2, \ldots, NS\}$ of sites to which we deliver. Let $DIST_{ij}$ be the distance between two towns i and j, DEM_i the quantity ordered by client i, and CAP the maximum capacity of the tankers. We also use variables $quant_i$ for the total amount of oil delivered on the route that includes client i up to and including client i. For example, if the route including 10 is 1,3,11,10,6,1, then $quant_{10}$ would be $DEM_3 + DEM_{11} + DEM_{10}$. With these notations, we may formulate the following mathematical model:

$$\text{minimize} \sum_{i \in SITES} \sum_{j \in SITES, i \neq j} DIST_{ij} \cdot prec_{ij} \tag{10.4.1}$$

$$\forall j \in CLIENTS : \sum_{i \in SITES, i \neq j} prec_{ij} = 1 \tag{10.4.2}$$

$$\forall i \in CLIENTS : \sum_{j \in SITES, j \neq i} prec_{ij} = 1 \tag{10.4.3}$$

$$\forall i \in CLIENTS : DEM_i \leq quant_i \leq CAP \tag{10.4.4}$$

$$\forall i \in CLIENTS : quant_i \leq CAP + (DEM_i - CAP) \cdot prec_{1i} \tag{10.4.5}$$

$$\forall i, j \in CLIENTS, i \neq j :$$
$$quant_j \geq quant_i + DEM_j - CAP + CAP \cdot prec_{ij} + (CAP - DEM_j - DEM_i) \cdot prec_{ji} \tag{10.4.6}$$

$$\forall i \in CLIENTS : quant_i \geq 0 \tag{10.4.7}$$

$$\forall i, j \in SITES, i \neq j : prec_{ij} \in \{0, 1\} \tag{10.4.8}$$

The objective (10.4.1) of this problem is to minimize the total number of kilometers driven. Every customer site has to be delivered to once. This is expressed through the two sets of constraints (10.4.2) and (10.4.3) that make the delivery enter and leave every town (except the depot) exactly once.

The quantity $quant_i$ must be at least as large as the quantity ordered by client i and within the capacity limit CAP of the tankers (10.4.4).

Furthermore, if client i is the first of a tour, then $quant_i$ is equal to the quantity ordered by this client. This constraint is expressed through the two sets of constraints (10.4.4) and (10.4.5). Indeed, if i is the first client of a tour, then $prec_{1i}$ is 1 and, after simplification, the constraint (10.4.5) is equivalent to the constraint (10.4.9).

$$quant_i \leq DEM_i \tag{10.4.9}$$

From (10.4.9) and (10.4.4) it follows that $quant_i$ is equal to the demand of client i. If i is not the first of a tour, $prec_{1i}$ is 0 and the constraint (10.4.5) is equivalent to the constraint (10.4.10), which is redundant since it is already expressed in constraint (10.4.4).

$$quant_i \leq CAP \tag{10.4.10}$$

Let us now consider the case where i is not the first customer of the tour. Then $quant_i$ must equal the sum of quantities delivered between the refinery and i inclusively. This means that if client j comes after client i in a tour, we can write that $quant_j$ must be equal to the quantity delivered on the tour from the refinery to i, plus the quantity ordered by j. This relation is stated by the constraint (10.4.6). If indeed j is the immediate successor of i in a tour, then $prec_{ij}$ is 1 and $prec_{ji}$ is 0, and the constraint (10.4.6) is equivalent to the constraint (10.4.11).

$$quant_j \geq quant_i + DEM_j \tag{10.4.11}$$

When j does not come immediately after i, constraint (10.4.6) still remains valid. If j is the immediate predecessor of i, the constraint (10.4.6) becomes (10.4.12).

$$quant_j \geq quant_i - DEM_i \tag{10.4.12}$$

This constraint means that the quantity delivered from the refinery up to j is no less than the quantity delivered between the depot and the successor i of j on the tour, a quantity that needs to be reduced by

the delivery at i. If j is the immediate predecessor of i, then i is the immediate successor of j. We therefore obtain in addition to (10.4.12) the constraint (10.4.13) by swapping the indices in (10.4.11).

$$quant_i \geq quant_j + DEM_i \tag{10.4.13}$$

The combination of constraints (10.4.12) and (10.4.13) is equivalent to the equation (10.4.14).

$$quant_i = quant_j + DEM_i \tag{10.4.14}$$

If i and j are not next to each other on a tour, we obtain the constraint (10.4.15). Since the terms on the right hand side of the inequality sign are less than or equal to DEM_j, this constraint is redundant since it is subsumed by the constraint (10.4.4).

$$quant_j \geq quant_i + DEM_j - CAP \tag{10.4.15}$$

And finally, constraints (10.4.7) and (10.4.8) indicate that the variables $quant_i$ are non-negative and that the $prec_{ij}$ are binary variables.

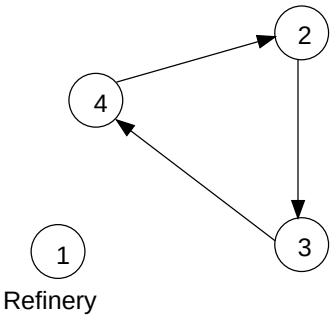

Figure 10.3: Example of an infeasible solution

To close we would like to remark that the assignment of variables $quant_i$ to every node i guarantees that the capacity limits of the tankers are not exceeded, whilst making any tour impossible that does not include the depot. Without these variables, it would be possible to obtain solutions like the one represented in Figure 10.3.

This solution satisfies the constraints (10.4.2) and (10.4.3) because every node is entered and left exactly once, but it is infeasible because the tour does not pass through the refinery. Assigning the strictly increasing values $quant_i$ all along a tour excludes this type of solution.

10.4.2 Implementation

The following Mosel program implements the mathematical model of lines (10.4.1) – (10.4.8).

```
model "E-4 Oil delivery"
 uses "mmxprs"

 declarations
  NS = 7
  SITES = 1..NS                      ! Set of locations, 1=refinery
  CLIENTS = 2..NS

  DEM: array(SITES) of integer       ! Demands
  DIST: array(SITES,SITES) of integer ! Distances between locations
  CAP: integer                       ! Lorry capacity

  prec: array(SITES,SITES) of mpvar  ! 1 if i immediately precedes j,
                                     ! 0 otherwise
  quant: array(CLIENTS) of mpvar     ! Quantity delivered up to i
 end-declarations

 initializations from 'e4deliver.dat'
  DEM DIST CAP
 end-initializations

! Objective: total distance driven
```

```
 Length:= sum(i,j in SITES | i<>j) DIST(i,j)*prec(i,j)

! Enter and leave every city only once (except the depot)
 forall(j in CLIENTS) sum(i in SITES| i<>j) prec(i,j) = 1
 forall(i in CLIENTS) sum(j in SITES| i<>j) prec(i,j) = 1

! If i is the first client of a tour, then quant(i)=DEM(i)
 forall(i in CLIENTS) quant(i) <= CAP + (DEM(i)-CAP)*prec(1,i)

! If j comes just after i in a tour, then quant(j) is greater than the
! quantity delivered during the tour up to i plus the quantity to be
! delivered at j (to avoid loops and keep capacity limit of the tanker)
 forall(i,j in CLIENTS| i<>j) quant(j) >= quant(i) + DEM(j) - CAP +
                           CAP*prec(i,j) + (CAP-DEM(j)-DEM(i))*prec(j,i)

 forall(i in CLIENTS) do
  quant(i) <= CAP
  quant(i) >= DEM(i)
 end-do

 forall(i,j in SITES | i<>j) prec(i,j) is_binary

! Solve the problem
 minimize(Length)

end-model
```

It is possible to define an additional set of constraints stating that $quant_i$ is greater than or equal to the sum of the quantities to deliver to client i and to his predecessor on the tour. These constraints may help improve the lower bound on the variables $quant_i$ for sites i that are not visited first in a tour. Since the model is fully stated through the version printed above we turn them into **model cuts**, leaving the choice to the optimizer whether to use these additional constraints or not. The reader is reminded that any constraint that is to become a model cut needs to be named and declared globally in the Mosel program.

```
 declarations
  modcut: array(CLIENTS) of linctr
 end-declarations

 forall(i in CLIENTS) do
  modcut(i):= quant(i) >= DEM(i) + sum(j in SITES| i<>j) DEM(j)*prec(j,i)
  setmodcut(modcut(i))
 end-do
```

10.4.3 Results

The optimal solution involves making two delivery tours. One tour delivers a total of 22000 liters of heating oil, visiting first Guérande and then Haie Fouassière. The other tour goes from the refinery to Mésanger, then Brain-sur-l'Authion, Les Ponts-de-Cé, and Craquefou, delivering a total of 37000 liters. A total distance of 497 km needs to be driven for these tours.

10.5 Combining different modes of transport

A load of 20 tonnes needs to be transported on a route passing through five cities, with a choice of three different modes of transport: rail, road, and air. In any of the three intermediate cities it is possible to change the mode of transport but the load uses a single mode of transport between two consecutive cities. Table 10.9 lists the cost of transport in $ per tonne between the pairs of cities.

Table 10.9: Transport costs with different modes

	Pairs of cities			
	1–2	**2–3**	**3–4**	**4–5**
Rail	30	25	40	60
Road	25	40	45	50
Air	40	20	50	45

The next table (10.10) summarizes the costs for changing the mode of transport in $ per tonne. The cost is independent of location.

Table 10.10: Cost for changing the mode of transport

from \ to	Rail	Road	Air
Rail	0	5	12
Road	8	0	10
Air	15	10	0

How should we organize the transport of the load at the least cost?

10.5.1 Model formulation

Let *MODES* be the set of modes of transport. The connection between a pair of consecutive cities is referred to as a **leg** of the route, the complete transport trajectory being given by the set *LEGS* of all legs. Let $CTRANS_{ml}$ be the transport cost using mode m on leg l, and $CCHG_{mn}$ the cost for changing from mode m to mode n which is independent of the site where it takes place in our example. We need two types of binary variables to handle the two types of costs: a first group (10.5.1) of variables use_{ml} that have the value 1 if mode m is used for leg l of the total trajectory, and a second group (10.5.2) of variables $change_{mnl}$ with value 1 if there is a change from mode m to mode n between legs l and $l+1$.

$$\forall m \in MODES, l \in LEGS : use_{ml} \in \{0, 1\} \tag{10.5.1}$$

$$\forall m, n \in MODES, l \in \{1, \ldots, NL-1\} : change_{mnl} \in \{0, 1\} \tag{10.5.2}$$

A single mode of transport has to be used on each leg (10.5.3).

$$\forall l \in LEGS : \sum_{m \in MODES} use_{ml} = 1 \tag{10.5.3}$$

A single change of the mode of transport may take place at every intermediate city (10.5.4). In this case, the change from a mode to the same mode does not cost anything since the load stays on board the same vehicle. It would of course be possible to have a non-zero cost if the load changed to a different vehicle of the same type.

$$\forall l \in \{1, \ldots, NL-1\} : \sum_{m,n \in MODES} change_{mnl} = 1 \tag{10.5.4}$$

In the city between the legs l and $l+1$, we have a change of the mode from m to n (proposition A) if and only if the mode m is used for leg l and mode n is used on leg $l+1$ (proposition B). The constraints (10.5.5) provide a linear formulation of the implication $A \Rightarrow B$: if $change_{mnl} = 1$, then $use_{ml} = 1$ and $use_{n,l+1} = 1$. Theoretically, taking these constraints isolated from the rest, it would be possible to have $use_{ml} = use_{n,l+1} = 1$ with $change_{mnl} = 0$, but this case is excluded through the constraints (10.5.4).

$$\forall m, n \in MODES, l \in \{1, \ldots, NL-1\} : use_{ml} + use_{n,l+1} \geq 2 \cdot change_{mnl} \tag{10.5.5}$$

An alternative formulation of these constraints are the two sets of constraints (10.5.6) and (10.5.7).

$$\forall m, n \in MODES, l \in \{1, \ldots, NL-1\} : use_{ml} \geq change_{mnl} \tag{10.5.6}$$

$$\forall m, n \in MODES, l \in \{1, \ldots, NL-1\} : use_{n,l+1} \geq change_{mnl} \tag{10.5.7}$$

These alternative constraints are **stronger**: they exclude more fractional solutions than the constraints (10.5.5), at the cost of defining twice as many constraints.

The objective function (10.5.8), in $ per tonne, comprises the transport cost for every leg of the trajectory (depending on the mode of transport used) and the sum of the costs for changing the mode in the intermediate cities.

$$\text{minimize} \sum_{m \in MODES} \sum_{l \in LEGS} CTRANS_{ml} \cdot use_{ml} + \sum_{m \in MODES} \sum_{n \in MODES} \sum_{l=1}^{NL-1} CCHG_{mn} \cdot change_{mnl} \tag{10.5.8}$$

10.5.2 Implementation

The mathematical model translates into the following Mosel program. The implementation uses the 'weak' constraints (10.5.5).

```
model "E-5 Combined transport"
 uses "mmxprs"

 declarations
  NL = 4
  LEGS = 1..NL                          ! Legs of the transport
  MODES: set of string                  ! Modes of transport

  CTRANS: array(MODES,LEGS) of integer  ! Transport cost
  CCHG: array(MODES,MODES) of integer   ! Cost of changing mode of transport
 end-declarations

 initializations from 'e5combine.dat'
  CTRANS CCHG
 end-initializations

 finalize(MODES)

 declarations
  use: array(MODES,LEGS) of mpvar        ! 1 if a mode is used, 0 otherwise
  change: array(MODES,MODES,1..NL-1) of mpvar  ! 1 if change from mode m to n
                                               ! at end of leg, 0 otherwise
 end-declarations

! Objective: total cost
 Cost:= sum(m in MODES, l in LEGS) CTRANS(m,l)*use(m,l) +
         sum(m,n in MODES,l in 1..NL-1) CCHG(m,n)*change(m,n,l)

! One mode of transport per leg
 forall(l in LEGS) sum(m in MODES) use(m,l) = 1

! Change or maintain mode of transport between every pair of legs
 forall(l in 1..NL-1) sum(m,n in MODES) change(m,n,l) = 1

! Relation between modes used and changes
 forall(m,n in MODES,l in 1..NL-1) use(m,l) + use(n,l+1) >= 2*change(m,n,l)

 forall(m in MODES, l in LEGS) use(m,l) is_binary
 forall(m,n in MODES,l in 1..NL-1) change(m,n,l) is_binary

! Solve the problem
 minimize(Cost)

end-model
```

10.5.3 Results

The minimum total cost calculated by the optimizer is \$104/t. This value is obtained from \$100/t of transport costs and \$4/t for changing the mode of transport. The following table displays the results in detail.

Table 10.11: Modes of transport used and respective costs

	1–2	Change at 2	2–3	Change at 3	3–4	Change at 4	4–5
Mode of transport	rail		rail		rail		road
Cost (in \$/t)	30	0	25	0	40	5	50

10.6 Fleet planning for vans

A chain of department stores uses a fleet of vans rented from different rental agencies. For the next six months period it has forecast the following needs for vans (Table 10.12):

Table 10.12: Requirements for vans for six months

January	February	March	April	May	June
430	410	440	390	425	450

At the 1st January, the chain has 200 vans, for which the rental period terminates at the end of February.

To satisfy its needs, the chain has a choice among three types of contracts that may start the first day of every month: 3-months contracts for a total cost of $1700 per van, 4-months contracts at $2200 per van, and 5-months contracts at $2600 per van. How many contracts of the different types need to be started every month in order to satisfy the company's needs at the least cost and to have no remaining vans rented after the end of June?

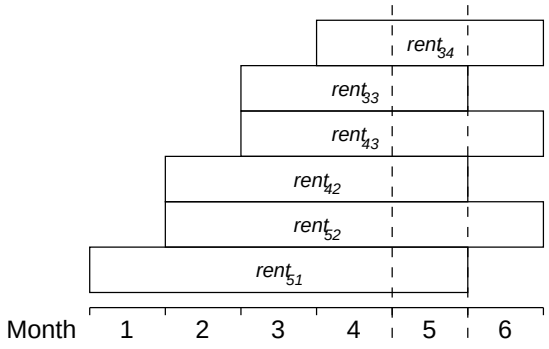

Figure 10.4: Running contracts in month 5 (May)

10.6.1 Model formulation

Let *NINIT* be the number of vans already rented at the beginning of the planning period. We define integer variables $rent_{cm}$ to denote the number of contracts of type c ($c \in CONTR = \{3, 4, 5\}$) to start at the beginning of month m ($m \in MONTHS = \{1, \ldots, 6\}$). To clarify the constraint formulation we are using, let us first look at a specific formulation. Take for example the constraint for month 5 (May) represented in Figure 10.4: the running contracts may be 5-months contracts signed in January, 4 or 5-months contracts signed in February, 3 or 4-months contracts signed in March, or 3-months contracts signed in April. These contracts must satisfy the requirement of 425 vans for the month of May. Note that for instance a 5-month contract cannot be signed in March because it does not terminate at the end of June, nor can any contract be signed in May.

$$
\begin{aligned}
m = 1: &\quad NINT + rent_{31} + rent_{41} + rent_{51} \geq 430 \\
m = 2: &\quad NINT + rent_{31} + rent_{41} + rent_{51} + rent_{32} + rent_{42} + rent_{52} \geq 410 \\
m = 3: &\quad rent_{31} + rent_{41} + rent_{51} + rent_{32} + rent_{42} + rent_{52} + rent_{33} + rent_{43} \geq 440 \\
m = 4: &\quad rent_{41} + rent_{51} + rent_{32} + rent_{42} + rent_{52} + rent_{33} + rent_{43} + rent_{34} \geq 390 \\
m = 5: &\quad rent_{51} + rent_{42} + rent_{52} + rent_{33} + rent_{43} + rent_{34} \geq 425 \\
m = 6: &\quad rent_{52} + rent_{43} + rent_{34} \geq 450
\end{aligned}
$$

It is possible to write these constraints in a generic way. Let *NM* denote the number of months, $COST_c$ the cost per van of contracts of duration c, and REQ_m the requirement of vans in month m.

$$\text{minimize} \sum_{c \in CONTR} \sum_{m \in MONTHS} COST_c \cdot rent_{cm} \tag{10.6.1}$$

$$\forall m = 1, 2 : NINIT + \sum_{c \in CONTR} \sum_{n=max(1,m-c+1)}^{min(m,NM-c+1)} rent_{cn} \geq REQ_m \tag{10.6.2}$$

$$\forall m = 3, \ldots, NM : \sum_{c \in CONTR} \sum_{n=max(1,m-c+1)}^{min(m,NM-c+1)} rent_{cn} \geq REQ_m \tag{10.6.3}$$

$$\forall c \in CONTR, m \in MONTHS : rent_{cm} \in \mathbb{N} \tag{10.6.4}$$

The objective function is to minimize the total cost for all contracts that are signed (10.6.1). A contract lasts c months, therefore the requirement of vans in any month m is covered by the contracts of type c signed in the months $max(1, m-c+1)$ and $min(m, NM-c+1)$. The constraints (10.6.3) specify that starting from March the signed contracts have to cover the need for vans of the month. For January and February, the constraints (10.6.2) also include the number of vans initially available *NINIT*.

The constraints (10.6.4) establish the integrality constraints for the variables. From LP theory it is possible to show that this mathematical program is equivalent to a minimum cost flow problem that is known to have integer values at the (linear) optimum. It is therefore possible to replace the constraints (10.6.4) by simple non-negativity conditions.

10.6.2 Implementation

The constraints (10.6.2) and (10.6.3) of the mathematical model are combined into a single statement in the following Mosel program.

```
model "E-6 van rental"
 uses "mmxprs"

 declarations
  NM = 6
  MONTHS = 1..NM                      ! Months
  CONTR = 3..5                        ! Contract types

  REQ: array(MONTHS) of integer       ! Monthly requirements
  COST: array(CONTR) of integer       ! Cost of contract types
  NINIT: integer                      ! Vans rented at beginning of plan

  rent: array(CONTR,MONTHS) of mpvar  ! New rentals every month
 end-declarations

 initializations from 'e6vanrent.dat'
  REQ COST NINIT
 end-initializations

! Objective: total cost
 Cost:= sum(c in CONTR, m in MONTHS) COST(c)*rent(c,m)

! Fulfill the monthly requirements
 forall(m in MONTHS)
  if(m<=2, NINIT, 0) +
   sum(c in CONTR, n in maxlist(1,m-c+1)..minlist(m,NM-c+1)) rent(c,n) >=
    REQ(m)

! Solve the problem
 minimize(Cost)
end-model
```

In this implementation, we use the Mosel functions `maxlist` and `minlist` to calculate respectively the maximum and minimum value of a list of numbers. In this example, each time only two numbers are given as arguments to these functions, but in the general case the lists may have any finite number of entries, such as `maxlist(2,-10,A(3),B(7,-5))` (where `A` and `B` are assumed to be arrays of integers or reals). These two functions should not be confused with the aggregate operators `max` and `min` that provide a similar functionality, but are used with set expressions, such as `max(i in ISET) A(i)` (where `A` is an array of integers or reals indexed by `ISET`).

10.6.3 Results

The optimization algorithm finds a minimum total cost of $1,261,000. The table shows the new contracts signed every month and the resulting totals.

Table 10.13: Plan of van rentals

New contracts	January	February	March	April	May	June
3 months	230	0	0	240	0	0
4 months	0	0	210	0	0	0
5 months	0	0	0	0	0	0
Total	430	430	440	450	450	450

Note that the requirement to have no vehicles on contract at the start of July is very distorting.

10.7 References and further material

In our transport example of Section 10.1 (car rental), the availability is equal to the total demand. When dealing with an unbalanced case like an offer that is larger than the demand, the constraints (10.1.2) must be changed into inequalities ensuring that no origin delivers more than it has available. Before

Linear Programming was invented, transport problems had already been studied in the 1930s and 40s, in the USA by Hitchcock [Hit41], in the USSR by Kantorovitch. There are efficient specialized algorithms like the **stepping stone algorithm** [BJ90].

The choice of modes of transport in Section 10.2 is a **minimum cost flow problem** in a graph. Fast algorithms for this problem that work directly on the graph are available. An algorithm with a good simplicity/performance ratio is the one by Busacker and Gowen; a Pascal source is provided in [Pri94a]. Other algorithms are given by Ahuja et al [AMO93].

The problem of depot location (**facility location problem**) in Section 10.3 is a mixed-integer problem with continuous variables for the transported quantities and binary variables for the construction decisions. This problem is NP-hard, even with depots of infinite capacity. Tree-based methods like the one by Erlenkotter [Erl78] are able to solve problems of a certain size (100 customers). A book by Daskin describes a large number of other location problems [Das95].

The heating oil delivery problem in Section 10.4 is a typical case of a **vehicle routing problem** (VRP). The formulation given here is only suitable for small instances (20-30 customers). Some specialized tree search methods are able to solve to optimality instances with up to 100 customers [LDN84] [BMR94], even for the case that the delivery to clients needs to take place in certain time windows [DDS92]. Beyond this limit of 100 customers, it is recommended to use heuristic methods like the one by Clarke and Wright [CW64]. An overview on classical heuristics is given by Christofides [CMT79b]. Metaheuristics like tabu search find good solutions for large sized instances [GHL94].

The mathematical program for the intermodal transport problem in Section 10.5 uses an excessively large number of variables when the number of cities and especially the number of modes of transport grow larger. Luckily, it may be solved efficiently with a Dynamic Programming method (a kind of recursive optimization) [Kas98]. The case studied here is relatively simple because the cities form a unique path. The problem gets very hard for an arbitrary graph.

Like the problem of personnel planning for a construction site in Chapter 14, the van fleet planning problem in Section 10.6 belongs to a category of optimization problems in which the requirements for a period are satisfied by resources that are used during more than one period. Another problem of this type consists of assigning shifts or other tasks to persons that are available for a certain number of hours. The associated mathematical programs have a series of consecutive coefficients of value 1 in their columns. It is possible to show that they are equivalent to minimum cost flow problems [AMO93].

Chapter 11
Air transport

The domain of air transport is a fertile ground for original and difficult optimization problems. The fierce competition between airlines has led to the development of Operations Research departments at the largest ones (such as American Airlines, Delta Airlines, British Airways, and more recently, the renaissance of such a department at Air France). This chapter illustrates the diversity of applications, involving passenger flows, flight crews, aircraft movements, location of **hubs** (flight connection platforms), and flight trajectories.

In the first problem (Section 11.1), the incoming planes at an airport need to be assigned to the flights leaving from the airport in order to minimize the number of passengers (and hence the luggage) who have to change planes. The problem in Section 11.2 consists of forming flight crews according to various compatibility and performance criteria. Section 11.3 describes an interesting and original problem, namely scheduling the landing sequence of planes on a runway. The choice of the architecture of a network (location of hubs) is dealt with in Section 11.4. The subject of Section 11.5 is a case of emergency logistics (the provision of a disaster-stricken country with fresh supplies).

11.1 Flight connections at a hub

The airline SafeFlight uses the airport Roissy-Charles-de-Gaulle as a hub to minimize the number of flight connections to European destinations. Six Fokker 100 airplanes of this airline from Bordeaux, Clermont-Ferrand, Marseille, Nantes, Nice, and Toulouse are landing between 11am and 12:30pm. These aircraft leave for Berlin, Bern, Brussels, London, Rome, and Vienna between 12:30 pm and 13:30 pm. The numbers of passengers transferring from the incoming flights to one of the outgoing flights are listed in Table 11.1.

Table 11.1: Numbers of passengers transferring between the different flights

		Destinations					
		Berlin	**Bern**	**Brussels**	**London**	**Rome**	**Vienna**
Origins	**Bordeaux**	35	12	16	38	5	2
	Clermont-Ferrand	25	8	9	24	6	8
	Marseille	12	8	11	27	3	2
	Nantes	38	15	14	30	2	9
	Nice	–	9	8	25	10	5
	Toulouse	–	–	–	14	6	7

For example, if the flight incoming from Bordeaux continues on to Berlin, 35 passengers and their luggage may stay on board during the stop at Paris. The flight from Nice arrives too late to be re-used on the connection to Berlin, the same is true for the flight from Toulouse that cannot be used for the destinations Berlin, Bern and Brussels (the corresponding entries in the table are marked with '–').

How should the arriving planes be re-used for the departing flights to minimize the number of passengers who have to change planes at Roissy?

11.1.1 Model formulation

Let *PLANES* be the set of aircraft (the number of which also corresponds the numbers of flight origins and flight destinations) and $PASS_{ij}$ the number of passengers transferring at the hub from origin i to the flight with destination j. We introduce binary variables $cont_{ij}$ that take the value 1 if and only if the plane

coming from i continues its journey to destination j. The following LP represents the problem:

$$\text{maximize} \sum_{i \in PLANES} \sum_{j \in PLANES} PASS_{ij} \cdot cont_{ij} \tag{11.1.1}$$

$$\forall j \in PLANES : \sum_{i \in PLANES} cont_{ij} = 1 \tag{11.1.2}$$

$$\forall i \in PLANES : \sum_{j \in PLANES} cont_{ij} = 1 \tag{11.1.3}$$

$$\forall i, j \in PLANES : cont_{ij} \in \{0, 1\} \tag{11.1.4}$$

The initial objective was to minimize the number of passengers that change planes, but this objective is equivalent to the objective (11.1.1) that maximizes the number of passengers staying on board their plane at the hub. The constraints (11.1.2) indicate that every destination is served by exactly one flight, and the constraints (11.1.3) that one and only one flight leaves every origin. The constraints (11.1.4) specify that the variables are binaries. Note that since this problem is an instance of the well known **assignment problem** the optimal LP solution calculated by the simplex algorithm always takes integer values. It is therefore sufficient simply to define non-negativity constraints for these variables. The upper bound of 1 on the variables results from the constraints (11.1.2) and (11.1.3).

11.1.2 Implementation

In the data array `PASS`, that is read in from the file `f1connect.dat` by the following Mosel implementation of the mathematical model, the '–' of the table are replaced by large negative coefficients (–1000). This negative cost prevents the choice of inadmissible flight connections. Another possibility would be to leave the corresponding entries of the array `PASS` undefined and to test in the definition of the variables whether a connection may be used. This formulation has the advantage of using fewer variables if not all flight connections are feasible, but it cannot be used if there are admissible flight connections that are not taken by any passengers.

```
model "F-1 Flight connections"
 uses "mmxprs"

 declarations
  PLANES = 1..6                          ! Set of airplanes

  PASS: array(PLANES,PLANES) of integer ! Passengers with flight connections

  cont: array(PLANES,PLANES) of mpvar   ! 1 if flight i continues to j
 end-declarations

 initializations from 'f1connect.dat'
  PASS
 end-initializations

! Objective: number of passengers on connecting flights
 Transfer:= sum(i,j in PLANES) PASS(i,j)*cont(i,j)

! One incoming and one outgoing flight per plane
 forall(i in PLANES) sum(j in PLANES) cont(i,j) = 1
 forall(j in PLANES) sum(i in PLANES) cont(i,j) = 1

! Solve the problem: maximize the number of passengers staying on board
 maximize(Transfer)

end-model
```

11.1.3 Results

In the optimal solution, 112 passengers stay on board their original plane. The following table lists the corresponding flight connections

Table 11.2: Optimal flight connections

Plane arriving from	continues to destination	Number of passengers
Bordeaux	London	38
Clermont-Ferrand	Bern	8
Marseille	Brussels	11
Nantes	Berlin	38
Nice	Rome	10
Toulouse	Vienna	7

11.2 Composing flight crews

During the Second World War the Royal Air Force (RAF) had many foreign pilots speaking different languages and more or less trained on the different types of aircraft. The RAF had to form pilot/co-pilot pairs ('crews') for every plane with a compatible language and a sufficiently good knowledge of the aircraft type. In our example there are eight pilots. In the following table every pilot is characterized by a mark between 0 (worst) and 20 (best) for his language skills (in English, French, Dutch, and Norwegian), and for his experience with different two-seater aircraft (reconnaissance, transport, bomber, fighter-bomber, and supply planes).

Table 11.3: Ratings of pilots

	Pilot	1	2	3	4	5	6	7	8
Language	**English**	20	14	0	13	0	0	8	8
	French	12	0	0	10	15	20	8	9
	Dutch	0	20	12	0	8	11	14	12
	Norwegian	0	0	0	0	17	0	0	16
Plane type	**Reconnaissance**	18	12	15	0	0	0	8	0
	Transport	10	0	9	14	15	8	12	13
	Bomber	0	17	0	11	13	10	0	0
	Fighter-bomber	0	0	14	0	0	12	16	0
	Supply plane	0	0	0	0	12	18	0	18

A valid flight crew consists of two pilots that both have each at least 10/20 for the same language and 10/20 on the same aircraft type.

Question 1: Is it possible to have all pilots fly?

Subsequently, we calculate for every valid flight crew the sum of their scores for every aircraft type for which both pilots are rated at least 10/20. This allows us to define for every crew the maximum score among these marks. For example, pilots 5 and 6 have marks 13 and 10 on bombers and 12 and 18 on supply planes. The score for this crew is therefore $max(13 + 10, 12 + 18) = 30$.

Question 2: Which is the set of crews with maximum total score?

11.2.1 Model formulation

Let *PILOTS* be the set of pilots. This type of problem is easily modeled through an undirected compatibility graph $G = (PILOTS, ARCS)$. Every node represents a pilot, two nodes p and q are connected by an undirected arc (or **edge**) $a = [p, q]$ if and only if pilots p and q are compatible, that is, they have a language and a plane type in common for which both are rated at least 10/20. The arcs are assigned weights corresponding to the maximum score $SCORE_a$ of the flight crew. Figure 11.1 shows the resulting graph with the scores for question 2.

A valid set of crews corresponds in G to a subset of arcs such that any two among them have no node in common. In Graph Theory, such a set is called a **matching**. For question 1 we are looking for the **maximum cardinality matching**, for question 2 for the **matching with maximum total weight**. The graph suggests we formulate the model as follows.

$$\text{maximize} \sum_{a \in ARCS} CREW_a \cdot fly_a \tag{11.2.1}$$

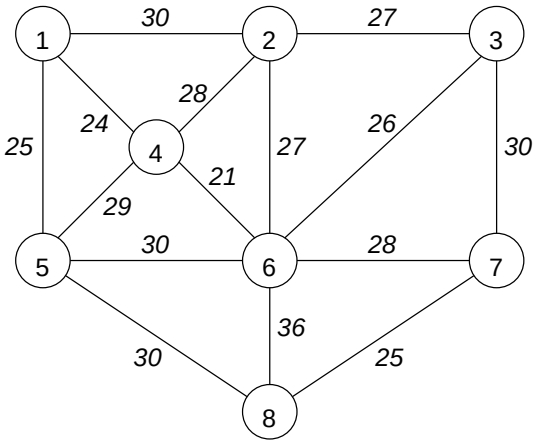

Figure 11.1: Compatibility graph for pilots

$$\forall r \in PILOTS: \sum_{\substack{a=[p,q]\in ARCS \\ p=r \vee q=r}} fly_a \leq 1 \tag{11.2.2}$$

$$\forall a \in ARCS: fly_a \in \{0, 1\} \tag{11.2.3}$$

For every edge $a = [p, q]$ in the graph, a binary variable fly_a indicates whether this edge is used or not (11.2.3). Through constraints (11.2.2) every node r is contained in at most one edge. The objective function (11.2.1) accumulates the weight of the chosen edges for question 2. For question 1, we maximize the number of flight crews to see whether all pilots are taken: the coefficients $SCORE_a$ need to be removed from the objective function. Note that the matching of maximum cardinality is a special case of matching with maximum weight, with all weights equal to 1.

11.2.2 Implementation

The Mosel program below first calculates the set of admissible crews based on the language skills and flight experience of the pilots: for every pair of pilots p and q the algorithm first checks whether they have compatible language skills, and if this is the case then their flight experience is compared. The two pilots are retained as an admissible crew if they both have sufficient experience on the same aircraft type.

The crews are saved in a two-dimensional array `CREW` indexed by the set of arcs, that is, the two pilots p and q of an arc $a = [p, q]$ are represented by `CREW(a,1)` and `CREW(a,2)`. With the convention $p < q$, every pair of pilots is listed only once. The following program solves the problem twice, once with the objective function for question 1 and once for question 2.

```
model "F-2 Flight crews"
 uses "mmxprs"

 forward procedure print_sol

 declarations
  PILOTS = 1..8                      ! Set of pilots
  ARCS: range                        ! Set of arcs representing crews
  RL, RT: set of string              ! Sets of languages and plane types

  LANG: array(RL,PILOTS) of integer  ! Language skills of pilots
  PTYPE: array(RT,PILOTS) of integer ! Flying skills of pilots
  CREW: array(ARCS,1..2) of integer  ! Possible crews
 end-declarations

 initializations from 'f2crew.dat'
  LANG PTYPE
 end-initializations

! Calculate the possible crews
 ct:=1
 forall(p,q in PILOTS| p<q and
                     (or(l in RL) (LANG(l,p)>=10 and LANG(l,q)>=10)) and
```

```
                     (or(t in RT) (PTYPE(t,p)>=10 and PTYPE(t,q)>=10)) ) do
  CREW(ct,1):=p
  CREW(ct,2):=q
  ct+=1
 end-do

 finalize(ARCS)

 declarations
  fly: array(ARCS) of mpvar           ! 1 if crew is flying, 0 otherwise
 end-declarations

! First objective: number of pilots flying
 NFlying:= sum(a in ARCS) fly(a)

! Every pilot is member of at most a single crew
 forall(r in PILOTS) sum(a in ARCS | CREW(a,1)=r or CREW(a,2)=r) fly(a) <= 1

 forall(a in ARCS) fly(a) is_binary

! Solve the problem
 maximize(NFlying)

! Solution printing
 writeln("Number of crews: ", getobjval)
 print_sol

! **** Extend the problem ****
 declarations
  SCORE: array(ARCS) of integer       ! Maximum scores of crews
 end-declarations

 forall(a in ARCS)
  SCORE(a):= max(t in RT | PTYPE(t,CREW(a,1))>=10 and PTYPE(t,CREW(a,2))>=10)
              (PTYPE(t,CREW(a,1)) + PTYPE(t,CREW(a,2)))

! Second objective: sum of scores
 TotalScore:= sum(a in ARCS) SCORE(a)*fly(a)

! Solve the problem
 maximize(TotalScore)

 writeln("Maximum total score: ", getobjval)
 print_sol

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

! Solution printing
 procedure print_sol
  forall(a in ARCS)
   if(getsol(fly(a))>0) then
    writeln(CREW(a,1),  " - ", CREW(a,2))
   end-if
 end-procedure

end-model
```

A feature of Mosel shown by the above implementation is the incremental definition of the array `CREW` that stores the list of crews (and hence of its index set `ARCS`). After the calculation of the crews is completed the (dynamic) index set `ARCS` is finalized so that the array of variables `fly` is defined as a static array on this index set.

Another feature that is new in this model is the cumulative `or` used for testing the pilots' compatibility. The expression

```
 or(l in RL) (LANG(l,p)>=10 and LANG(l,q)>=10)
```

evaluates to `true` if for at least one `l` the values of `LANG(l,p)` and `LANG(l,q)` are both greater than or equal to 10.

11.2.3 Results

In answer to the first question, the program finds that four crews are flying, that is, all eight pilots. For the second question, the optimization calculates a maximum total score of 125 for the four crews [1,2], [3,7], [4,5], and [6,8].

11.3 Scheduling flight landings

The plane movements in large airports are subject to numerous security constraints. The problem presented in this section consist of calculating a schedule for flight landings on a single runway. More general problems have been studied but they are fairly complicated (dynamic cases, for instance through planes arriving late, instances with several runways, *etc.*).

Ten planes are due to arrive. Every plane has an **earliest arrival time** (time when the plane arrives above the zone if traveling at maximum speed) and a **latest arrival time** (influenced among other things by its fuel supplies). Within this time window the airlines choose a **target time**, communicated to the public as the flight arrival time. The early or late arrival of an aircraft with respect to its target time leads to disruption of the airport and causes costs. To take into account these cost and to compare them more easily, a penalty per minute of early arrival and a second penalty per minute of late arrival are associated with every plane. The time windows (in minutes from the start of the day) and the penalties per plane are given in the following table.

Table 11.4: Characteristics of flight time windows

Plane	1	2	3	4	5	6	7	8	9	10
Earliest arrival	129	195	89	96	110	120	124	126	135	160
Target time	155	258	98	106	123	135	138	140	150	180
Latest Arrival	559	744	510	521	555	576	577	573	591	657
Earliness penalty	10	10	30	30	30	30	30	30	30	30
Lateness penalty	10	10	30	30	30	30	30	30	30	30

Due to turbulence and the duration of the time during which a plane is on the runway, a security interval has to separate any two landings. An entry in line p of column q in the following Table 11.5 denotes the minimum time interval (in minutes) that has to lie between the landings of planes p and q, even if they are not consecutive. Which landing schedule minimizes the total penalty subject to arrivals within the given time windows and the required intervals separating any two landings?

Table 11.5: Matrix of minimum intervals separating landings

	1	2	3	4	5	6	7	8	9	10
1	–	3	15	15	15	15	15	15	15	15
2	3	–	15	15	15	15	15	15	15	15
3	15	15	–	8	8	8	8	8	8	8
4	15	15	8	–	8	8	8	8	8	8
5	15	15	8	8	–	8	8	8	8	8
6	15	15	8	8	8	–	8	8	8	8
7	15	15	8	8	8	8	–	8	8	8
8	15	15	8	8	8	8	8	–	8	8
9	15	15	8	8	8	8	8	8	–	8
10	15	15	8	8	8	8	8	8	8	–

11.3.1 Model formulation

Let *PLANES* be the set of planes due to arrive at the airport. A plane p has an arrival time window $[START_p, STOP_p]$ with the target arrival time $TARGET_p$. The penalty $CEARLY_p$ applies per minute of early arrival, and $CLATE_p$ per minute of late arrival. The minimum interval between the landings of two planes p and q is denoted by $DIST_{pq}$. In the given data instance any two time windows are overlapping. We therefore formulate a model for this overlapping situation and describe later how to generalize it for any type of time windows.

We need to define variables $land_p$ for the landing time of every plane p. They are bounded by the earliest

and latest arrival times (11.3.1).

$$\forall p \in PLANES : START_p \leq land_p \leq STOP_p \tag{11.3.1}$$

To account for the interval separating the landing times of any two planes p and q, binary variables $prec_{pq}$ are required with $prec_{pq} = 1$ if the landing of aircraft p precedes the landing of q.

$$\forall p, q \in PLANES, p \neq q : prec_{pq} \in \{0, 1\} \tag{11.3.2}$$

The constraints (11.3.3) specify that p arrives before q or q arrives before p. The constraints (11.3.4) guarantee the separation of the landings; these are classical **disjunctive (exclusion) constraints** that are used for instance in scheduling to prevent the overlapping of two tasks on the same machine. M denotes some large positive constant.

$$\forall p, q \in PLANES, p \neq q : prec_{pq} + prec_{qp} = 1 \tag{11.3.3}$$

$$\forall p, q \in PLANES, p \neq q : land_p + DIST_{pq} - M_{pq} \cdot prec_{qp} \leq land_q \tag{11.3.4}$$

If the plane p arrives before q, we have $prec_{pq} = 1$, and hence $prec_{qp} = 0$ through (11.3.3). The constraint (11.3.4) for p and q results in $land_p + DIST_{pq} \leq land_q$ ensuring the required separation of the two flights. If $prec_{pq} = 0$, then $prec_{qp} = 1$ and the constraint (11.3.4) is trivially satisfied because the left side of the inequality is a large negative value. To avoid problems with numerical stability, the value of M_{pq} should not be chosen too large, $M_{pq} = STOP_p + DIST_{pq} - START_q$ is sufficient for every constraint (11.3.4).

To halve the number of variables, we define the binary variables $prec_{pq}$ with $p < q$ to take the value 1 if p lands before q and 0 otherwise. The constraints (11.3.2) are replaced by (11.3.5) and the constraints (11.3.3) become redundant.

$$\forall p, q \in PLANES, p < q : prec_{pq} \in \{0, 1\} \tag{11.3.5}$$

The constraints (11.3.4) are rewritten to (11.3.6) and (11.3.7). (11.3.6) are the disjunctive constraints for the pairs of planes (p, q) with $p > q$ and (11.3.7) the disjunctions for $p < q$. If for instance, $p < q$ (constraints (11.3.7)) and p arrives before q, then $prec_{pq} = 1$ and hence $1 - prec_{pq} = 0$, and $land_p + DIST_{pq} \leq land_q$. If $p < q$ but q arrives before p, then $prec_{pq} = 0$ and (11.3.7) is trivially satisfied.

$$\forall p, q \in PLANES, q < p : land_p + DIST_{pq} \leq land_q + M \cdot prec_{qp} \tag{11.3.6}$$

$$\forall p, q \in PLANES, p < q : land_p + DIST_{pq} \leq land_q + M \cdot (1 - prec_{pq}) \tag{11.3.7}$$

Taking into account the early or late arrival of a plane p with respect to its targeted arrival time $TARGET_p$ is a delicate matter. We introduce a variable $early_p$ for the earliness and a variable $late_p$ for the lateness. We may then write the objective function (11.3.8) with the penalties per minute of early or late arrival, $CEARLY_p$ and $CLATE_p$.

$$\text{minimize} \sum_{p \in PLANES} (CEARLY_p \cdot early_p + CLATE_p \cdot late_p) \tag{11.3.8}$$

The variables $early_p$ and $late_p$ are bounded from above (constraints (11.3.9) and (11.3.10)) so that the arrival is scheduled within the time window $[START_p, STOP_p]$.

$$\forall p \in PLANES : 0 \leq early_p \leq TARGET_p - START_p \tag{11.3.9}$$

$$\forall p \in PLANES : 0 \leq late_p \leq STOP_p - TARGET_p \tag{11.3.10}$$

The landing time is linked to the earliness or lateness by the constraints (11.3.11).

$$\forall p \in PLANES : land_p = TARGET_p - early_p + late_p \tag{11.3.11}$$

The fact that we are minimizing prevents $early_p$ and $late_p$ from both being non-zero simultaneously in (11.3.11).

We finally obtain a MIP model consisting of the lines (11.3.1), and (11.3.5) – (11.3.11).

11.3.2 Generalization to arbitrary types of time windows

In the general case, there are three sets *OVERLAP*, *SEP*, and *NONSEP* of plane pairs (p, q) with $p < q$. *OVERLAP* contains all plane pairs with overlapping time windows. The variables $prec_{pq}$ are used to decide whether p arrives before q. As a simplification, the preceding model deals with the case that all pairs of time windows are in the set *OVERLAP*.

There may also be a set *SEP* of pairs (p, q) with disjoint time windows and guaranteed separation: for instance the time windows [10,50], [70,110] and a separation time of 15 that will obviously be satisfied. More formally, the relation defining such a pair of time windows is $(STOP_p + DIST_{pq} < START_q) \vee (STOP_q + DIST_{qp} < START_p)$. The third set *NONSEP* of plane pairs that may be defined contains all pairs with disjoint time windows without guaranteed separation, such as [10,50], [70,110] with a separation interval of 30.

To obtain a general model, we need to add the following sets of constraints (11.3.12) to (11.3.15) to the constraints (11.3.6) – (11.3.7) that only concern the plane pairs in *OVERLAP*. The constraints (11.3.12) and (11.3.13) force the variables to be 0 or 1 in the case of disjoint time windows. The constraints (11.3.14) and (11.3.15) are simple precedence constraints as in the stadium construction problem of Chapter 7.

$$\forall (p, q) \in SEP \cup NONSEP, STOP_p < START_q : prec_{pq} = 1 \quad (11.3.12)$$

$$\forall (p, q) \in SEP \cup NONSEP, STOP_q < START_p : prec_{pq} = 0 \quad (11.3.13)$$

$$\forall (p, q) \in NONSEP, STOP_p < START_q : land_p + DIST_{pq} \leq land_q \quad (11.3.14)$$

$$\forall (p, q) \in NONSEP, STOP_q < START_p : land_q + DIST_{qp} \leq land_p \quad (11.3.15)$$

11.3.3 Implementation

The following Mosel program implements the mathematical model of Section 11.3.1. The variables $prec_{pq}$ are defined for all pairs p and q, but only those $prec_{pq}$ with $p < q$ are used in the constraints so that the remainder of these variables (all pairs with $p \geq q$) do not appear in the problem.

```
model "F-3 Landing schedule"
 uses "mmxprs"

 declarations
  PLANES = 1..10                          ! Set of airplanes

  START, STOP: array(PLANES) of integer ! Start, end of arrival time windows
  TARGET: array(PLANES) of integer      ! Planned arrival times
  CEARLY, CLATE: array(PLANES) of integer ! Cost of earliness/lateness
  DIST: array(PLANES,PLANES) of integer ! Minimum interval between planes
  M: array(PLANES,PLANES) of integer    ! Sufficiently large positive values

  prec: array(PLANES,PLANES) of mpvar   ! 1 if plane i precedes j
  land: array(PLANES) of mpvar          ! Arrival time
  early,late: array(PLANES) of mpvar    ! Earliness/lateness
 end-declarations

 initializations from 'f3landing.dat'
  START STOP TARGET CEARLY CLATE DIST
 end-initializations

 forall(p,q in PLANES) M(p,q):= STOP(p) + DIST(p,q) - START(q)

! Objective: total penalty for deviations from planned arrival times
 Cost:= sum(p in PLANES) (CEARLY(p)*early(p) + CLATE(p)*late(p))

! Keep required intervals between plan arrivals
 forall(p,q in PLANES | p>q)
  land(p) + DIST(p,q) <= land(q) + M(p,q)*prec(q,p)
 forall(p,q in PLANES | p<q)
  land(p) + DIST(p,q) <= land(q) + M(p,q)*(1-prec(p,q))

! Relations between earliness, lateness, and effective arrival time
 forall(p in PLANES) do
  early(p) >= TARGET(p) - land(p)
  late(p) >= land(p) - TARGET(p)
  land(p) = TARGET(p) - early(p) + late(p)
 end-do

 forall(p in PLANES) do
  START(p) <= land(p); land(p) <= STOP(p)
  early(p) <= TARGET(p)-START(p)
  late(p) <= STOP(p)-TARGET(p)
 end-do

 forall(p,q in PLANES | p<q) prec(p,q) is_binary

! Solve the problem
```

```
 minimize(Cost)

end-model
```

As mentioned earlier in this book, the obvious pair START – END for naming the beginning and end of the time windows cannot be used because END is a reserved word in Mosel (see Section 5.2.3 for the complete list of reserved words).

11.3.4 Results

The program calculates a total deviation cost of 700. The following table lists the scheduled arrivals together with the targeted arrival times and the resulting deviations (planes arriving earlier or later than the announced target time). It may be worth mentioning that the LP solution to this problem has an objective value of 0, with many fractional variables $prec_{pq}$.

Table 11.6: Arrival times and deviations

Plane	1	2	3	4	5	6	7	8	9	10
Scheduled arrival	165	258	89	106	118	126	134	142	150	180
Target time	155	258	98	106	123	135	138	140	150	180
Deviation	10	0	0	0	-5	-9	-4	2	0	0

11.4 Airline hub location

The airline FAL (French Air Lines) specializes in freight transport. The company links the major French cities with cities in the United States, namely: Atlanta, Boston, Chicago, Marseille, Nice, and Paris. The average quantities in tonnes transported every day by this company between these cities are given in the following table.

Table 11.7: Average quantity of freight transported between every pair of cities

	Atlanta	Boston	Chicago	Marseille	Nice	Paris
Atlanta	0	500	1000	300	400	1500
Boston	1500	0	250	630	360	1140
Chicago	400	510	0	460	320	490
Marseille	300	600	810	0	820	310
Nice	400	100	420	730	0	970
Paris	350	1020	260	580	380	0

We shall assume that the transport cost between two cities i and j is proportional to the distance that separates them. The distances in miles are given in the next table.

Table 11.8: Distances between pairs of cities

	Boston	Chicago	Marseille	Nice	Paris
Atlanta	945	605	4667	4749	4394
Boston		866	3726	3806	3448
Chicago			4471	4541	4152
Marseille				109	415
Nice					431

The airline is planning to use two cities as connection platforms (**hubs**) to reduce the transport costs. Every city is then assigned to a single hub. The traffic between cities assigned to a given hub H_1 to the cities assigned to the other hub H_2 is all routed through the single connection from H_1 to H_2 which allows the airline to reduce the transport cost. We consider that the transport cost between the two hubs decreases by 20%. Determine the two cities to be chosen as hubs in order to minimize the transport cost.

11.4.1 Model formulation

We write *CITIES* for the set of cities and *NHUBS* for the number of hubs. Let $DIST_{ij}$ be the distance between two cities i and j and $QUANT_{ij}$ the quantity to be transported from i to j. The transport cost per tonne of

freight depends on the cities that are chosen as hubs. The freight to be transported from any city i to any city j transits through two (not necessarily distinct) hubs k and l. Let $COST_{ijkl}$ be the transport cost from i to j through the hubs k and l. This cost is equal to the transport cost from i to k, plus the cost from k to l, plus the transport cost from l to j. The cost from k to l corresponds to 80% of the normal cost from k to l displayed in Table 11.8 since this is an inter-hub transport.

Let the binary variable $flow_{ijkl}$ be 1 if the freight from i to j is transported via the hubs k and l in this order, and 0 otherwise. We also introduce variables hub_i that are 1 if city i is a hub and 0 otherwise.

The objective function (11.4.1) minimizes the total transport cost. The constraint (11.4.2) indicates that we wish to create exactly *NHUBS* hubs. Through the constraints (11.4.3) every pair of cities (i, j) is assigned to a single pair of hubs. The constraints (11.4.4) and (11.4.5) imply that if a variable $flow_{ijkl}$ is at 1, then the variables hub_k and hub_l are 1. In other words, any freight to be transported from i to j may only transit through k and l if both, k and l, are hubs. To complete the model, the constraints (11.4.6) and (11.4.7) define the variables as binaries.

Remark: in the constraints (11.4.3) k may be equal to l which this means that freight may transit via a single hub. This is the case if two origin/destination cities are assigned to the same hub. The inter-hub transport cost is 0 in this case.

$$\text{minimize} \sum_{i \in CITIES} \sum_{j \in CITIES} \sum_{k \in CITIES} \sum_{l \in CITIES} COST_{ijkl} \cdot QUANT_{ij} \cdot flow_{ijkl} \tag{11.4.1}$$

$$\sum_{i \in CITIES} hub_i = NHUBS \tag{11.4.2}$$

$$\forall i, j \in CITIES : \sum_{k \in CITIES} \sum_{l \in CITIES} flow_{ijkl} = 1 \tag{11.4.3}$$

$$\forall i, j, k, l \in CITIES : flow_{ijkl} \leq hub_k \tag{11.4.4}$$

$$\forall i, j, k, l \in CITIES : flow_{ijkl} \leq hub_l \tag{11.4.5}$$

$$\forall i \in CITIES : hub_i \in \{0, 1\} \tag{11.4.6}$$

$$\forall i, j, k, l \in CITIES : flow_{ijkl} \in \{0, 1\} \tag{11.4.7}$$

11.4.2 Implementation

In the following Mosel program implementation of the mathematical model, first the cost $COST_{ijkl}$ for freight transport from i to j via the hubs k and l is calculated, using the inter-hub transport reduction factor *FACTOR* that is defined in the data file.

```
model "F-4 Hubs"
 uses "mmxprs"

 declarations
  CITIES = 1..6                          ! Cities
  NHUBS = 2                              ! Number of hubs

  COST: array(CITIES,CITIES,CITIES,CITIES) of real ! (i,j,k,l) Transport cost
                                         ! from i to j via hubs k and l
  QUANT: array(CITIES,CITIES) of integer ! Quantity to transport
  DIST: array(CITIES,CITIES) of integer  ! Distance between cities
  FACTOR: real                           ! Reduction of costs between hubs

  flow: array(CITIES,CITIES,CITIES,CITIES) of mpvar ! flow(i,j,k,l)=1 if
                                         ! freight from i to j goes via k & l
  hub: array(CITIES) of mpvar            ! 1 if city is a hub, 0 otherwise
 end-declarations

 initializations from 'f4hub.dat'
  QUANT DIST FACTOR
 end-initializations

! Calculate costs
 forall(i,j,k,l in CITIES)
  COST(i,j,k,l):= DIST(i,k)+FACTOR*DIST(k,l)+DIST(l,j)

! Objective: total transport cost
 Cost:= sum(i,j,k,l in CITIES) QUANT(i,j)*COST(i,j,k,l)*flow(i,j,k,l)

! Number of hubs
```

```
 sum(i in CITIES) hub(i) = NHUBS

! One hub-to-hub connection per freight transport
 forall(i,j in CITIES) sum(k,l in CITIES) flow(i,j,k,l) = 1

! Relation between flows and hubs
 forall(i,j,k,l in CITIES) do
  flow(i,j,k,l) <= hub(k)
  flow(i,j,k,l) <= hub(l)
 end-do

 forall(i in CITIES) hub(i) is_binary
 forall(i,j,k,l in CITIES) flow(i,j,k,l) is_binary

! Solve the problem
 minimize(Cost)

end-model
```

11.4.3 Revised formulation

The formulation of the mathematical model in lines (11.4.1) to 11.4.7), and as a consequence its implementation with Mosel, uses a large number of variables for this relatively small problem (more than 1300 binaries). Looking at the distance data more carefully, one can see that the six airports are clustered geographically: the American airports are relatively close to each other and far from the European ones, that are again close to each other. We may therefore reasonably assume that one hub will be located in the US and one in Europe. The American airports will be connected via the US hub and the European airports will use the European hub. This implies that we are able to exclude a large number of index-combinations (*e.g.* EU airports connected to a US hub). We define the sets *EU* of European airports and *US* of US airports, the union of which is the set *CITIES* of our previous model formulation.

In addition to the geographical considerations, we may reduce even further the number of variables by accumulating the quantities to be transported between any pair of destinations to a single value and only working with destination pairs i and j where $i < j$.

The index tuples for variables *flow* that remain after these operations are: $\forall i,j,k \in US, i < j : (i,j,k,k)$ (intra-American flights) $\forall i,j,k \in EU, i < j : (i,j,k,k)$ (intra-European flights), and $\forall i,k \in US, j,l \in EU : (i,j,k,l)$ (inter-continental flights). For our problem there are just more than 100 variables.

All constraints of the previous problem remain the same except that we only sum over the variables that have been defined. We may even add two additional sets of constraints based on the observation that the intra-American (11.4.8) and intra-European (11.4.9) flights will use only a single hub:

$$\forall i,j \in US, i < j : \sum_{k \in US} flow_{ijkk} = 1 \tag{11.4.8}$$

$$\forall i,j \in EU, i < j : \sum_{k \in EU} flow_{ijkk} = 1 \tag{11.4.9}$$

The Mosel implementation of the revised formulation is given below. Note that the array of decision variables `flow` is defined as a `dynamic array` so that we only create the entries that are required. Another change from the previous model is the replacement of the `COST` array by a function to reduce the storage space required by the data (for larger problem sizes than ours this may be a significant advantage).

```
model "F-4 Hubs (2)"
 uses "mmxprs"

 forward function calc_cost(i,j,k,l:integer):real

 declarations
  US = 1..3; EU = 4..6
  CITIES = US + EU                        ! Cities
  NHUBS = 2                               ! Number of hubs

  QUANT: array(CITIES,CITIES) of integer ! Quantity to transport
  DIST: array(CITIES,CITIES) of integer  ! Distance between cities
  FACTOR: real                           ! Reduction of costs between hubs

  flow: dynamic array(CITIES,CITIES,CITIES,CITIES) of mpvar
```

```
                                        ! flow(i,j,k,l)=1 if freight
                                        ! from i to j goes via k and l
  hub: array(CITIES) of mpvar           ! 1 if city is a hub, 0 otherwise
 end-declarations

 initializations from 'f4hub.dat'
  QUANT DIST FACTOR
 end-initializations

 forall(i,j in CITIES | i<j) QUANT(i,j):=QUANT(i,j)+QUANT(j,i)

 forall(i,j,k in US | i<j) create(flow(i,j,k,k))
 forall(i,j,k in EU | i<j) create(flow(i,j,k,k))
 forall(i,k in US, j,l in EU) create(flow(i,j,k,l))

! Objective: total transport cost
 Cost:= sum(i,j,k,l in CITIES | exists(flow(i,j,k,l)))
                  QUANT(i,j)*calc_cost(i,j,k,l)*flow(i,j,k,l)

! Number of hubs
 sum(i in CITIES) hub(i) = NHUBS

! One hub-to-hub connection per freight transport
 forall(i,j in CITIES | i<j) sum(k,l in CITIES) flow(i,j,k,l) = 1
 forall(i,j in US | i<j) sum(k in US) flow(i,j,k,k) = 1
 forall(i,j in EU | i<j) sum(k in EU) flow(i,j,k,k) = 1

! Relation between flows and hubs
 forall(i,j,k,l in CITIES | exists(flow(i,j,k,l))) do
  flow(i,j,k,l) <= hub(k)
  flow(i,j,k,l) <= hub(l)
 end-do

 forall(i in CITIES) hub(i) is_binary
 forall(i,j,k,l in CITIES | exists(flow(i,j,k,l))) flow(i,j,k,l) is_binary

! Solve the problem
 minimize(Cost)

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

! Transport cost from i to j via hubs k and l
 function calc_cost(i,j,k,l:integer):real
  returned:=DIST(i,k)+FACTOR*DIST(k,l)+DIST(l,j)
 end-function

end-model
```

11.4.4 Results

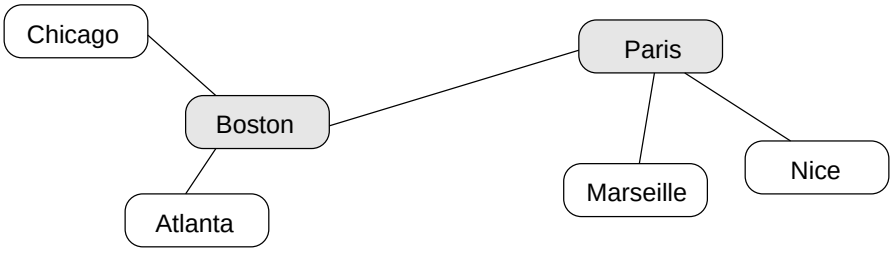

Figure 11.2: Choice of hubs

Both models have the same solution. To minimize the total cost of transport the cities Boston and Paris have to be used as hubs. Not surprisingly, Atlanta and Chicago are assigned to the hub Boston, Nice and Marseille to Paris. This means, the freight from Atlanta, Boston, and Chicago is gathered at the hub Boston. There, any freight destined for any of the American cities is directly sent to its destination. All merchandise for European destinations is first shipped to the hub at Paris and from there distributed to the European cities. The freight originating from Europe and with destination USA first transits through the hub at Paris and then through Boston before arriving at its destination. The total cost of transport is

€ 42,153,794.

11.5 Planning a flight tour

A country in south-east Asia is experiencing widespread flooding. The government, with international help, decides to establish a system of supply by air. Unfortunately, only seven runways are still in a usable state, among which is the one in the capital.

The government decides to make the planes leave from the capital, have them visit all the other six airports and then come back to the capital. The following table lists the distances between the airports. Airport A1 is the one in the capital. In which order should the airports be visited to minimize the total distance covered?

Table 11.9: Distance matrix between airports (in km)

	A2	A3	A4	A5	A6	A7
A1	786	549	657	331	559	250
A2		668	979	593	224	905
A3			316	607	472	467
A4				890	769	400
A5					386	559
A6						681

11.5.1 Model formulation

To model this problem, we denote by $CITIES = \{1, \ldots, 7\}$ the set of airports to be served and $DIST_{ij}$ the distance between the airports A_i and A_j. Since the journey is carried out by plane, the distance matrix is symmetric. We use binary variables fly_{ij} that take the value 1 if the plane flies from i to j. The tour we are looking for is a **cycle** that visits every airport once and once only. At every airport i, the cycle comes from a single predecessor and continues to a single successor airport. The constraints (11.5.1) and (11.5.2) express these conditions.

$$\forall j \in CITIES : \sum_{i \in CITIES, i \neq j} fly_{ij} = 1 \tag{11.5.1}$$

$$\forall i \in CITIES : \sum_{j \in CITIES, i \neq j} fly_{ij} = 1 \tag{11.5.2}$$

If we only use the constraints (11.5.1) and (11.5.2), the cycle may be fragmented into several **sub-cycles**, for instance going from airport 1 to airport 2 and from there back to 1. This sub-cycle satisfies these two constraints but does not lead to a valid solution. It is therefore necessary to prevent the forming of sub-cycles from occurring. This is the aim of the constraints (11.5.3): every sub-cycle that appears in a subset S of airports is 'broken' by requiring that the number of arcs in S must be strictly less than the cardinality of S.

If N is the number of cities, for even a relatively small N it is simply impossible to test all sub-cycles that may be formed (their number is in the order of 2^N). To start with, we are therefore going to solve the problem without the constraints (11.5.3), which usually leads to a solution comprising sub-cycles, and then one by one add constraints that render the forming of the sub-cycles impossible. This procedure will be described in more detail in the next section.

$$\forall S : \sum_{(i,j) \in S} fly_{ij} \leq |S| - 1 \tag{11.5.3}$$

The objective function (11.5.4) is the total length of the cycle passing through all cities, that is the length of all arcs that are used.

$$\text{minimize} \sum_{i \in CITIES} \sum_{j \in CITIES} DIST_{ij} \cdot fly_{ij} \tag{11.5.4}$$

$$\forall j \in CITIES : \sum_{i \in CITIES, i \neq j} fly_{ij} = 1 \tag{11.5.5}$$

$$\forall i \in CITIES : \sum_{j \in CITIES, i \neq j} fly_{ij} = 1 \tag{11.5.6}$$

$$\forall S : \sum_{(i,j)\in S} fly_{ij} \leq |S| - 1 \tag{11.5.7}$$

$$\forall i, j \in CITIES : fly_{ij} \in \{0, 1\} \tag{11.5.8}$$

11.5.2 Implementation and results

The following Mosel program corresponds to the mathematical model without the constraints (11.5.3). Since the distance matrix is symmetric, the data file only contains the upper triangle (distance from i to j for which $i < j$); the other half is completed after reading in the data.

```
model "F-5 Tour planning"
 uses "mmxprs"

 declarations
  NCITIES = 7
  CITIES = 1..NCITIES                    ! Cities

  DIST: array(CITIES,CITIES) of integer  ! Distance between cities
  NEXTC: array(CITIES) of integer        ! Next city after i in the solution

  fly: array(CITIES,CITIES) of mpvar     ! 1 if flight from i to j
 end-declarations

 initializations from 'f5tour.dat'
  DIST
 end-initializations

 forall(i,j in CITIES | i<j) DIST(j,i):=DIST(i,j)

! Objective: total distance
 TotalDist:= sum(i,j in CITIES | i<>j) DIST(i,j)*fly(i,j)

! Visit every city once
 forall(i in CITIES) sum(j in CITIES | i<>j) fly(i,j) = 1
 forall(j in CITIES) sum(i in CITIES | i<>j) fly(i,j) = 1

 forall(i,j in CITIES | i<>j) fly(i,j) is_binary

! Solve the problem
 minimize(TotalDist)

end-model
```

The solution to this problem is represented in Figure 11.3. It contains three subcycles that we need to exclude. The strategy to adopt suppresses one sub-cycle at a time, starting with the smallest one.

We have implemented the following procedure `break_subtour` that is added to the Mosel program above and called after the solution of the initial problem. The procedure first saves the successor of every city into the array `NEXTC`. It then calculates the set of cities that are on the tour starting from city 1: if the size of the set corresponds to the total number of cities `NCITIES`, we have found the optimal solution and the algorithm stops, otherwise there are subtours.

The algorithm then enumerates all subtours to find the smallest one: if a subtour with two cities is found we can stop (there are no subtours of length 1), otherwise the tour starting from the next city that has not yet been enumerated as part of a tour (set `ALLCITIES`) is calculated. When the smallest subtour has been found, we add a constraint of type (11.5.3) for this set and re-solve the problem.

```
 procedure break_subtour
  declarations
   TOUR,SMALLEST,ALLCITIES: set of integer
  end-declarations

  forall(i in CITIES)
   NEXTC(i):= integer(round(getsol(sum(j in CITIES) j*fly(i,j) )))

! Get (sub)tour containing city 1
  TOUR:={}
  first:=1
  repeat
```

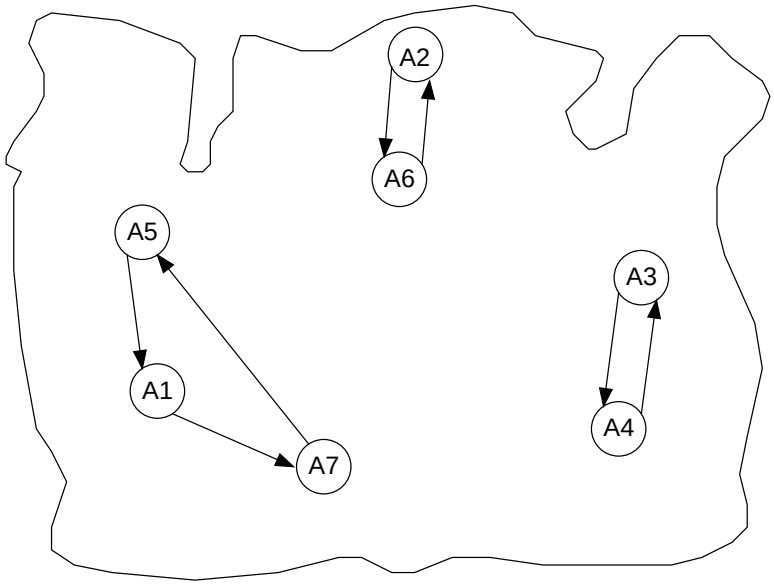

Figure 11.3: Initial solution with three sub-cycles

```
   TOUR+={first}
   first:=NEXTC(first)
  until first=1
  size:=getsize(TOUR)

! Find smallest subtour
  if size < NCITIES then
   SMALLEST:=TOUR
   if size>2 then
    ALLCITIES:=TOUR
    forall(i in CITIES) do
     if(i not in ALLCITIES) then
      TOUR:={}
      first:=i
      repeat
       TOUR+={first}
       first:=NEXTC(first)
      until first=i
      ALLCITIES+=TOUR
      if getsize(TOUR)<size then
       SMALLEST:=TOUR
       size:=getsize(SMALLEST)
      end-if
      if size=2 then
       break
      end-if
     end-if
    end-do
   end-if

! Add a subtour breaking constraint
   sum(i in SMALLEST) fly(i,NEXTC(i)) <= getsize(SMALLEST) - 1

! Re-solve the problem
   minimize(TotalDist)

! New round of subtour elimination
   break_subtour
  end-if
 end-procedure
```

During its first execution the subtour elimination procedure adds the following constraint to the model to eliminate the subtour (2,6,2):

```
 fly(2,6) + fly(6,2) <= 1
```

With this additional constraint we obtain the solution represented in the following figure: it still consists of three (different) subtours.

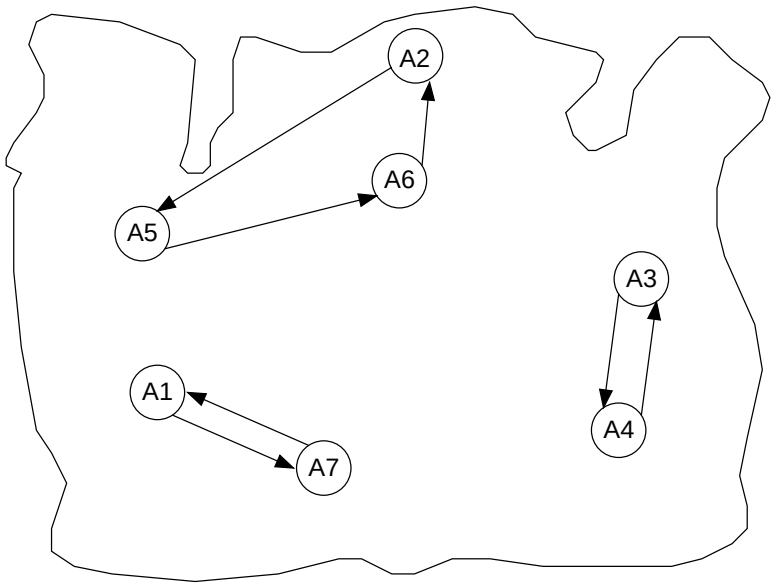

Figure 11.4: Second solution

When the subtour elimination procedure is called again, it adds a constraint to exclude the subcycle (1,7,1):

```
fly(1,7) + fly(7,1) <= 1
```

After this second execution, we finally obtain a single tour with a total length of 2575 km. The arcs that are directed in the model may be replaced by undirected arcs (edges) without any consequence for the solution since the distance matrix is symmetric.

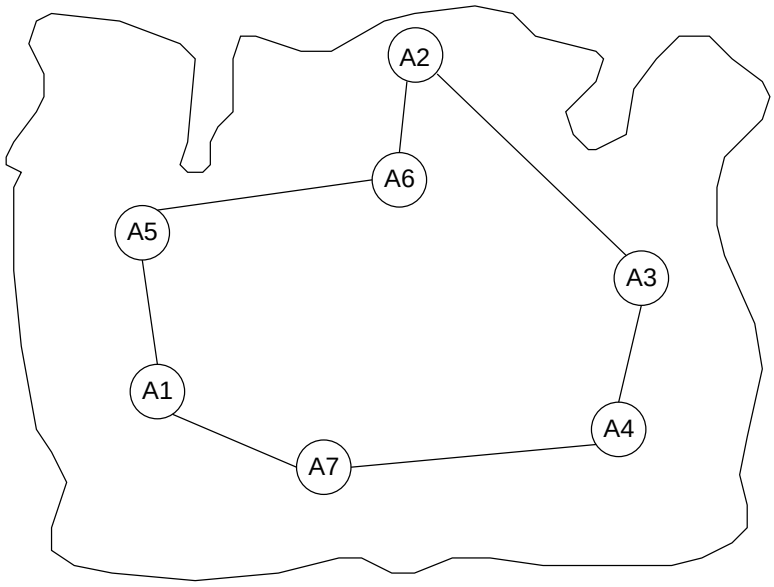

Figure 11.5: Optimal solution

The Mosel implementation of the subtour elimination procedure uses a certain number of set operators and other functionality related to sets: the function `getsize` returns the size (= number of elements) of a set or array. `{}` denotes the empty set. A set may be assigned using the usual assignment operator `:=`, *e.g.* `TOUR:={}`, and a set is added to another set using the operator `+=`.

11.6 References and further material

The problem of flight connections in Section 11.1 is a classical assignment problem. This problem is well solved by the Hungarian algorithm described, for instance, by Papadimitriou [PS98] or by techniques searching for the maximum flow in a transport network [Pri94a].

The composition of flight crews in Section 11.2 is a problem of matching in an arbitrary graph whilst the assignment of personnel to workposts in Chapter 14 is a matching problem in a **bipartite graph**: the edges connect nodes of two types, the persons and the workposts. The matrix of the linear program is totally unimodular in the bipartite case, which means that the variables automatically take the values 0 or 1 at the optimum. In the non-bipartite case of the flight crew, the simplex algorithm may find solutions with variables at 0.5 in the optimum. The problems of maximum cardinality or maximum weight matching in an arbitrary graph (bipartite or not) are all polynomial. An algorithm in $O(n^3)$ (with n the number of nodes) with a Fortran code is given in Minoux et al. [MB86]. A simpler algorithm for maximum cardinality matching, but still of complexity $O(n^3)$ is given by Syslo [SDK83] with a Pascal source code.

The problem of scheduling flight landings in Section 11.3 represents a class of scheduling problems with time windows and distance constraints between tasks. It has been studied by Beasley et al. [BKSA00] for the case of multiple runways. The authors describe a specialized tree search method for large size instances. They are able to solve realistic instances with 50 planes and four runways to optimality. Beasley has also published an interesting article [BC96] concerning crew scheduling. Based on a set of flights with given dates and durations this problem consists of defining the sequence of flights for the crews in order to cover all the flights but without exceeding a limited duration of work.

With the formulation presented in Section 11.4 for the hub location problem, the number of variables used may be very large. If it is possible to transform any city into a hub, the number of the variables $flow_{ijkl}$ is in the order of $O(n^4)$ (with n the number of cities). And hence, a problem with just over thirty cities would require a million variables. In Daskin [Das95] and Campbell [Cam96] several heuristics for this type of problem are given.

The problem in Section 11.5 is the famous **Traveling Salesman Problem** (TSP). It consists of organizing a tour that visits every node of a graph once and only once, whilst minimizing the total cost of the arcs used on the tour. The flight tour planning problem presented here is **symmetrical** because the cost matrix is a symmetrical distance matrix. The paint production problem in Chapter 7 is an asymmetrical TSP. It is solved via a direct formulation, that is, without progressively adding constraints. The TSP is such a well known NP-hard problem problem that it is used for testing new optimization techniques. The CD that comes with the book contains a data file with a distance matrix for several French cities and a slightly modified version of the Mosel program of Section 11.5, adapted to the structure of this file.

The TSP is so hard that the MIP models of this book take a very long time to solve for more than twenty cities. Beyond this number, it becomes necessary to use specialized tree search methods [HK70]. Pedagogical examples of such methods are given by Roseaux [Ros85] and Evans [EM92], and a Pascal code is given by Syslo [SDK83]. Certain Euclidean problems with hundreds of cities can be dealt with by the method called **Branch and Cut** [GH91]. Otherwise, one has to give up trying to find optimal solutions and use heuristic techniques, some of which, like tabu search or simulated annealing, are very efficient in practice. Such heuristics are described in detail with listings of Pascal code in the book about graphs by Prins [Pri94a].

Chapter 12
Telecommunication problems

In recent years, the domain of telecommunications has experienced an explosive growth. It is extraordinarily rich in original optimization problems, which explains the fact that this book devotes an entire chapter to this topic. The short term exploitation of telecom networks are problems of on-line control and problems tackled by the theory of telecommunication protocols. But design problems, the sizing or location of resources, and planning are well suited to optimization approaches. These are therefore described here.

The design of a network needs to satisfy the traffic forecasts between nodes, minimize the construction costs and fulfill conditions on its reliability. It starts with the choice of the location of nodes. Section 12.6 deals with a mobile phone network in which transmitters need to be placed to cover the largest possible part of the population within given budget limits. Once the nodes are chosen, one needs to select the pairs of nodes that will be connected with direct lines for the transmission of data. In Section 12.4, a cable network with a tree structure is constructed with the objective of minimizing the total cost. The problem in Section 12.2 studies the connection of cells to a main ring in a mobile phone network with capacity constraints. Reliability considerations demand multiple connections between every cell and the ring.

Other optimization problems arise in the analysis and the exploitation of existing networks. A natural question dealt with in the problem of Section 12.3 is to know the maximum traffic that a network with known capacity limits may support. The problem in Section 12.1 concerns the reliability of the data exchange between two nodes of a given network. This reliability is measured by the number k of disjoint paths (that is, paths without any intermediate node in common) between two nodes. If $k > 1$, then communication is still possible even if $k - 1$ nodes break down. The problem in Section 12.5 is an example of traffic planning: the data packets in the repeater of a telecommunications satellite need to be scheduled.

12.1 Network reliability

We consider the military telecommunications network represented in Figure 12.1. This network consists of eleven sites connected by bidirectional lines for data transmission.

For reliability reasons in the case of a conflict, the specifications require that the two sites (nodes) 10 and 11 of the network remain able to communicate even if any three other sites of the network are destroyed. Does the network in Figure 12.1 satisfy this requirement?

12.1.1 Model formulation

This problem can be converted into a maximum flow problem. The latter is dealt with in detail in Chapter 15 (Problem of water supply), but we recap here briefly. Let a directed graph be given as G = (*NODES, ARCS*), where *NODES* is a set of nodes, *ARCS* a set of arcs. An arc connecting node n to node m is written (n, m) and has an integer capacity of CAP_{nm}.

Imagine that a liquid enters the network at a given node *SOURCE* and leaves it at another node *SINK*. By spreading over the network, this liquid creates a flow that can be defined as a flow $flow_{nm}$ in every arc (n, m). This flow is valid if it stays within the capacity limits of the arcs and if it does not incur any loss when passing through intermediate nodes (that is, nodes other than *SOURCE* and *SINK*). The maximum flow problem consists of finding a flow that maximizes the total throughput injected at *SOURCE* (or arriving at *SINK* since there are no losses). If this problem is formulated as an LP, the simplex algorithm automatically finds an optimal solution with integer flows.

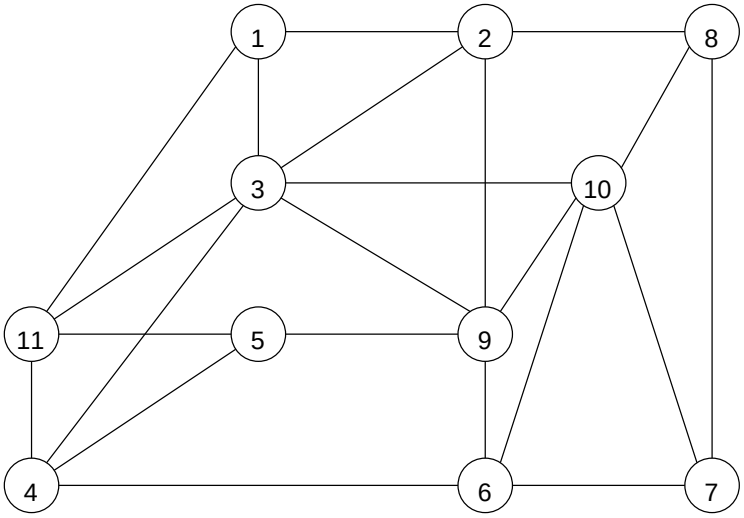

Figure 12.1: Telecommunications network

At first sight, our reliability problem does not quite resemble the maximum flow problem. Let us imagine a modified network H with capacities equal to 1 on all connections: the optimal flows on arcs therefore will take the values 0 or 1. We also impose a capacity of 1 at every node: through this, two units of flow leaving $S = 10$ through two different arcs will follow two paths to $T = 11$, without passing through any common node. Such paths are called **node-disjunctive**.

Since a flow on every arc leaving *SOURCE* is either 0 or 1, the maximum number of disjunctive paths from *SOURCE* to *SINK* in G is equal to the throughput of a maximum flow in the modified graph H. If we find k paths, the network is resilient to $k-1$ broken nodes: in the worst case, these nodes will be on $k-1$ disjunctive paths, but communication will remain possible via the one remaining path. In the end, the answer to our reliability problem consists of comparing $k-1$ with the maximum number of node breakdowns allowed by the specifications.

The only minor difficulty is that the flow problem is usually defined in a directed graph. But in our telecommunications network the bidirectional connections are not directed. To distinguish the flow in both senses on a single connection between the nodes n and m, we replace the connection by two arcs (n, m) and (m, n). We therefore obtain a directed graph $G = (NODES, ARCS)$ that allows us to formulate the problem in a simple manner.

$$\text{maximize} \sum_{m \text{ succ.of } SOURCE} flow_{SOURCE,m} \tag{12.1.1}$$

$$\forall n \neq SOURCE, SINK : \sum_{m \text{ succ.of } n} flow_{nm} = \sum_{m \text{ pred.of } n} flow_{mn} \tag{12.1.2}$$

$$\forall n \neq SOURCE, SINK : \sum_{m \text{ succ.of } n} flow_{nm} \leq 1 \tag{12.1.3}$$

$$\sum_{m \text{ pred.of } SOURCE} flow_{m,SOURCE} = 0 \tag{12.1.4}$$

$$\forall (n, m) \in ARCS : flow_{nm} \in \{0, 1\} \tag{12.1.5}$$

(12.1.6)

The objective function (12.1.1) is to maximize the total throughput, the sum of all flows on the arcs leaving *SOURCE* (we could also sum the flows arriving at *SINK*). The constraints (12.1.2) model the flow conservation law at every intermediate node n (also called Kirchhoff's law): the total flow arriving from the predecessors must be equal to the total flow leaving to the successors. The constraints (12.1.3) limit the flow through every intermediate node to 1; we consider here the flow leaving every node.

The constraint (12.1.4) is necessary to avoid the flow injected at *SOURCE* returning to this node. It is not required in the water supply problem in Chapter 15 because in this problem the source has no predecessors. And finally, the constraints (12.1.5) specify that all flow variables $flow_{nm}$ are binary. Since this is a maximum flow problem, the LP solution will automatically have integer values. Furthermore, the constraints (12.1.3) imply that the variables are upper bounded by 1. The constraints (12.1.5) could therefore be replaced by the usual non-negativity constraints.

12.1.2 Implementation

The following Mosel program implements the mathematical model. A graph with N nodes may be encoded in two different ways: in the form of a **list of arcs** or as an **adjacency matrix** . The first method is employed in this book, for example, in the line balancing problem in Section 7.6 and the problem of forming flight crews in Section 11.2. We use here the matrix representation that is also used in the problem of gritting roads in Section 15.4.

The adjacency matrix is a binary matrix ARC of size $N \times N$, with $ARC_{nm} = 1$ if and only if the arc (n, m) exists. This array is read from the data file in sparse format, that is, only the non-zero elements or ARC are given, in the form `(n m) ARC(n,m)`. The entries of ARC that are not defined have the default value 0 if they are addressed in the program, but we simply test for existing entries of ARC using the Mosel function `exists` that enumerates the array in a very efficient way, especially if only relatively few of the N^2 possible entries are indeed defined.

To obtain a more compact model and hence a smaller LP that solves faster, only the variables $flow_{nm}$ that correspond to existing arcs are defined. Note that it is not necessary to repeat this condition in every sum over these variables.

The data file contains the representation of the graph in an undirected form: the bidirectional connection between n and m only appears in one sense in the file (with the convention $n < m$). We therefore construct the oriented version of the graph that is required for the maximum flow formulation: for every defined arc (n, m) we define its opposite (m, n). The rest of the model is a straight translation from the mathematical formulation. This model is valid for any choice of *SOURCE* and *SINK* among the nodes of the network.

```
model "G-1 Network reliability"
 uses "mmxprs"

 declarations
  NODES: range                        ! Set of nodes
  SOURCE = 10; SINK = 11              ! Source and sink nodes

  ARC: array(NODES,NODES) of integer  ! 1 if arc defined, 0 otherwise

  flow: array(NODES,NODES) of mpvar   ! 1 if flow on arc, 0 otherwise
 end-declarations

 initializations from 'g1rely.dat'
  ARC
 end-initializations

 forall(n,m in NODES | exists(ARC(n,m)) and n<m ) ARC(m,n):= ARC(n,m)
 forall(n,m in NODES | exists(ARC(n,m)) ) create(flow(n,m))

! Objective: number of disjunctive paths
 Paths:= sum(n in NODES) flow(SOURCE,n)

! Flow conservation and capacities
 forall(n in NODES | n<>SOURCE and n<>SINK) do
  sum(m in NODES) flow(m,n) = sum(m in NODES) flow(n,m)
  sum(m in NODES) flow(n,m) <= 1
 end-do

! No return to SOURCE node
 sum(n in NODES) flow(n,SOURCE) = 0

 forall(n,m in NODES | exists(ARC(n,m)) )  flow(n,m) is_binary

! Solve the problem
 maximize(Paths)

! Solution printing
 writeln("Total number of paths: ", getobjval)

 forall(n in NODES | n<>SOURCE and n<>SINK)
  if(getsol(flow(SOURCE,n))>0) then
   write(SOURCE, " - ",n)
   nnext:=n
   while (nnext<>SINK) do
    nnext:=round(getsol(sum(m in NODES) m*flow(nnext,m)))
```

```
    write(" - ", nnext)
   end-do
   writeln
  end-if

end-model
```

The solution display for this example uses a small algorithm to print all paths between the *SOURCE* and *SINK* nodes: for every node to which there is a flow from the *SOURCE* node, we calculate the chain of its successors until we arrive at the *SINK* node. To obtain the successor of a node *n*, we exploit the fact that every intermediate node only appears in a single path and hence only has a single successor. We calculate the index number of this successor as the sum of flows leaving node *n* multiplied by the node index. The result of `getsol` is type `real` and needs to be transformed to an integer using the function `round`.

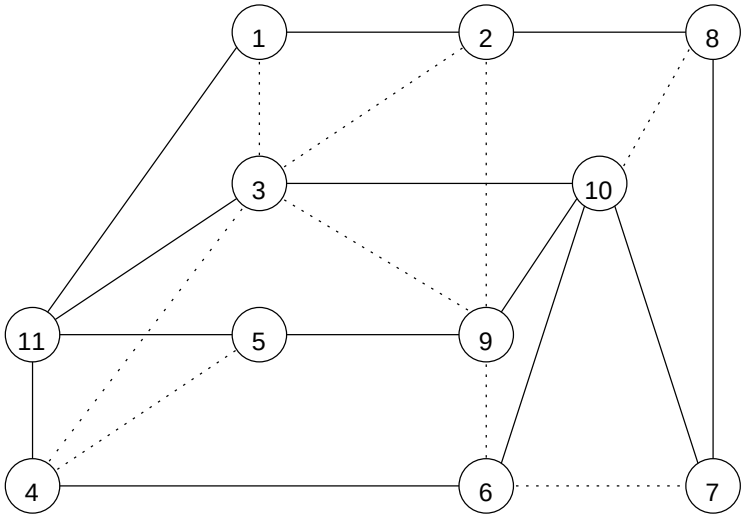

Figure 12.2: Disjunctive paths

12.1.3 Results

The solution algorithm calculates a maximum throughput of 4, which means there are 4 disjunctive paths between nodes 10 and 11 and hence, the two nodes may continue to communicate in spite of the destruction of three intermediate sites. The specifications are satisfied. The four disjunctive paths between 10 and 11 are represented in Figure 12.2.

12.2 Dimensioning of a mobile phone network

Figure 12.3 represents the typical architecture of a mobile phone network. Every elementary geographical zone or **cell** is served by a transmitter-receiver called a **relay**. The calls originating from a mobile phone first pass through these relays. Every relay is connected by cable or electro-magnetic waves to a transit node (**hub**). One of the hubs controls the network, this is the MTSO (Mobile Telephone Switching Office). A very reliable ring of fiber optic cable connects the hubs and the MTSO with high capacity links. It is capable of re-establishing itself in the case of a breakdown (self-healing ring) and there is no need to replicate it.

At the present state of technology, there are no dynamic connections between the relays and the MTSO. The connections are fixed during the design phase, so it is necessary to choose the nodes of the ring that a relay should be connected to. The number of links between a cell *c* and the ring is called the diversity of the cell *c*, denoted by $CNCT_c$. A diversity larger than 1 is recommended for making the system more reliable.

The traffic in this kind of system is entirely digitized, expressed in values equivalent to **bidirectional circuits** at 64kbps (kilobit per second). This capacity corresponds to the number of simultaneous calls during peak periods. The ring has edges of known capacity *CAP*. The traffic $TRAF_c$ from a cell *c* is divided into equal parts ($TRAF_c$ / $CNCT_c$) among the connections to the ring. This traffic is transmitted via the ring

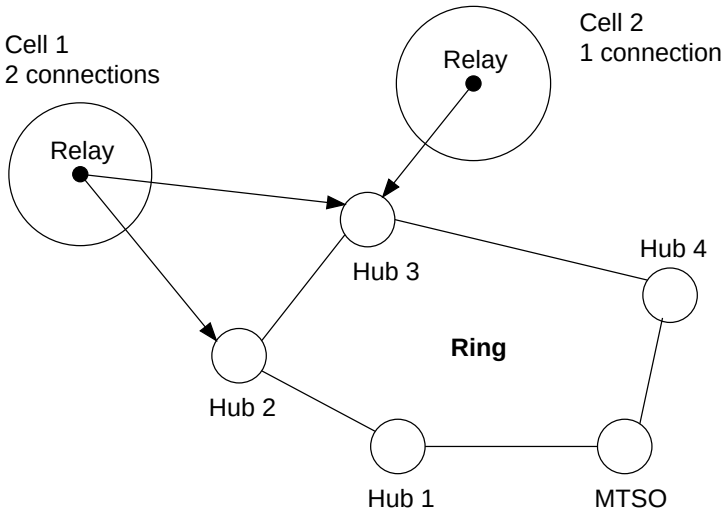

Figure 12.3: Structure of a mobile phone network

to the MTSO, that then routes it to another cell or to a hub that serves as the interface to the fixed-line telephone network. A relay may be connected directly to the MTSO that also has the functions of an ordinary hub.

We consider a network of 10 cells and a ring of 5 nodes with a capacity of $CAP = 48$ circuits. The MTSO is at node 5. The following table indicates the traffic, the required number of connections and the cost per connection in thousand $ per cell. For example, cell 1 is connectable with node 1 for a cost of $15,000. Its diversity is 2, which means it must be connected to at least two nodes of the ring. Its traffic capacity is of 22 simultaneous circuits. The objective is to define the connections of the cells to the ring that minimize the connection costs whilst remaining within the capacity limits and satisfying the constraints on the number of connections.

Table 12.1: Connection costs, traffic and number of connections per cell

Cell	1	2	3	4	5	6	7	8	9	10
Hub 1	15	9	12	17	8	7	19	20	21	25
Hub 2	8	11	6	5	22	25	25	9	22	24
Hub 3	7	8	7	9	21	15	21	15	14	13
Hub 4	11	5	15	18	19	9	20	18	16	4
Hub 5 (MTSO)	10	14	15	24	6	17	22	25	20	11
Traffic	22	12	20	12	15	25	15	14	8	22
Connections	2	2	2	2	3	1	3	2	2	2

12.2.1 Model formulation

We write $CELLS$ for the set of cells to be connected, $NODES = HUBS \cup \{MTSO\}$ the set of nodes, composed of the set of simple hubs and an MTSO, and $COST_{cn}$ the cost of connecting cell c to node n. The connections can be defined through binary variables $connect_{cn}$, that are 1 if and only if cell c is connected to node n (12.2.4). The objective function (12.2.1) measures the total connection cost. The constraints (12.2.2) ensure that every cell is connected to the required number of nodes.

$$\text{minimize} \sum_{c \in CELLS} \sum_{n \in NODES} COST_{cn} \cdot connect_{cn} \tag{12.2.1}$$

$$\forall c \in CELLS : \sum_{n \in NODES} connect_{cn} = CNCT_c \tag{12.2.2}$$

$$\sum_{c \in CELLS} \sum_{n \in HUBS} \frac{TRAF_c}{CNCT_c} \cdot connect_{cn} \leq 2 \cdot CAP \tag{12.2.3}$$

$$\forall c \in CELLS, n \in NODES : connect_{cn} \in \{0, 1\} \tag{12.2.4}$$

Constraint (12.2.3) is a necessary condition for keeping within the capacity limits of the ring. It is based on the fact that all demands of any origin are routed through the MTSO, following the ring in one sense

or the other. Since every edge of the ring has a capacity *CAP*, the total traffic on the ring may not exceed $2 \cdot CAP$. Note that the traffic of a cell directly connected to the MTSO does not enter the ring.

For the problem to have any solution the ring must have a certain minimum capacity to route all the traffic. The least possible traffic in the ring is obtained if every cell c has a direct connection with the MTSO among its $CNCT_c$ required connections with the ring. The ring must be able to handle the remaining traffic, which results in the minimum capacity constraint (12.2.5): for example, a cell with diversity 2 routes at least $1 - \frac{1}{2} = \frac{1}{2}$ of its traffic through the ring; the traffic of a cell with diversity 1 in the best case (*i.e.* if it is connected to the MTSO) does not enter the ring at all. This relation does not contain any variable, it is therefore not included in the mathematical model. It is however necessary to check whether the given data fulfill this condition, which is the case here.

$$\sum_{c \in CELLS} TRAF_c \cdot \left(1 - \frac{1}{CNCT_c}\right) \leq 2 \cdot CAP \tag{12.2.5}$$

The non-specialist in telecommunications might raise the question of what has happened to the return traffic. The traffic from a cell c is expressed in bidirectional circuits. If someone in cell c calls someone in cell d, then the call traces its path from c to the MTSO, reserving a path for the answer. The MTSO then transmits the call to cell d, still reserving a return path. When the called person in d picks up the phone, an access path in both directions is established, consuming one circuit from the capacity of c and one circuit from the capacity of d.

12.2.2 Implementation

The mathematical model is easily translated into a Mosel program. The cost data array from table 12.1 is given in transposed form in the data file to match the format of the array $COST_{cn}$.

Before defining the model we check whether the ring has a sufficiently large capacity to satisfy all traffic demands (inequality (12.2.5)). If this condition does not hold then the program is stopped using the function `exit`.

```
model "G-2 Mobile network dimensioning"
 uses "mmxprs"

 declarations
  HUBS = 1..4                          ! Set of hubs
  MTSO = 5                             ! Node number of MTSO
  NODES = HUBS+{MTSO}                  ! Set of nodes (simple hubs + MTSO)
  CELLS = 1..10                        ! Cells to connect

  CAP: integer                         ! Capacity of ring segments
  COST: array(CELLS,NODES) of integer  ! Connection cost
  TRAF: array(CELLS) of integer        ! Traffic from every cell
  CNCT: array(CELLS) of integer        ! Connections of a cell to the ring

  connect: array(CELLS,NODES) of mpvar ! 1 if cell connected to node,
                                       ! 0 otherwise
 end-declarations

 initializations from 'g2dimens.dat'
  CAP COST TRAF CNCT
 end-initializations

! Check ring capacity
 if not (sum(c in CELLS) TRAF(c)*(1-1/CNCT(c)) <= 2*CAP) then
  writeln("Ring capacity not sufficient")
  exit(0)
 end-if

! Objective: total cost
 TotCost:= sum(c in CELLS, n in NODES) COST(c,n)*connect(c,n)

! Number of connections per cell
 forall(c in CELLS) sum(n in NODES) connect(c,n) = CNCT(c)

! Ring capacity
 sum(c in CELLS, n in HUBS) (TRAF(c)/CNCT(c))*connect(c,n) <= 2*CAP

 forall(c in CELLS, n in NODES) connect(c,n) is_binary
```

```
! Solve the problem
 minimize(TotCost)

end-model
```

12.2.3 Results

The optimal MIP solution has a total cost of $249,000. The traffic on the ring is 94, compared to a capacity of 96. The following table lists the details of the connections to be established between cells and nodes of the ring.

Table 12.2: Connection costs, traffic and number of connections per cell

Cell	1	2	3	4	5	6	7	8	9	10
Host nodes	3,5	3,4	2,5	2,3	1,4,5	5	1,4,5	2,3	3,5	4,5

12.3 Routing telephone calls

A private telephone company exploits a network, represented in Figure 12.4, between five cities: Paris, Nantes, Nice, Troyes, and Valenciennes.

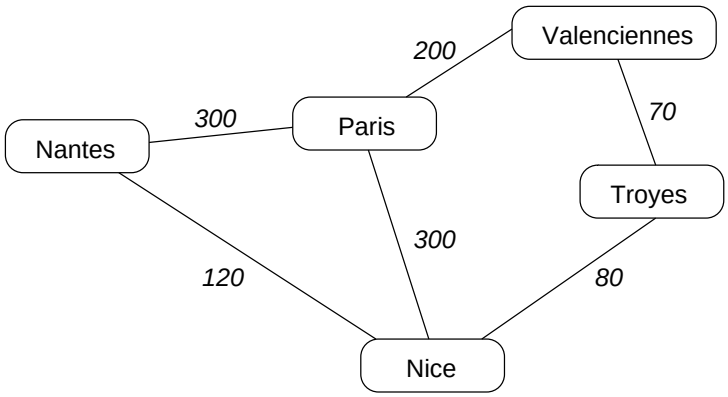

Figure 12.4: Structure of the network of the company

The number beside each edge (connection) is the capacity of the link in terms of circuits. This issue is worth some further explanation. Suppose that a person A at Nantes calls a person B at Nice. The company needs to find a path formed by non-saturated edges from Nantes to Nice to route the binary flow corresponding to the digitized voice of A. But the conversation is only possible if a flow from Nice to Nantes is established so that B may answer A. In digital telephone networks, the binary flows all have the same throughput standard (often 64 kbps). The associated capacity is called a **channel**. The return channel uses the same edges as the first channel, but in the opposite direction. This linked pair of channels necessary for a telephone conversation is a **circuit**.

The routing of the return channel through the same edges is due to the fact that the call could fail if one waited until the callee picked up the phone before searching for a non-saturated return path. This is why, at the moment of the call, the routing system constructs the path edge by edge and immediately reserves the edges of the return channel. As a consequence, as the capacity of an edge is consumed in increments of a bidirectional circuit, we do not consider any directed flows. For example, there may be 10 persons calling from Nantes to Nice and 20 from Nice to Nantes, or the opposite: in both cases, 30 circuits are used.

At a given moment, the company is facing demands for circuits given in the following table. Is it possible to satisfy the demands entirely? If this is not possible, try to transmit as much as possible. In every case, indicate the corresponding routing, that is, the access paths used.

12.3.1 Model formulation

In the maximum flow problem in Section 15.1 (water supply management) the task is to maximize the total throughput of a single product flow that traverses a limited capacity network between an origin

Table 12.3: Demand of circuits

Names of the cities	Circuits
Nantes-Nice	100
Nantes-Troyes	80
Nantes-Valenciennes	75
Nice-Valenciennes	100
Paris-Troyes	70

(source) and a destination (sink). Here, we are dealing with a generalization called the **multi-commodity network flow** (MCNF) problem: several flows that may not be mixed and that correspond to different products use the same network, every flow having its own source and sink. The objective is still to maximize the total throughput. Here, a 'product' corresponds to the totality of calls exchanged between two given cities. The calls between two different pairs of cities may not be mixed: calls between Nantes and Nice must not end in Troyes!

The MCNF problem is a difficult combinatorial problem. In the classical maximum flow problem, Linear Programming uses flow variables $flow_{nm}$ for every arc (n, m). The simplex algorithm automatically finds integer-valued optimal flows if the arc capacities are integers. It is possible to generalize this model to solve the MCNF problem in a directed graph with variables $flow_{knm}$ for the flow of product k on the arc (n, m) but the values at the end of the LP are not usually integers. Optimally rounding the fractional flows is very difficult because there may be several flows that are routed through the same saturated arc. Furthermore, the model with three-index variables requires certain tricks for dealing with an undirected network as is the case in the present example. We are therefore going to use a simpler formulation called **arc-paths**.

This formulation is based on different **elementary paths** (without intermediate nodes in common) that may be used between pairs of cities that are communicating. Instead of using variables $flow_{knm}$ for every pair of cities k and every arc (n, m), this formulation uses a single variable $flow_p$ for the flow **all along** a path p, where this path corresponds to a known pair of cities. The flow conservation laws per node are therefore implicitly fulfilled. This formulation cannot be employed in dense networks with an enormous number of possible paths. But for telecommunication networks that have a skeleton-like structure it is better than modeling with three-index flow variables.

For the data, we use *ARCS* for the set of (undirected) arcs, *CALLS* for the set of city pairs between which there are demands for calls, and *PATHS* the set of paths. Let CAP_a denote the capacity of an arc and DEM_c the demand for a city pair c. We also need to know which pair of cities is at the ends of every path, given by the call index $CINDEX_p$. The arc-paths formulation is then remarkably easy. It is called 'arc-paths' because it works with arcs and paths, but not with nodes.

$$\text{maximize} \sum_{p \in PATHS} flow_p \tag{12.3.1}$$

$$\forall a \in ARCS : \sum_{\substack{p \in PATHS \\ a \in p}} flow_p \leq CAP_a \tag{12.3.2}$$

$$\forall c \in CALLS : \sum_{\substack{p \in PATHS \\ CINDEX_p = c}} flow_c \leq DEM_c \tag{12.3.3}$$

$$\forall p \in PATHS : flow_p \in \mathbb{N} \tag{12.3.4}$$

The objective function is to maximize (12.3.1), the sum of the flows on all paths that may be used. The constraints (12.3.2) consider every arc a: the sum of flows on the paths passing through a must not exceed the capacity of a. The constraints (12.3.3) concern every pair of cities c: the sum of flows exchanged between them over the various paths must not exceed the demand for circuits. The integrality constraints (12.3.4) are necessary because the MCNF problem is not automatically integer at its linear optimum.

12.3.2 Implementation

The translation of the mathematical model for Mosel is given in the following program. This is a typical case where after the formulation of the mathematical model some work still is left to do for the formulation of the model with Mosel since no direct representation of the paths is immediately available. The pairs of cities are named as represented in Table 12.3. Similarly, the arcs are named with the cities they connect: Nantes-Paris, Nantes-Nice, Paris-Nice, Paris-Valenciennes, Troyes-Nice and Troyes-Valenciennes.

The paths are coded as a two-dimensional array $ROUTE_{pa}$, every line of which corresponds to a path and

contains the list of edges it uses. In the worst case, a path may contain all the arcs since the paths are elementary and do not visit any city twice. The size of the second dimension of the array *ROUTE* is therefore given by the number *NARC* of arcs in the problem. The following table lists the complete set of paths and the corresponding city pairs at the ends of every path (array $CINDEX_p$). With this representation, the model can now be implemented easily.

Table 12.4: Definition of paths

Path	City pair	List of arcs
1	Nantes-Nice	Nantes-Nice
2	Nantes-Nice	Nantes-Paris, Paris-Nice
3	Nantes-Nice	Nantes-Paris, Paris-Valenciennes, Valenciennes-Troyes, Troyes-Nice
4	Nantes-Troyes	Nantes-Paris, Paris-Valenciennes, Valenciennes-Troyes
5	Nantes-Troyes	Nantes-Paris, Paris-Nice, Nice-Troyes
6	Nantes-Troyes	Nantes-Nice, Nice-Troyes
7	Nantes-Troyes	Nantes-Nice, Nice-Paris, Paris-Valenciennes, Valenciennes-Troyes
8	Nantes-Valenciennes	Nantes-Paris, Paris-Valenciennes
9	Nantes-Valenciennes	Nantes-Nice, Nice-Paris, Paris-Valenciennes
10	Nantes-Valenciennes	Nantes-Paris, Paris-Nice, Nice-Troyes, Troyes-Valenciennes
11	Nantes-Valenciennes	Nantes-Nice, Nice-Troyes, Troyes-Valenciennes
12	Nice-Valenciennes	Nice-Nantes, Nantes-Paris, Paris-Valenciennes
13	Nice-Valenciennes	Nice-Paris, Paris-Valenciennes
14	Nice-Valenciennes	Nice-Troyes, Troyes-Valenciennes
15	Paris-Troyes	Paris-Valenciennes, Valenciennes-Troyes
16	Paris-Troyes	Paris-Nantes, Nantes-Nice, Nice-Troyes
17	Paris-Troyes	Paris-Nice, Nice-Troyes

Making the path inventory is quite tedious, but the collection of data is usually a major problem in Mathematical Programming. In industrial applications, the data files must be constructed automatically by some special purpose software (in the examples of Sections 11.2 or 12.5 we use Mosel for doing some data preprocessing) or be extracted from databases or spreadsheets. The full distribution of Mosel includes modules for accessing databases or spreadsheets directly.

```
model "G-3 Routing telephone calls"
 uses "mmxprs"

 declarations
  CALLS: set of string                 ! Set of demands
  ARCS: set of string                  ! Set of arcs
  PATHS: range                         ! Set of paths (routes) for demands

  CAP: array(ARCS) of integer          ! Capacity of arcs
  DEM: array(CALLS) of integer         ! Demands between pairs of cities
  CINDEX: array(PATHS) of string       ! Call (demand) index per path index
 end-declarations

 initializations from 'g3routing.dat'
  CAP DEM INDEX
 end-initializations

 finalize(CALLS); finalize(ARCS); finalize(PATHS)
 NARC:=getsize(ARCS)

 declarations
  ROUTE: array(PATHS,1..NARC) of string ! List of arcs composing the routes
  flow: array(PATHS) of mpvar          ! Flow on paths
 end-declarations

 initializations from 'g3routing.dat'
  ROUTE
 end-initializations

! Objective: total flow on the arcs
 TotFlow:= sum(p in PATHS) flow(p)

! Flow within demand limits
 forall(d in CALLS) sum(p in PATHS | CINDEX(p) = d) flow(p) <= DEM(d)
```

```
! Arc capacities
 forall(a in ARCS)
  sum(p in PATHS, b in 1..NARC | ROUTE(p,b)=a) flow(p) <= CAP(a)

 forall(p in PATHS) flow(p) is_integer

! Solve the problem
 maximize(TotFlow)

end-model
```

12.3.3 Results

The LP solution to this problem has integer values. 380 out of the required 425 calls are routed, that is all but 45 Nantes-Troyes calls. The following table lists the corresponding routing of telephone calls (there are several equivalent solutions). On the connections Nantes-Paris, Nantes-Nice, Paris-Nice, and Troyes-Valenciennes there is unused capacity; the other arcs are saturated.

Table 12.5: Optimal routing of telephone calls

City pair	Demand	Routed	Path
Nantes-Nice	100	100	Nantes-Paris, Paris-Nice
Nantes-Troyes	80	35	Nantes-Nice, Nice-Paris, Paris-Valenciennes, Valenciennes-Troyes
Nantes-Valenciennes	75	75	Nantes-Paris, Paris-Valenciennes
Nice-Valenciennes	100	20	Nice-Paris, Paris-Valenciennes
		80	Nice-Troyes, Troyes-Valenciennes
Paris-Troyes	70	70	Paris-Valenciennes, Valenciennes-Troyes

12.4 Construction of a cabled network

A university wishes to connect six terminals located in different buildings of its campus. The distances, in meters, between the different terminals are given in Table 12.6.

Table 12.6: Distances between the different terminals (in meters)

	Terminal 1	Terminal 2	Terminal 3	Terminal 4	Terminal 5	Terminal 6
Terminal 1	0	120	92	265	149	194
Terminal 2	120	0	141	170	93	164
Terminal 3	92	141	0	218	103	116
Terminal 4	265	170	218	0	110	126
Terminal 5	149	93	103	110	0	72
Terminal 6	194	164	116	126	72	0

These terminals are to be connected via underground cables. We suppose the cost of connecting two terminals is proportional to the distance between them. Determine the connections to install to minimize the total cost.

12.4.1 Model formulation

We define the undirected labeled graph G = (*TERMINALS, DIST, CONNECTIONS*) where the set of nodes *TERMINALS* corresponds to the set of terminals and the set of edges *CONNECTIONS* contains the possible connections [*s, t*] between these terminals which are labeled with the distance $DIST_{st}$ that separates them. We define binary variables $connect_{st}$ that are 1 if and only if terminals s and t are directly connected. Since the connections are undirected, it is sufficient to define the variables for $s < t$. The objective is to connect all the terminals at the least cost. The objective function is therefore given by (12.4.1).

$$\text{minimize} \sum_{s \in TERMINALS} \sum_{\substack{t \in TERMINALS \\ s<t}} DIST_{st} \cdot connect_{st} \tag{12.4.1}$$

To connect *NTERM* terminals, the most economical network is a tree of *NTERM* − 1 connections. A tree connecting *NTERM* nodes is a connected graph without any cycles or, equivalently a connected graph

with $NTERM - 1$ connections. The constraint (12.4.2) imposes these $NTERM - 1$ connections.

$$\sum_{s \in TERMINALS} \sum_{\substack{t \in TERMINALS \\ s<t}} connect_{st} = NTERM - 1 \tag{12.4.2}$$

Furthermore, every terminal must be connected to at least one other terminal in the tree. A first idea is to add the constraints (12.4.3).

$$\forall s \in TERMINALS : \sum_{\substack{t \in TERMINALS \\ s<t}} connect_{st} \geq 1 \tag{12.4.3}$$

However, these constraints are not sufficient. The constraints (12.4.2) and (12.4.3) may lead to an infeasible solution as shown in Figure 12.5 in which a cycle is created.

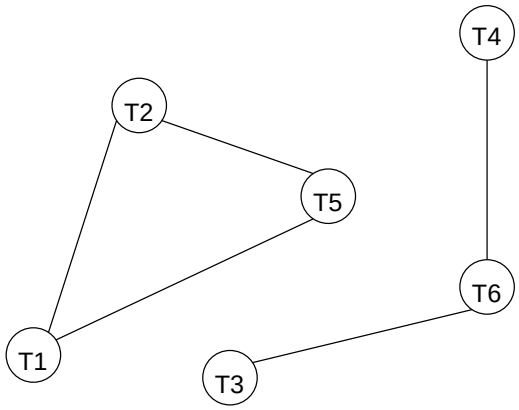

Figure 12.5: An infeasible unconnected solution

To avoid this type of solution, we need to impose the constraint (12.4.4) for any subset S of the set of terminals. The translation of this constraint into a program is unfortunately a difficult task: on one hand there is no possibility of enumerating all subsets, and on the other hand the number of constraints of this type is enormous (2^{NTERM}, that is, about one million for twenty nodes).

$$\forall S \subseteq TERMINALS : \sum_{s \in S} \sum_{t \in S,\ s<t} connect_{st} \leq |S| - 1 \tag{12.4.4}$$

There follows therefore, a different way of preventing cycles. Consider a tree with its edges directed departing from a root r, that is, a node that is connected to a single other node, like T_1 in Figure 12.6 (any root will do). We may now assign every node t a level value $level_t$ that may be interpreted as the length (in terms of number of edges) of the path connecting t with r (in a tree this path exists and is unique). For example, we have $level_r = 0$ and $level_t = 1$ for any node t directly connected to r. The anti-cycle constraints (12.4.5) are based on positive level value variables. For this formulation, the connections must be directed: we have either $connect_{st} = 1$ or $connect_{ts} = 1$ for two directly connected nodes s and t. This has no impact on the solution.

$$\forall s, t \in TERMINALS, s \neq t : level_t \geq level_s + 1 - NTERM + NTERM \cdot connect_{st} \tag{12.4.5}$$

To understand these constraints, let us suppose that a solution of the mathematical model contains a cycle, for instance the cycle $1 \rightarrow 2 \rightarrow 5 \rightarrow 1$ of Figure 12.5, oriented in this sense. The constraints (12.4.5) for which $connect_{st} = 1$ in this cycle result in:

$$level_2 \geq level_1 + 1 \tag{12.4.6}$$
$$level_5 \geq level_2 + 1 \tag{12.4.7}$$
$$level_1 \geq level_5 + 1 \tag{12.4.8}$$

By summing up term by term, we obtain a contradiction: $0 \geq 3$! A solution containing a cycle therefore violates the constraints (12.4.5). On the contrary, if we have a tree, values of $level_t$ exist that satisfy the constraints, for instance the level numbers obtained by traversing the tree starting from an arbitrary root r. If s is directly before t on the path from r to t, we have $level_t = level_s + 1$ and $connect_{st} = 1$. The

constraint (12.4.5) for s and t is satisfied since it reduces to $level_t = level_s + 1$. If s is not directly connected to t, we have $connect_{st} = 0$ and the constraint (12.4.5) reduces to $level_t - level_s \geq 1 - NTERM$, an inequality that is trivially satisfied since the level numbers range between 0 and $NTERM - 1$.

The constraints (12.4.5) only detect cycles with connections directed around the cycle. For instance, they do not exclude the cycle $1 \rightarrow 2 \leftarrow 5 \rightarrow 1$. Such a cycle cannot occur when we add the following constraints that direct all connections to the (arbitrarily chosen) root node.

$$\forall s = 2, \ldots, NTERM: \sum_{\substack{t \in TERMINALS \\ s \neq t}} connect_{st} = 1 \tag{12.4.9}$$

In the formulation of the constraints (12.4.9) we have chosen node 1 as the root node. Every node must be connected to at least one other node. Since a tree does not contain cycles there must be a single path from every node in the tree to the root node. That means, every node s other than the root node has exactly one outgoing connection. In other words, for every $s \neq 1$ exactly one variable $connect_{st}$ must be at 1.

We thus obtain the following mathematical model.

$$\text{minimize} \sum_{s \in TERMINALS} \sum_{t \in TERMINALS} DIST_{st} \cdot connect_{st} \tag{12.4.10}$$

$$\sum_{s \in TERMINALS} \sum_{t \in TERMINALS} connect_{st} \leq NTERM - 1 \tag{12.4.11}$$

$$\forall s, t \in TERMINALS, s \neq t: level_t \geq level_s + 1 - NTERM + NTERM \cdot connect_{st} \tag{12.4.12}$$

$$\forall s = 2, \ldots, NTERM: \sum_{\substack{t \in TERMINALS \\ s \neq t}} connect_{st} = 1 \tag{12.4.13}$$

$$\forall s, t \in TERMINALS: connect_{st} \in \{0, 1\} \tag{12.4.14}$$

$$\forall t \in TERMINALS: level_t \geq 0 \tag{12.4.15}$$

12.4.2 Implementation

The translation of the mathematical model into a Mosel program is direct.

```
model "G-4 Cable connections between terminals"
 uses "mmxprs"

 declarations
  NTERM = 6
  TERMINALS = 1..NTERM                  ! Set of terminals to connect

  DIST: array(TERMINALS,TERMINALS) of integer  ! Distance between terminals

  connect: array(TERMINALS,TERMINALS) of mpvar ! 1 if direct connection
                                        ! between terminals, 0 otherwise
  level: array(TERMINALS) of mpvar      ! level value of nodes
 end-declarations

 initializations from 'g4cable.dat'
  DIST
 end-initializations

! Objective: length of cable used
 Length:= sum(s,t in TERMINALS) DIST(s,t)*connect(s,t)

! Number of connections
 sum(s,t in TERMINALS | s<>t) connect(s,t) = NTERM - 1

! Avoid subcycle
 forall(s,t in TERMINALS | s<>t)
  level(t) >= level(s) + 1 - NTERM + NTERM*connect(s,t)

! Direct all connections towards the root (node 1)
 forall(s in 2..NTERM) sum(t in TERMINALS | s<>t) connect(s,t) = 1

 forall(s,t in TERMINALS | s<>t) connect(s,t) is_binary
```

```
! Solve the problem
 minimize(Length)

end-model
```

12.4.3 Results

The optimal solution connects the terminals as shown in Figure 12.6. The total length of cable required is 470 meters.

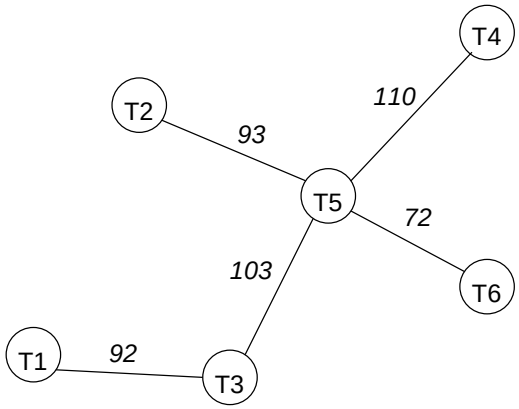

Figure 12.6: Optimal network of connections

12.5 Scheduling of telecommunications via satellite

A digital telecommunications system via satellite consists of a satellite and a set of stations on earth which serve as interfaces with the terrestrial network. With SS-TDMA (Satellite-Switch, Time Division Multiple Access) access technology, the satellite divides its time among the stations. Consider for example the transmissions from four transmitter stations in the US to four receiver stations in Europe. The following table gives a possible 4×4 traffic matrix. $TRAF_{tr}$ is the quantity of data transmitted from station t to station r. It is expressed in seconds of transmission duration, because all lines have the same constant transmission rate.

Table 12.7: Matrix *TRAF* with its lower bound *LB*

TRAF	1	2	3	4	row_t
1	0	7	11	15	33
2	15	8	13	9	45
3	17	12	6	10	45
4	6	13	15	4	38
col_r	38	40	45	38	$LB = 45$

The satellite has a switch that allows any permutation between the four transmitters and the four receivers. Figure 12.8 gives an example of a permutation connecting the transmitters 1 to 4 to the receivers 3, 4, 1, and 2 respectively. These connections allow routing a part of the traffic matrix, called a **mode**. A part of a matrix entry transmitted during a mode is a **packet** of data. A mode is thus a 4×4 matrix M with at most one non-zero packet per row and column and such that $M_{tr} \leq TRAF_{tr}$ for all t, r. To every mode corresponds a slice of a schedule as shown in Figure 12.8.

Table 12.8: Example of a mode and associated schedule

	1	2	3	4		Stations	Packets
1	0	0	11	0		1 to 3	11
2	0	0	0	9		2 to 4	9
3	15	0	0	0		3 to 1	15
4	0	13	0	0		4 to 2	13
col_r	38	40	45	38	$LB = 45$		

A valid schedule of transmissions defines a sequence of permutations for the on-board switch that routes all the traffic of the matrix *TRAF*. In other words, this boils down to decomposing *TRAF* into a sum of mode matrices. An element of *TRAF* may be fragmented, like $TRAF_{31}$ that is only partially transmitted in the mode represented in Figure 12.8. A fragmented element will appear in several packets and several modes of the final decomposition. The **duration** of a mode is the length of its longest packet. The objective is to find a schedule with minimal total duration.

12.5.1 Model formulation

Let $TRANSM = \{1, \ldots, NT\}$ be the set of transmitter stations and $RECV = \{1, \ldots, NT\}$ the set of receiver stations. We shall first formulate a model for this problem when fragmentations are not allowed. With a square matrix as in the present case, *NT* modes are sufficient for decomposing *TRAF*. We define binary variables $flow_{trm}$ that are 1 if the element $TRAF_{tr}$ is transmitted in mode number $m \in MODES = \{1, \ldots, NT\}$ (12.5.7). The constraints (12.5.2) indicate that every $TRAF_{tr}$ must be transmitted in a single mode (no preemption). The constraints (12.5.3) and (12.5.4) ensure that every mode is valid, with a single entry per row and per column.

The continuous variables dur_m represent the duration of every mode, that is, the duration of its largest element. The constraints (12.5.5) bound every $TRAF_{tr}$ transmitted in mode m by dur_m. The objective function (12.5.1) denotes the total duration of the decomposition to modes, that is, the sum of the durations of the modes. The mathematical model we obtain is a **bottleneck assignment problem** with three indices. It requires $NT^3 + NT$ variables and $NT^3 + 3NT^2$ constraints.

$$\text{minimize} \sum_{m \in MODES} dur_m \tag{12.5.1}$$

$$\forall t \in TRANSM, r \in RECV : \sum_{m \in MODES} flow_{trm} = 1 \tag{12.5.2}$$

$$\forall t \in TRANSM, m \in MODES : \sum_{r \in RECV} flow_{trm} = 1 \tag{12.5.3}$$

$$\forall r \in RECV, m \in MODES : \sum_{t \in TRANSM} flow_{trm} = 1 \tag{12.5.4}$$

$$\forall t \in TRANSM, r \in RECV, m \in MODES : TRAF_{tr} \cdot flow_{trm} \leq dur_m \tag{12.5.5}$$

$$\forall m \in MODES : dur_m \geq 0 \tag{12.5.6}$$

$$\forall t \in TRANSM, r \in RECV, m \in MODES : flow_{trm} \in \{0, 1\} \tag{12.5.7}$$

The problem described initially in fact allows preemptions. A formulation in the form of a single mathematical model is still possible in this case, but with an even larger number of variables, since theory shows that of the order of NT^2 modes may be required for decomposing *TRAF* with a minimal total duration. This is why we are going to use an iterative algorithm due to Gonzalez and Sahni [GS76]. Every iteration requires us to solve an LP to extract a mode from *TRAF*.

Given that the on-board switch connects a transmitter with a receiver station, the elements from the same row or column of *TRAF* cannot be transmitted in parallel. The sum row_t of the elements of a row t is therefore a lower bound on the duration of any decomposition. The same remark holds for the sum col_r of every column r of *TRAF*. The maximum of these sums defines an even better lower bound *LB*. The table 12.7 displays these sums: we find $LB = 45$.

The algorithm is to find a schedule with duration *LB*. This duration is optimal since it is not possible to find anything shorter than the lower bound *LB*. For the algorithm to work correctly, the traffic matrix *TRAF* must be converted into a matrix for which the sum of every row and every column is *LB*. To obtain this so-called **quasi bistochastic** (QBS) matrix *TQBS*, we need to add fictitious traffic to the matrix elements so that the row and columns add up to *LB*. This may be done systematically with the following algorithm.

- Calculate the row and column sums row_t and col_r of matrix *TRAF*
- Calculate *LB*, the maximum of all row and column sums row_t and col_r
- $\forall t \in TRANS, r \in RECV$:
  - Calculate q, the minimum of $LB - row_t$ and $LB - col_r$
  - Set $TQBS_{tr} = TRAF_{tr} + q$
  - Add q to row_t and to col_r

Table 12.9: Matrix *TQBS*

TQBS	1	2	3	4
1	7	12	11	15
2	15	8	13	9
3	17	12	6	10
4	6	13	15	11

The following table contains the matrix *TQBS* for our example. The elements of all rows and columns sum to 45.

The algorithm of Gonzalez and Sahni has the following general structure:

- Calculate *LB*, the maximum of the row and column sums of *TRAF*
- Convert *TRAF* into a QBS matrix *TQBS* by adding fictitious traffic
- While *TQBS* is non-zero do
  - Calculate a mode *m* with *NT* non-zero elements using Mathematical Programming
  - Calculate *pmin*, the minimum packet length among the elements of *m*
  - Create a slice of the schedule with the packets of *m* cut at *pmin*
  - Deduce *pmin* from every element of *TQBS* contained in mode *m*

  end-do
- Remove the fictitious traffic from the modes that have been obtained

This algorithm **converges** because at every extraction of a mode at least one element of *TQBS* becomes 0: the one that sets the value of *pmin*. The algorithm is optimal because after the subtraction of *pmin*, *TQBS* remains a QBS matrix with the sums of rows and columns equal to $LB - pmin$. The decomposition creates a slice of the schedule of duration *pmin* and leaves a matrix with the bound $LB - pmin$. Continuing this way, we obtain a total duration equal to *LB*.

Any mode may be extracted at every iteration, but to have the least number of modes in the decomposition it is desirable to construct modes that are well filled with traffic. A possibility is to calculate **maximin** modes in which the size of the smallest element is maximized. This may be done using Mathematical Programming. The following model formulates a **maximin assignment problem** analogous to the one in Chapter 14 for the assignment of personnel to machines.

$$\text{maximize } pmin \tag{12.5.8}$$

$$\forall t \in TRANSM: \sum_{\substack{r \in RECV \\ TQBS_{tr} > 0}} flow_{tr} = 1 \tag{12.5.9}$$

$$\forall r \in RECV: \sum_{\substack{r \in RECV \\ TQBS_{tr} > 0}} flow_{tr} = 1 \tag{12.5.10}$$

$$\forall t \in TRANSM: \sum_{t \in TRANSM} TQBS_{tr} \cdot flow_{tr} \geq pmin \tag{12.5.11}$$

$$\forall t \in TRANSM, r \in RECV, TQBS_{tr} > 0: flow_{tr} \in \{0, 1\} \tag{12.5.12}$$

$$pmin \geq 0 \tag{12.5.13}$$

A binary variable $flow_{tr}$ (12.5.12) has the value 1 if and only if the element $TQBS_{tr}$ is taken in the mode. Right from the beginning the matrix *TQBS* may contain zero-valued elements and in any case, zeros will appear during the iterations of the algorithm of Gonzalez and Sahni. This is why the variables $flow_{tr}$ are only defined for the non-zero $TQBS_{tr}$. The constraints (12.5.9) make it impossible to take more than one element per row, the constraints (12.5.10) play the same role for the columns. A continuous variable *pmin* bounds the chosen entries (12.5.11) from below. This minimum packet length variable is maximized in the objective function (12.5.8).

To apply the algorithm of Gonzalez and Sahni, the preceding mathematical model must be solved for the quasi bistochastic matrix *TQBS*. The $flow_{tr}$ at 1 indicate the elements $TQBS_{tr}$ chosen for the mode. The solution value of *pmin* is then subtracted from the chosen $TQBS_{tr}$ and the MIP problem is solved again with the remaining traffic of *TQBS*. The process stops when *TQBS* is completely 0.

12.5.2 Implementation

The following Mosel program implements the algorithm of Gonzalez and Sahni described in the previous section. After every solution of the MIP problem, the corresponding elements of `TQBS` are updated, and at the next loop the problem is redefined correspondingly. Since the entire MIP problem is redefined in every iteration of the `while` loop, all constraints are named in this problem so that their previous definition may be replaced by the new one. Proceeding in such a way is necessary because the problem definition in Mosel is incremental: constraints may only be removed by explicitly resetting them. Once defined, the variables `flow(t,r)` cannot be removed, but through the tests `TQBS(t,r)>0` in the definition of the constraints only the variables actually required are used in the definition of the current problem.

In this program, we save all the solutions found (elements and durations of the modes) for printing out the complete solution in the form of a diagram at the end of the program. During the solution printout, the scheduled packets are compared with the original traffic matrix *TRAF* so that the additional fictitious traffic in *TQBS* is not shown, but only the original traffic demands.

```
model "G-5 Satellite scheduling"
 uses "mmxprs"

 declarations
  TRANSM = 1..4                          ! Set of transmitters
  RECV = 1..4                            ! Set of receivers

  TRAF: array(TRANSM,RECV) of integer    ! Traffic betw. terrestrial stations
  TQBS: array(TRANSM,RECV) of integer    ! Quasi bistochastic traffic matrix
  row: array(TRANSM) of integer          ! Row sums
  col: array(RECV) of integer            ! Column sums
  LB: integer                            ! Maximum of row and column sums
 end-declarations

 initializations from 'g5satell.dat'
  TRAF
 end-initializations

! Row and column sums
 forall(t in TRANSM) row(t):= sum(r in RECV) TRAF(t,r)
 forall(r in RECV) col(r):= sum(t in TRANSM) TRAF(t,r)
 LB:=maxlist(max(r in RECV) col(r), max(t in TRANSM) row(t))

! Calculate TQBS
 forall(t in TRANSM,r in RECV) do
  q:= minlist(LB-row(t),LB-col(r))
  TQBS(t,r):= TRAF(t,r)+q
  row(t)+=q
  col(r)+=q
 end-do

 declarations
  MODES: range

  flow: array(TRANSM,RECV) of mpvar         ! 1 if transmission from t to r,
                                            ! 0 otherwise
  pmin: mpvar                               ! Minimum exchange
  onerec, minexchg: array(TRANSM) of linctr ! Constraints on transmitters
                                            ! and min exchange
  onerec: array(RECV) of linctr             ! Constraints on receivers

  solflowt: array(TRANSM,MODES) of integer  ! Solutions of every iteration
  solflowr: array(RECV,MODES) of integer    ! Solutions of every iteration
  solpmin: array(MODES) of integer          ! Objective value per iteration
 end-declarations

 forall(t in TRANSM,r in RECV) flow(t,r) is_binary

 ct:= 0
 while(sum(t in TRANSM,r in RECV) TQBS(t,r) > 0) do
  ct+=1

 ! One receiver per transmitter
  forall(t in TRANSM) onerec(t):= sum(r in RECV | TQBS(t,r)>0) flow(t,r) =1
```

```
 ! One transmitter per receiver
  forall(r in RECV) onetrans(r):= sum(t in TRANSM | TQBS(t,r)>0) flow(t,r) =1

 ! Minimum exchange
  forall(t in TRANSM)
   minexchg(t):= sum(r in RECV | TQBS(t,r)>0) TQBS(t,r)*flow(t,r) >= pmin

 ! Solve the problem: maximize the minimum exchange
  maximize(pmin)

 ! Solution printing
  writeln("Round ", ct, " objective: ", getobjval)

 ! Save the solution
  solpmin(ct):= round(getobjval)
  forall(t in TRANSM,r in RECV | TQBS(t,r)>0)
   if(getsol(flow(t,r))>0) then
    solflowt(t,ct):= t
    solflowr(t,ct):= r
   end-if

 ! Update TQBS
  forall(t in TRANSM)
   TQBS(solflowt(t,ct),solflowr(t,ct)) -= solpmin(ct)

 end-do

! Solution printing
 writeln("\nTotal duration: ", sum(m in MODES) solpmin(m))

 write("      ")
 forall(i in 0..ceil(LB/5)) write(strfmt(i*5,5))
 writeln
 forall(t in TRANSM) do
  write("From ", t, " to: ")
  forall(m in MODES)
   forall(i in 1..solpmin(m)) do
    write(if(TRAF(solflowt(t,m),solflowr(t,m))>0, string(solflowr(t,m)),"-"))
    TRAF(solflowt(t,m),solflowr(t,m))-=1
   end-do
  writeln
 end-do

end-model
```

12.5.3 Results

The following tables show the successive contents of *TQBS* during the repeated solution of the MIP. The boxed elements are the elements chosen for the mode. The duration of every mode *pmin* is given above the corresponding table.

The decomposition terminates after eight iterations and the resulting schedule has the expected length *LB* = 45. The upper half of Figure 12.7 shows the schedule obtained with the fictitious traffic of the matrix *TQBS*, the lower half displays the result for the original traffic data without any fictitious traffic (matrix *TRAF*).

12.6 Location of GSM transmitters

A mobile phone operator decides to equip a currently uncovered geographical zone. The management allocates a budget of € 10 million to equip this region. A study shows that only 7 locations are possible for the construction of the transmitters and it is also known that every transmitter only covers a certain number of communities. Figure 12.8 represents a schematic map of the region with the division into communities and the possible locations for transmitters. Every potential site is indicated by a black dot with a number, every community is represented by a polygon. The number in the center of a polygon is the number of the community.

Certain geographical and topological constraints add to the construction cost and reduce the reach of the GSM transmitters. Table 12.11 lists the communities covered and the cost for every site.

Table 12.10: Decomposition of *TQBS* into modes

Mode 1, *pmin* = 13

7	12	11	**15**
15	8	**13**	9
17	12	6	10
6	**13**	15	11

Mode 2, *pmin* = 11

7	12	**11**	2
15	8	0	9
4	**12**	6	10
6	0	15	**11**

Mode 3, *pmin* = 7

7	12	0	2
4	**8**	0	9
4	1	6	**10**
6	0	**15**	0

Mode 4, *pmin* = 6

0	**12**	0	2
4	1	0	**9**
4	1	**6**	3
6	0	8	0

Mode 5, *pmin* = 3

0	**6**	0	2
4	1	0	**3**
4	1	0	3
0	0	**8**	0

Mode 6, *pmin* = 3

0	**3**	0	2
4	1	0	0
1	1	0	**3**
0	0	**5**	0

Mode 7, *pmin* = 1

0	0	0	**2**
1	1	0	0
1	1	0	0
0	0	**2**	0

Mode 8, *pmin* = 1

0	0	0	**1**
0	**1**	0	0
1	0	0	0
0	0	**1**	0

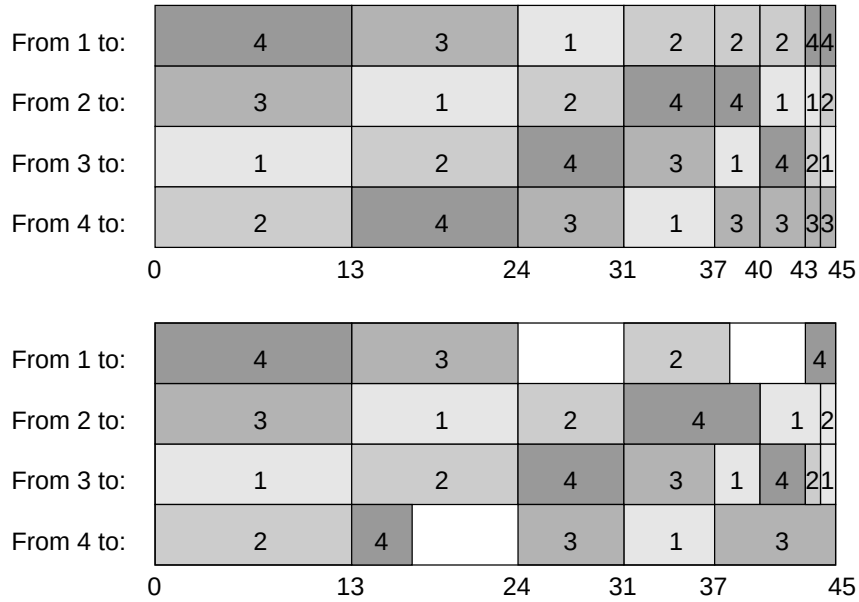

Figure 12.7: Schedule with and without fictitious traffic

Table 12.11: Cost and communities covered for every site

Site	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Cost (in million €)	1.8	1.3	4.0	3.5	3.8	2.6	2.1
Communities covered	1,2,4	2,3,5	4,7,8,10	5,6,8,9	8,9,12	7,10,11,12,15	12,13,14,15

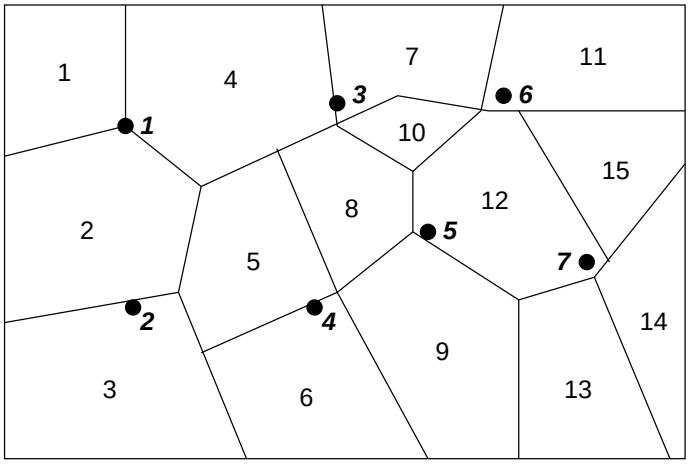

Figure 12.8: Map of the region to cover

For every community the number of inhabitants is known (see Table 12.12). Where should the transmitters be built to cover the largest population with the given budget limit of € 10M?

Table 12.12: Inhabitants of the communities

Community	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Population (in 1000)	2	4	13	6	9	4	8	12	10	11	6	14	9	3	6

12.6.1 Model formulation

Let *COMMS* be the set of communities, *PLACES* the potential sites for constructing transmitters and *BUDGET* the budget allocated by the management. We further write $COST_p$ for the cost of building a transmitter a site p, POP_c the number of inhabitants of the community c and $COVER_{pc}$ a binary constant that indicates whether a transmitter placed at p covers community c ($COVER_{pc} = 1$) or not ($COVER_{pc} = 0$). Two types of binary variables are required: variables $covered_c$ that are 1 if and only if a community c is covered by a transmitter, and variables $build_p$ that are 1 if and only if a transmitter is built at site p.

We formulate the constraints for the coverage of the communities first . We need to translate the equivalence 'community c receives a GSM signal $\Leftrightarrow$ at least one site covering this community is a transmitter is built', or '$covered_c = 1 \Leftrightarrow$ there is at least one p with $COVER_{pc} \cdot build_p = 1$'. This type of equivalence cannot be translated directly into a linear form. Since a community may be covered by more than one transmitter, the constraints (12.6.1) express one direction of the equivalence by specifying that the sum $COVER_{pc} \cdot build_p$ is greater than or equal to $covered_c$.

$$\forall c \in COMMS : \sum_{p \in PLACES} COVER_{pc} \cdot build_p \geq covered_c \tag{12.6.1}$$

The other direction of the equivalence is ensured through the maximization of the population covered (12.6.2). The optimization algorithm will not leave any $covered_c$ at 0 if any transmitter that covers this community is built.

It is also necessary to remain within the budgetary limits. The constraint (12.6.4) bounds the total construction cost for transmitters with the given maximum budget. And finally, the constraints (12.6.5) and (12.6.6) define all variables to be binaries.

$$\text{maximize} \sum_{c \in COMMS} POP_c \cdot covered_c \tag{12.6.2}$$

$$\forall c \in COMMS : \sum_{p \in PLACES} COVER_{pc} \cdot build_p \geq covered_c \tag{12.6.3}$$

$$\sum_{p \in PLACES} COST_p \cdot build_p \leq BUDGET \tag{12.6.4}$$

$$\forall c \in COMMS : covered_c \in \{0, 1\} \tag{12.6.5}$$

$$\forall p \in PLACES : build_p \in \{0, 1\} \tag{12.6.6}$$

12.6.2 Implementation

The translation of the mathematical model into a Mosel program is straightforward. The matrix *COVER* is stored in sparse format in the data file (that is, only the entries with value 1 are given).

```
model "G-6 Transmitter placement"
 uses "mmxprs"

 declarations
  COMMS = 1..15                          ! Set of communities
  PLACES = 1..7                          ! Set of possible transm. locations

  COST: array(PLACES) of real            ! Cost of constructing transmitters
  COVER: array(PLACES,COMMS) of integer  ! Coverage by transmitter locations
  POP: array(COMMS) of integer           ! Number of inhabitants (in 1000)
  BUDGET: integer                        ! Budget limit

  build: array(PLACES) of mpvar          ! 1 if transmitter built, 0 otherwise
  covered: array(COMMS) of mpvar         ! 1 if community covered, 0 otherwise
 end-declarations

 initializations from 'g6transmit.dat'
  COST COVER POP BUDGET
 end-initializations

! Objective: total population covered
 Coverage:= sum(c in COMMS) POP(c)*covered(c)

! Towns covered
 forall(c in COMMS) sum(p in PLACES) COVER(p,c)*build(p) >= covered(c)

! Budget limit
 sum(p in PLACES) COST(p)*build(p) <= BUDGET

 forall(p in PLACES) build(p) is_binary
 forall(c in COMMS) covered(c) is_binary

! Solve the problem
 maximize(Coverage)

end-model
```

12.6.3 Results

When solving this problem, we find that 109,000 persons can be covered by the transmitters. Four transmitters are built, at the sites 2, 4, 6, and 7. Communities 1 and 4 are not covered by these transmitter locations. This operation requires a total budget of € 9.5 million which remains under the limit set by the management of the GSM operator.

In practice, for the choice of transmitter locations other factors also have to be taken into account. The areas covered by neighboring transmitters typically overlap each other: these overlappings should neither be too large (cost inefficient) nor too small (some overlapping is required to pass on calls). Another typical issue is that the transmitters must be placed and dimensioned in such a way that they may be allocated frequencies (out of a set attributed to the operator) that do not cause any interference problems.

12.7 References and further material

The recent work by Sansò et al. [SS98] is dedicated to optimization problems in the design of telecommunication networks. The problem of independent routings in Section 12.1 is a problem of connectivity in a graph. An undirected graph is **k-connected regarding its nodes** if at least k disjoint chains exist between any pair of nodes. Efficient graph algorithms for simple connectivity and 2-connectivity are given in the books by Baase [Baa88] and Prins [Pri94a]. The general case is dealt with for instance by Papadimitriou et al. [PS98] and Ahuja et al. [AMO93].

The problem from Section 12.2 of connecting mobile phone relays to a central network in ring form is dealt with in an article by Dutta et al. [DK99]. The maximization of the throughput of a network in Section 12.3 is a **multi-commodity network flow** (MCNF) problem. The main MCNF problems and solution

methods are given by Gondran [GM90] and Ahuja et al. [AMO93]. For large telecommunications networks, a compatible MCNF algorithm is described by Minoux [Min75], and an algorithm for the minimum cost MCNF problem by Gersht et al. [GS87]. A column generation method for MCNF in telecommunications is described in [BHJS95].

The problem of the cable network in Section 12.4 is a classical minimum weight spanning tree problem. Its model formulation by mathematical LP/MIP is heavy and only valid for small problem ($n \leq 20$ nodes). It is in fact easily solved by graph algorithms like the one by Prim in $O(n^2)$ or the one by Kruskal in $O(m \log n)$ where m denotes the number of edges. These two algorithms are described by Ahuja et al. [AMO93].

The problem of telecommunications via satellite in Section 12.5 is also known as a workshop scheduling problem (**open shop**). The transmitter stations become the machines, the receiver stations the products to produce. $TRAF_{tr}$ indicates the processing duration of a product r on a machine t. A machine may only produce a single product at a time, and a product may only be processed on a single machine at any time. As opposed to the flow shop and job shop problems in Chapter 7, the sequence of processing a product on the different machines is free. The objective is to find the schedule that minimizes the total duration.

The open shop problem is NP-hard if preemptions are not allowed. Heuristics have to be used beyond the limit of ten machines/ten products. See for instance Guéret and Prins [GP98]. The major variants of open shop to be found in telecommunications by satellite are presented in an overview article by Prins [Pri94b]. An example of planning traffic for a longer term, solved by Mathematical Programming, is given by Scott et al. [SSR00]. The bottleneck assignment model with three indices presented at the beginning of Section 12.5.1 for the nonpreemptive case stems from Balas and Landweer [BL83].

The choice of locations for GSM transmitters in Section 12.6 is a covering problem. The closely related problems of covering and partitioning are classics in 0-1 (M)IP, summarized in the section 'References and further material' to Chapter 15. We introduce a partitioning problem (electoral districts) in Chapter 15 and another type of covering problem (cutting sheet metal) in Chapter 9.

The covering problem, although NP-hard, is relatively well solved since the simplex algorithms finds an optimal LP solution where the majority of the binary variables have integer values, which renders the tree search on the remaining fractional variables easier. See for instance Beasley [Bea87]. It is possible to solve problems with several hundred variables to optimality. Very efficient heuristics based on LP/MIP have recently been described for covering problems of very large size [CFT99]. Syslo describes heuristics and exact tree search methods for two types of problems, with source code in Pascal [SDK83].

Chapter 13
Economics and finance

Economists often work with uncertain data and naturally turn towards statistical tools but frequently a simple mathematical model can solve a problem with many variables very quickly to optimality. It is then possible to restart the optimization by varying certain parameters (those that are uncertain). In this case Mathematical Programming becomes an efficient simulation tool. The problems in this chapter concern applications in finance and marketing, aimed at financial decision makers in companies but also for individuals.

Section 13.1 describes a problem of choosing loans to finance expansion projects. Sections 13.2 and 13.5 deal with the optimal use of a given budget, in the first case for organizing an efficient publicity campaign, in the second case to maximize the amount of money a family may spend on leisure. The selection of the most profitable expansion projects over several time periods subject to limited annual budgets is the topic of Section 13.6. The problems in Sections 13.3 and 13.7 concern the optimal composition of investment portfolios according to different criteria (maximum total return and minimum variance). The problem in Section 13.7 differs from all other examples in this book in that it has a **quadratic objective function**. For both portfolio problems it is shown how the models may be used for repeated (simulation) runs by varying certain parameters. Section 13.4 also deals with the choice of investments by looking at a multi-period time horizon and with a different objective: every time period a certain amount of money needs to be available to finance an early retirement scheme.

13.1 Choice of loans

Mr Chic, director of a chain of shops selling clothes, wishes to open three new shops: one in London, one in Munich, and one in Rome. To open a new shop costs respectively €2.5 million, €1 million and €1.7 million. To finance his project, he calls at three different banks.

Table 13.1: Rates offered by the banks for the different projects

	Shop in London	Shop in Munich	Shop in Rome
Bank 1	5%	6.5%	6.1%
Bank 2	5.2%	6.2%	6.2%
Bank 3	5.5%	5.8%	6.5%

Depending on the location of the shops and the evaluation of the related risks, each bank decides to finance at most €3 million over 8 years with different interest rates for the shops (Table 13.1). Determine the amount to borrow from each bank for financing each shop in order to minimize the total expenses of Mr Chic.

13.1.1 Model formulation

Let $borrow_{bs}$ be the amount borrowed from bank b to finance shop s. Let *BANKS* and *SHOPS* respectively be the sets of banks willing to finance and of new shops. Let *DUR* be the number of years over which the repayment of the credits stretches, *VMAX* the maximum amount that every bank is prepared to finance, $RATE_{bs}$ the interest rate offered by bank b for shop s, and $PRICE_s$ the cost of opening shop s. The mathematical model is the following:

$$\text{minimize} \sum_{b \in BANKS} \sum_{s \in SHOPS} borrow_{bs} \cdot \frac{RATE_{bs}}{1 - (1 + RATE_{bs})^{-DUR}} \tag{13.1.1}$$

$$\forall s \in SHOPS : \sum_{b \in BANKS} borrow_{bs} = PRICE_s \tag{13.1.2}$$

$$\forall b \in BANKS : \sum_{s \in SHOPS} borrow_{bs} \leq VMAX \tag{13.1.3}$$

$$\forall b \in BANKS, s \in SHOPS : borrow_{bs} \geq 0 \tag{13.1.4}$$

The objective is to minimize Mr. Chic's expenses, that is, to minimize the sum of annual payments he has to make. If the amount $borrow_{bs}$ is borrowed from a bank b to finance the shop s at the rate $RATE_{bs}$ over DUR years, then the annual payment pay_{bs} to be made by Mr Chic for this shop to the bank b is calculated as follows: the net present value of all DUR annual payments pay_{bs} for shop s to the bank b must equal the value of the loan $borrow_{bs}$:

$$borrow_{bs} = \sum_{t=1}^{DUR} \frac{pay_{bs}}{(1 + RATE_b)^t} = pay_{bs} \cdot \frac{1 - (1 + RATE_b)^{-DUR}}{RATE_b} \tag{13.1.5}$$

By dividing the equation by the fraction on the right hand side we obtain the following equation for the annuities pay_{bs}:

$$borrow_{bs} \cdot \frac{RATE_{bs}}{1 - (1 + RATE_{bs})^{-DUR}} = pay_{bs} \tag{13.1.6}$$

The annual payments pay_{bs} have to be made for all shops s to all banks b during DUR years. But since the annual payments are the same every year, it is sufficient to minimize the sum of the payments for one year. We thus obtain the objective function as the sum of pay_{bs} over all b and s. By replacing pay_{bs} with the left hand side of equation (13.1.6) we obtain the objective function in the form (13.1.1).

Every shop has to be completely financed, which means that the amounts borrowed for financing every shop s must correspond to its cost $PRICE_s$. This constraint is expressed through (13.1.2). The constraint (13.1.3) indicates that every bank must not finance more than $VMAX$ and the constraints (13.1.4) state the non-negativity of the variables.

13.1.2 Implementation

The following Mosel program gives the translation of the mathematical model above.

```
model "H-1 Loan choice"
 uses "mmxprs"

 declarations
  BANKS = 1..3                          ! Set of banks
  SHOPS = {"London", "Munich", "Rome"} ! Set of shops
  DUR: integer                          ! Duration of credits

  PRICE: array(SHOPS) of integer        ! Price of shops
  RATE: array(BANKS,SHOPS) of real      ! Interest rates offered by banks
  VMAX: integer                         ! Maximum loan volume per bank

  borrow: array(BANKS,SHOPS) of mpvar   ! Loan taken from banks per project
 end-declarations

 initializations from 'h1loan.dat'
  PRICE RATE VMAX DUR
 end-initializations

! Objective: interest payments
 Interest:=
  sum(b in BANKS, s in SHOPS) borrow(b,s)*RATE(b,s)/(1-(1+RATE(b,s))^(-DUR))

! Finance all projects
 forall(s in SHOPS) sum(b in BANKS) borrow(b,s) = PRICE(s)

! Keep within maximum credit volume per bank
 forall(b in BANKS) sum(s in SHOPS) borrow(b,s) <= VMAX

! Solve the problem
 minimize(Interest)

end-model
```

13.1.3 Results

To minimize his expenses, Mr. Chic finances the shop in London (shop 1) entirely through a credit from bank 1 and the shop in Munich (shop 2) through a credit from bank 3. For the shop in Rome (shop 3) he borrows €0.5 million from bank 1 and €1.2 million from bank 2. The sum of the annual payments to be made by Mr. Chic is then €822,181.

13.2 Publicity campaign

The small company, Pronuevo, launches a new product into a regional market and wishes to have a publicity campaign using different media. It therefore contacts a regional publicity agency, PRCo, that specializes in this type of regional campaign and completely hands over this task for a total budget of €250,000. The agency knows the market well, that is, the impact of publicity in a local magazine or over the radio, or as a TV spot on the regional channel. It suggests addressing the market for two months through six different media. For each medium, it knows the cost and the number of people on which this medium has an impact. An index of the quality of perception of the campaign is also known for every medium.

The publicity agency establishes a maximum number of uses of every medium (for instance, not more than eight transmissions of a TV spot). The following Table 13.2 lists all this information. Pronuevo wants the impact of the publicity campaign to reach at least 100,000 people. Which media should be chosen and in which proportions to obtain a maximum index of perception quality?

Table 13.2: Data for the publicity campaign

Number	Media type	People potentially reached	Unit cost	Maximum use	Perception quality
1	Free weekly newspaper	12,000	1,500	4 weeks	3
2	Monthly magazine	1,500	8,000	2 months	7
3	Weekly magazine	2,000	12,000	8 weeks	8
4	Radio spot	6,000	9,000	60 broadcasts	2
5	Billboard 4x3 m	3,000	24,000	4 boards	6
6	TV spot	9,000	51,000	8 broadcasts	9

13.2.1 Model formulation

We write *MEDIA* for the set of media that may be used in this campaign. The constants $REACH_m$, $COST_m$, $MAXUSE_m$, and $SCORE_m$ denote respectively the number of people potentially reached, the unit cost of using a medium, the maximum number of uses and the quality index for every medium used. The decision variables use_m that we choose are the number of uses made of every medium m during this campaign.

To start with, the budgetary constraint needs to be fulfilled: the constraint (13.2.2) indicates that the sum of the costs for every medium must not exceed the total allocated budget *BUDGET*. Every medium may be used only up to its maximum authorized number of uses (constraints (13.2.3)). The targeted number of people *TARGET* needs to be reached, that means, that the sum of the numbers of people potentially reached through every medium must be at least as large as the targeted 100,000 persons (constraint (13.2.4)). The objective function (13.2.1) has to maximize the sum of the quality indices of every media used. The decision variables use_m must be integer, constraints (13.2.5).

$$\text{maximize} \sum_{m \in MEDIA} SCORE_m \cdot use_m \tag{13.2.1}$$

$$\sum_{m \in MEDIA} COST_m \cdot use_m \leq BUDGET \tag{13.2.2}$$

$$\forall m \in MEDIA : use_m \leq MAXUSE_m \tag{13.2.3}$$

$$\sum_{m \in MEDIA} REACH_m \cdot use_m \geq TARGET \tag{13.2.4}$$

$$\forall m \in MEDIA : use_m \in \mathbb{N} \tag{13.2.5}$$

13.2.2 Implementation

This model could be implemented quite simply in a non-generic way but we have given preference to a generic form (using data arrays read from file) that is easier to modify for future extensions.

```
model "H-2 Publicity"
 uses "mmxprs"

 declarations
  MEDIA = 1..6                        ! Set of media types

  REACH: array(MEDIA) of integer      ! Number of people reached
  COST: array(MEDIA) of integer       ! Unitary cost
  MAXUSE: array(MEDIA) of integer     ! Maximum use
  SCORE: array(MEDIA) of integer      ! Quality rating (best=highest value)
  BUDGET: integer                     ! Available publicity budget
  TARGET: integer                     ! Number of people to be reached

  use: array(MEDIA) of mpvar          ! Use made of different media
 end-declarations

 initializations from 'h2publ.dat'
  REACH COST MAXUSE SCORE BUDGET TARGET
 end-initializations

! Objective: quality of perception of the campaign
 Perceive:= sum(m in MEDIA) SCORE(m)*use(m)

! Budgetary limit
 sum(m in MEDIA) COST(m)*use(m) <= BUDGET

! Outreach of campaign
 sum(m in MEDIA) REACH(m)*use(m) >= TARGET

 forall(m in MEDIA) do
  use(m) is_integer
  use(m) <= MAXUSE(m)
 end-do

! Solve the problem
 maximize(Perceive)

end-model
```

13.2.3 Results

With the program above we find a quality index of 122 units. The budget is fully used and 103,000 people are reached. As detailed in the following table, all media types except the TV spot are used. Given that the problem is of very moderate size, the LP directly calculates an integer solution.

Table 13.3: Use made of different media types

Media type	Number of uses
Free weekly newspaper	4
Monthly magazine	2
Weekly magazine	8
Radio spot	4
Billboard 4x3 m	4
TV spot	0

We should note that the problem described in this section is a much simplified version of reality. For instance, it is unlikely that the effectiveness of radio advertising is proportional to the number of broadcasts of a particular advert. Nor are adverts in different media independent of each other. In practice, much more sophisticated models will be used.

13.3 Portfolio selection

A consultant in finance has to choose for one of his wealthy female clients a certain number of shares in which to invest. She wishes to invest € 100,000 in 6 different shares. The consultant estimates for her the return on investment that she may expect for a period of six months. The following table gives for each share its country of origin, the category (T: technology, N: non-technology) and the expected return on investment (ROI). The client specifies certain constraints. She wishes to invest at least € 5,000 and at most € 40,000 into any share. She further wishes to invest half of her capital in European shares and at most 30% in technology. How should the capital be divided among the shares to obtain the highest expected return on investment?

Table 13.4: List of shares

Number	Origin	Category	Expected ROI
1	Japan	T	5.3%
2	UK	T	6.2%
3	France	T	5.1%
4	USA	N	4.9%
5	Germany	N	6.5%
6	France	N	3.4%

13.3.1 Model formulation

In the mathematical formulation, we write *SHARES* for the set of shares chosen for the investment, *CAPITAL* the total capital that is to be invested, RET_s the expected ROI. *EU* denotes the subset of the shares that are of European origin and *TECHNOLOGY* the set of technology values. The decision variables buy_s denote the amount of money invested in share s. Every variable is bounded, at least *VMIN* = 5000 and at most *VMAX* = 40000 have to be invested into every share. The constraint (13.3.1) establishes these bounds on the variables buy_s.

$$\forall s \in SHARES : VMIN \leq buy_s \leq VMAX \tag{13.3.1}$$

At most 30% of the values may be technology values, *i.e.* the sum invested into this category must not exceed € 30,000, constraint (13.3.2). The investor also insists on spending at least 50% on European shares, that is, at least € 50,000, constraint (13.3.3).

$$\sum_{s \in TECHNOLOGY} buy_s \leq 0.3 \cdot CAPITAL \tag{13.3.2}$$

$$\sum_{s \in EU} buy_s \geq 0.5 \cdot CAPITAL \tag{13.3.3}$$

The constraint (13.3.8) specifies the the total invested sum must correspond to the initial capital *CAPITAL*. The constraints (13.3.9) are the usual non-negativity constraints. The objective function (13.3.4) maximizes the return on investment of all shares.

$$\text{maximize} \sum_{s \in SHARES} \frac{RET_s}{100} \cdot buy_s \tag{13.3.4}$$

$$\forall s \in SHARES : VMIN \leq buy_s \leq VMAX \tag{13.3.5}$$

$$\sum_{s \in TECHNOLOGY} buy_s \leq 0.3 \cdot CAPITAL \tag{13.3.6}$$

$$\sum_{s \in EU} buy_s \geq 0.5 \cdot CAPITAL \tag{13.3.7}$$

$$\sum_{s \in SHARES} buy_s = CAPITAL \tag{13.3.8}$$

$$\forall s \in SHARES : buy_s \geq 0 \tag{13.3.9}$$

After some further discussion, it turns out that the client actually does not wish to invest *VMIN* into **every** share the consultant has chosen for her. She is unwilling to have small share holdings. If she buys any shares in a company she wants to invest at least € 5,000. The model therefore needs to be modified: instead of constraining every variable buy_s to take a value between *VMIN* and *VMAX*, it must either lie in

this interval or take the value 0. This type of variable is known as **semi-continuous variable**. In this new model, we replace (13.3.5) by the following constraint (13.3.10):

$$\forall s \in SHARES : buy_s = 0 \vee VMIN \leq buy_s \leq VMAX \tag{13.3.10}$$

13.3.2 Implementation

The translation of the mathematical model into the following Mosel program is straightforward. In the implementation, we have turned the limits on technology values and European shares into **parameters** of the model (`MAXTECH` and `MINEU`). Similarly, the bounds `VMIN` and `VMAX` are defined as parameters.

```
model "H-3 Portfolio"
 uses "mmxprs"

 parameters
  MAXTECH = 0.3                    ! Maximum investment into tech. values
  MINEU = 0.5                      ! Minimum investment into European shares
  VMIN = 5000                      ! Minimum amount for a single value
  VMAX = 40000                     ! Maximum amount for a single value
 end-parameters

 declarations
  SHARES = 1..6                    ! Set of shares

  RET: array(SHARES) of real       ! Estimated return in investment
  CAPITAL: integer                 ! Capital to invest
  EU: set of integer               ! European values among the shares
  TECHNOLOGY: set of integer       ! Technology values among shares

  buy: array(SHARES) of mpvar      ! Amount of values taken into portfolio
 end-declarations

 initializations from 'h3portf.dat'
  RET CAPITAL EU TECHNOLOGY
 end-initializations

! Objective: total return
 Return:= sum(s in SHARES) RET(s)/100*buy(s)

! Requirements concerning portfolio composition
 sum(s in TECHNOLOGY) buy(s) <= MAXTECH*CAPITAL
 sum(s in EU) buy(s) >= MINEU*CAPITAL

! Total capital to invest
 sum(s in SHARES) buy(s) = CAPITAL

 forall(s in SHARES) do
  VMIN <= buy(s)
  buy(s) <= VMAX
 end-do

! Solve the problem
 maximize(Return)

end-model
```

To define the variables `buy(s)` as semi-continuous variables, we need to replace the lower bound constraints

```
  VMIN <= buy(s)
```

by the line

```
  buy(s) is_semcont(VMIN)
```

13.3.3 Results

The solution to the problem indicates the sums that should be invested into the different shares. The total expected ROI with the initial problem formulation is € 5,755 for six months, for the problem with semi-continuous variables it is € 5,930. The funds are divided among the shares as shown in the following table.

Table 13.5: Optimal portfolio

Share number	Invested amount (in €) $buy_s \geq VMIN$	buy_s semi-cont.
1	5,000	0
2	20,000	30,000
3	5,000	0
4	25,000	30,000
5	40,000	40,000
6	5,000	0

As mentioned in the introduction to this chapter, Mathematical Programming may be used as a simulation tool in financial applications. In our case, what would happen if we allowed investing up to 40% into technology values? Since the corresponding limit `MAXTECH` has been defined as a parameter, there is no need to modify the Mosel program to test the result for this value. It is sufficient to set the new value of the parameter when calling Mosel. With the Mosel Command Line Interpreter this may be done with the following command to execute the model:

```
mosel -c "exec h3portf 'MAXTECH=0.4'"
```

With this increased limit on technology values, the expected return becomes € 5,885 for the first model (and € 6,060 for the second). Compared to the previous plan, € 10,000 are transfered from share 4 to share 2, and the amounts invested in the other shares remain the same. By varying the different parameters (it is possible to set new values for several parameters at the same time), the consultant may be able to satisfy his client.

13.4 Financing an early retirement scheme

The National Agricultural Bank (NAB) decides to establish an early retirement scheme for fifteen employees who are taking early retirement. These employees will retire over a period of seven years starting from the next year. To finance this early retirement scheme, the bank decides to invest in bonds for this seven-year period. The necessary amounts to cover the pre-retirement leavers are given in the following Table 13.6; they have to be paid at the beginning of every year.

Table 13.6: Amounts required every year

Year	1	2	3	4	5	6	7
Amount (in 1000 €)	1000	600	640	480	760	1020	950

For the investments, the bank decides to take three different types of bonds, SNCF bonds, Fujitsu bonds, and Treasury bonds. The money that is not invested in these bonds is placed as savings with a guaranteed yield of 3.2%. Table 13.7 lists all information concerning the yield and the durations of the bonds and also the value of a bond. In this type of bond, it is only possible to buy an integer number of bonds, and the invested capital remains locked in for the total duration of the bond. Every year, only the interest on the capital is distributed. The person in charge of the retirement plan decides to buy bonds at the beginning of the first year, but not in the following years. How should he organize the investments in order to spend the least amount of money to cover the projected retirement plan?

Table 13.7: Information about loans

Loan	Value of bonds (in 1000 €)	Interest	Duration
SNCF	1.0	7.0%	5 years
Fujitsu	0.8	7.0%	4 years
Treasury	0.5	6.5%	6 years

13.4.1 Model formulation

Let $YEARS = \{1, \ldots, NT\}$ be the time period over which the early retirement scheme stretches, and *BONDS* the set of loans. *capital* denotes the total capital that needs to be invested in the first year in order to be able to finance the scheme over all seven years. Let $VALUE_b$ be the unit price for a bond b, DUR_b its duration, and $RATE_b$ the annual interest on bond b. The variable buy_b represents the number of bonds of type b bought in the first year, the variable $invest_t$ the sum invested (into forms of savings other than the bonds) in year t. The funds required by the early retirement plan per year t are denoted by DEM_t.

The general principle of the constraints is the following: every year, the set of interest payments has to cover the investments of this year and the amount required by the retirement plan. The constraint (13.4.1) represents the investment in the first year, the year during which the bonds are bought. The total invested capital minus the price paid for the bonds and minus the investment of the first year must be equal to the funds required in the first year.

$$capital - \sum_{b \in BONDS} VALUE_b \cdot buy_l - invest_1 = DEM_1 \tag{13.4.1}$$

While no bond has come to term, that is during the three years following the first, no more money is invested into bonds, but we receive the interest (first sum of constraint (13.4.2)). We also receive the interest and the amount invested into other forms of savings than bonds in the preceding year. From these benefits we have to subtract the investments (other than loans) and the funds required for the current year to obtain an equilibrium, constraints (13.4.2).

$$\forall t = 2, \ldots, 4 : \sum_{b \in BONDS} VALUE_b \cdot buy_b \cdot \frac{RATE_b}{100} + \left(1 + \frac{3.2}{100}\right) \cdot invest_{t-1} - invest_t = DEM_t \tag{13.4.2}$$

At the beginning of year 5, the Fujitsu bond falls due and similarly at the beginning of year 6, the EDF bond falls due. The capital invested into the corresponding bond is recovered and available for a new investment. We also receive the interest for the preceding year for all bonds. The rest of constraint (13.4.3) is identical to the constraints (13.4.2).

$$\begin{aligned} \forall t = 5, 6 : & \sum_{\substack{b \in BONDS \\ DUR_b = t-1}} VALUE_b \cdot buy_l \cdot \left(1 + \frac{RATE_b}{100}\right) + \sum_{\substack{b \in BONDS \\ DUR_b \geq t}} VALUE_b \cdot buy_b \cdot \frac{RATE_b}{100} \\ & + \left(1 + \frac{3.2}{100}\right) \cdot invest_{t-1} - invest_t = DEM_t \end{aligned} \tag{13.4.3}$$

In the last year, only the Treasury bond that ran over 6 years remains. We recover the capital initially invested into this bond and the investments from the preceding year. In this year, there is no new investment (constraint 13.4.4)

$$\sum_{\substack{b \in BONDS \\ DUR_b = 6}} VALUE_b \cdot buy_b \cdot \left(1 + \frac{RATE_b}{100}\right) + \left(1 + \frac{3.2}{100}\right) \cdot invest_6 = DEM_7 \tag{13.4.4}$$

To the constraints (13.4.1) – (13.4.4) we still need to add the objective function (13.4.5) that minimizes the initially invested capital, and to declare the definition spaces of the variables (constraints (13.4.6) – (13.4.8))

$$\text{minimize } capital \tag{13.4.5}$$
$$cap \geq 0 \tag{13.4.6}$$
$$\forall t \in YEARS : invest_t \geq 0 \tag{13.4.7}$$
$$\forall b \in BONDS : buy_b \in \mathbb{N} \tag{13.4.8}$$

13.4.2 Implementation

In the following Mosel program, the constraints for the annual balances of years 2-7 are combined into a single constraint expression selecting the corresponding constraint terms through the inline `if` function. To simplify the notation of the constraints, we define an auxiliary array `RET` representing the annual interest per bond.

```
model "H-4 Retirement"
 uses "mmxprs"

 declarations
  BONDS = {"SNCF","Fujitsu","Treasury"}  ! Set of bonds
  NT = 7                                 ! Length of planning period
  YEARS = 1..NT

  DEM: array(YEARS) of integer           ! Annual payments for retirement
  VALUE: array(BONDS) of real            ! Unit price of bonds
```

```
  RATE: array(BONDS) of real          ! Remuneration rates paid by bonds
  RET: array(BONDS) of real           ! Unit annual interest of bonds
  DUR: array(BONDS) of real           ! Duration of loans
  INTEREST: real                      ! Interest for other secure investment

  buy: array(BONDS) of mpvar          ! Number of bonds acquired
  invest: array(YEARS) of mpvar       ! Other annual investment
  capital: mpvar                      ! Total capital required
 end-declarations

 initializations from 'h4retire.dat'
  DEM VALUE RATE DUR INTEREST
 end-initializations

 forall(b in BONDS) RET(b):= VALUE(b)*RATE(b)/100

! Annual balances
 capital - sum(b in BONDS) VALUE(b)*buy(b) - invest(1) = DEM(1)

 forall(t in 2..NT)
  sum(b in BONDS | DUR(b)+1>=t) (RET(b)*buy(b) +
                                 if(DUR(b)+1=t, VALUE(b)*buy(b), 0)) +
    (1+INTEREST/100)*invest(t-1) - if(t<NT, invest(t), 0) = DEM(t)

 forall(b in BONDS) buy(b) is_integer

! Solve the problem: minimize invested capital
 minimize(capital)

end-model
```

13.4.3 Results

The initially required capital is €4548.92k. The optimal solution consists of investing €911k in SNCF bonds (911 bonds), €597.6k in Fujitsu bonds (747 bonds), and €880k in Treasury coupons (1760 coupons). The following table lists the investments into forms of savings other than loans for the first six years.

Table 13.8: Investments (other than loans)

Year	1	2	3	4	5	6
Amount (in 1000 €)	1160.317	760.249	307.379	0.017	0.420	12.403

13.5 Family budget

The mother of a family wishes to use her son's computer. On the internet she has found optimization software and would now like to formulate a mathematical model to help plan their annual budget. She has prepared a list of the monthly expenses and receipts of money. Every month the expenses for living are €550. The monthly rent for the apartment is €630. She also budgets €135 for telephone every two months, €850 for gas and electricity bills every six months, €340 per month for the car and €100 of tax every four months. Her receipts of money are a monthly payment of €150 of state allowance for families with dependent children and a net salary of €1900 per month. She knows that she pays at least €165 for leisure every month (subscription to the swimming pool for the older children, football club for the youngest, gym for herself) but she would like to spend more (restaurant, cinema, holidays). How should the budget be balanced all through the year to maximize the money available for leisure?

13.5.1 Model formulation

Let *MONTHS* be the set of twelve month representing the year to plan and *ITEMS* the set of expense types. We write $EXPENSE_i$ for the amount of an expense and $FREQ_i$ for its periodicity (*e.g.* $FREQ_i = 6$ means that i is a 6-monthly expense). The salary is denoted by *INCOME*, and the allowance by *ALLOW*. We use two types of variables: $hobby_m$, the amount that may be spent for leisure in month m, and $save_m$, the savings of month m. Every month, the family wishes to spend at least $HMIN = €165$ for leisure. This is expressed by the constraints (13.5.1).

$$\forall m \in MONTHS : hobby_m \geq HMIN \tag{13.5.1}$$

For the system to work, the budget must be balanced. The expenses in every month must not exceed the receipts of money of this month, but it is possible to put aside some money to save it for a month with higher expenses. The constraints (13.5.2) express the fact that the expenses of the current month plus the savings of this month and the money spent for leisure must be less than or equal to the perceived salary, plus the allowance, plus the savings from the preceding month. Since no information about savings at the beginning of the planning period (variable $save_0$) is given, we assume them to be 0. The operator *mod* denotes the remainder of integer division (*e.g.* 7 mod 6 = 1, that is, 6-monthly payments are only taken into account in months 6 and 12).

$$\forall m \in MONTHS: \sum_{\substack{i \in ITEMS \\ m \bmod FREQ_i = 0}} EXPENSE_i + save_m + hobby_m \leq INCOME + ALLOW + save_{m-1} \tag{13.5.2}$$

The objective function is to maximize the total amount available for leisure in the entire year (13.5.3). The non-negativity constraints for the variables $hobby_m$ are superseded by the constraints (13.5.1).

$$\text{maximize} \sum_{m \in MONTHS} hobby_m \tag{13.5.3}$$

$$\forall m \in MONTHS: hobby_m \geq HMIN \tag{13.5.4}$$

$$\forall m \in MONTHS: \sum_{\substack{i \in ITEMS \\ m \bmod FREQ_i = 0}} EXPENSE_i + save_m + hobby_m \leq INCOME + ALLOW + save_{m-1} \tag{13.5.5}$$

$$\forall m \in MONTHS: save_m \geq 0 \tag{13.5.6}$$

13.5.2 Implementation

This model is very similar to a production planning model with transfer of stocks from one time period to the next. To deal with the different periodicities of the expenses, we use the Mosel operator `mod`.

For the savings at the beginning of the first time period, we may either define a variable $save_0$ and fix its value to 0 or, as shown below, directly employ the value 0 in the corresponding constraint.

```
model "H-5 Family budget"
 uses "mmxprs"

 declarations
  MONTHS = 1..12                      ! Time period
  ITEMS: set of string                ! Set of shops

  INCOME, ALLOW: integer              ! Monthly income and allowance
  HMIN: integer                       ! Min. amount required for hobbies
  EXPENSE: array(ITEMS) of integer    ! Expenses
  FREQ: array(ITEMS) of integer       ! Frequency (periodicity) of expenses

  hobby: array(MONTHS) of mpvar       ! Money to spend for leisure/hobbies
  save: array(MONTHS) of mpvar        ! Savings
 end-declarations

 initializations from 'h5budget.dat'
  INCOME ALLOW HMIN
  [EXPENSE, FREQ] as 'PAYMENT'
 end-initializations

! Objective: money for hobby
 Leisure:= sum(m in MONTHS) hobby(m)

! Monthly balances
 forall(m in MONTHS)
  sum(i in ITEMS | m mod FREQ(i) = 0) EXPENSE(i) + hobby(m) +
                  save(m) <= INCOME + ALLOW + if(m>1, save(m-1), 0)

 forall(m in MONTHS) hobby(m) >= HMIN

! Solve the problem
 maximize(Leisure)

end-model
```

13.5.3 Results

In the course of the year, the family may spend up to €3,550 for leisure. The following table shows a possible plan of their monthly leisure budget (there are other equivalent solutions).

Table 13.9: Monthly leisure budget and savings

Month	1	2	3	4	5	6	7	8	9	10	11	12
Hobby	530	395	405	165	165	165	530	295	405	165	165	165
Savings	0	0	125	255	620	0	0	0	125	355	720	0

13.6 Choice of expansion projects

The large company Tatayo in the north of Italy has specialized in the construction of cars for more than ten years. The company wishes to expand and has issued internally a call for proposals for expansion projects for a planning period of five years. Among the many, often cranky, propositions the management has retained five projects. Every project has an annual cost and is designed to produce a benefit after five years. Table 13.10 gives a list of the projects with short descriptions and the expected benefit after five years. The forecast annual costs of the projects for the next five years are detailed in Table 13.11, together with the funds available. Which project(s) should the management choose now to maximize the total benefit after five years?

Table 13.10: Estimated benefits of the projects (in million €)

Project	Description	Expected benefit
1	Expand assembly line	10.8
2	Reorganize the main shop	4.8
3	New painting facilities	3.2
4	Research for a new concept car	4.44
5	Reorganize the logistics chain	12.25

Table 13.11: Annual costs of projects and available funds (in million €)

Project	Year 1	Year 2	Year 3	Year 4	Year 5
1	1.8	2.4	2.4	1.8	1.5
2	1.2	1.8	2.4	0.6	0.5
3	1.2	1.0	0	0.48	0
4	1.4	1.4	1.2	1.2	1.2
5	1.6	2.1	2.5	2.0	1.8
Funds	4.8	6.0	4.8	4.2	3.5

13.6.1 Model formulation

We write *PROJECTS* for the set of projects retained by the management, and *TIME* for the set of time periods (years) within the planning horizon. The benefit from every project p is RET_p and $COST_{pt}$ the cost of project p in year t. CAP_t denotes the capital available for funding projects in year t. The binary variable $choose_p$ has the value 1 if project p is chosen and 0 otherwise. The only constraints given in this problem are the limits on the capital that the management is prepared to invest every year. The constraints (13.6.2) ensure that the sum of the costs $COST_{pt}$ of the chosen projects ($choose_p = 1$) does not exceed the available capital CAP_t in year t. The objective function (13.6.1) maximizes the total profit, that is, the sum of the benefits RET_p from the chosen projects. The variables $choose_p$ are binary (13.6.3).

$$\text{maximize} \sum_{p \in PROJECTS} RET_p \cdot choose_p \tag{13.6.1}$$

$$\forall t \in TIME : \sum_{p \in PROJECTS} COST_{pt} \cdot choose_p \leq CAP_t \tag{13.6.2}$$

$$\forall p \in PROJECTS : choose_p \in \{0, 1\} \tag{13.6.3}$$

13.6.2 Implementation

The following Mosel program implements the mathematical model above.

```
model "H-6 Expansion"
 uses "mmxprs"

 declarations
  PROJECTS = 1..5                     ! Set of possible projects
  TIME = 1..5                         ! Planning period

  COST: array(PROJECTS,TIME) of real  ! Annual costs of projects
  CAP: array(TIME) of real            ! Annually available capital
  RET: array(PROJECTS) of real        ! Estimated profits
  DESCR: array(PROJECTS) of string    ! Description of projects

  choose: array(PROJECTS) of mpvar    ! 1 if project is chosen, 0 otherwise
 end-declarations

 initializations from 'h6expand.dat'
  COST CAP RET DESCR
 end-initializations

! Objective: Total profit
 Profit:= sum(p in PROJECTS) RET(p)*choose(p)

! Limit on capital availability
 forall(t in TIME) sum(p  in PROJECTS) COST(p,t)*choose(p) <= CAP(t)

 forall(p in PROJECTS) choose(p) is_binary

! Solve the problem
 maximize(Profit)

end-model
```

13.6.3 Results

When solving the above program as is, we obtain a solution with a total profit of € 19.89 million: the projects 'New painting facility', 'Concept car', and 'Reorganize logistics' are chosen.

We may try to experiment a little with this problem. If we solve the model again after modifying the call to the optimization as follows:

```
  maximize(XPRS_LIN, Profit)
```

the linear relaxation of the problem is solved (that is, neglecting the integrality condition for the variables $choose_p$). Looking at the solution of this relaxed problem, we find that the variables for choosing projects 'New paint facility' and 'Reorganize logistics' are at 1, the variable for 'Expand assembly line' is almost 1, more precisely 0.9542, and the value of 'Reorganize shop' is almost 0 (0.0042). The total profit of the linear solution is € 25.775M. One might wonder what would happen if we 'rounded' the variable for 'Expand assembly line' to 1, thus forcing the acceptance of project 1. This may be done by adding the following line to the Mosel program before calling the optimization algorithm:

```
  choose(1)=1
```

With this additional constraint, the solution consists of choosing the projects 'Expand assembly line' (the project that was forced to be accepted) and 'New painting facility', with a total profit of € 14M. This solution demonstrates that rounding fractional variables to the closest integer value is not always a good heuristic for creating integer feasible solutions from the solution to the LP relaxation.

13.7 Mean variance portfolio selection

An investor wishes to invest a certain amount of money. He is evaluating four different securities (assets) for his investment. The securities are US Treasury Bills ('T-bills'), a computer hardware company, a computer software company, and a high-risk investment in a theater production. He estimates the mean yield on each dollar he invests in each of the securities, and also adopts the Markowitz idea of getting estimates of the variance/covariance matrix of estimated returns on the securities. (For example, hardware

and software company worths tend to move together, but are oppositely correlated with the success of theatrical production, as people go to the theater more when they have become bored with playing with their new computers and computer games.) The return on theatrical productions are highly variable, whereas the T-bill yield is certain. The estimated returns and the variance/covariance matrix are given in Table 13.12

Table 13.12: Estimated returns and variance/covariance matrix

	Hardware	Software	Show-biz	T-bills
Estimated return	8	9	12	7
Hardware	4	3	-1	0
Software	3	6	1	0
Show-biz	-1	1	10	0
T-bills	0	0	0	0

Question 1: Which investment strategy should the investor adopt to minimize the variance subject to getting some specified minimum target yield?

Question 2: Which is the least variance investment strategy if the investor wants to choose at most two different securities (again subject to getting some specified minimum target yield)?

13.7.1 Model formulation for question 1

The objective of this problem is to minimize the variance subject to getting some specified minimum target yield, say $TARGET$. Let RET_s be the expected return of a security s and VAR_{st} the variance/covariance matrix of estimated returns on the securities. If we introduce decision variables $frac_s$ to be the fraction of the total capital that is invested in security s, then we obtain the following **Quadratic Program**:

$$\text{minimize} \sum_{s \in SECS} \sum_{t \in SECS} VAR_{st} \cdot frac_s \cdot frac_t \tag{13.7.1}$$

$$\sum_{s \in SECS} frac_s = 1 \tag{13.7.2}$$

$$\sum_{s \in SECS} RET_s \cdot frac_s \geq TARGET \tag{13.7.3}$$

$$\forall s \in SECS : frac_s \geq 0 \tag{13.7.4}$$

The objective function (13.7.1) is the variance of the estimated returns. Constraint (13.7.2) says that the investor wishes to spend all the money. The desired target yield is enforced through constraint (13.7.3). The constraints (13.7.4) are the usual non-negativity constraints.

Note that this is not a Linear Programming problem as we have product terms in the objective function.

13.7.2 Implementation for question 1

The Mosel implementation of this problem is straightforward. Notice that in addition to the Xpress-Optimizer (module `mmxprs`) we are using the Mosel module `mmquad` which contains all that is required to handle QP problems.

```
model "H-7 QP Portfolio"
 uses "mmxprs", "mmquad"

 parameters
  TARGET = 7.0                      ! Minimum target yield
 end-parameters

 declarations
  SECS = 1..4                       ! Set of securities

  RET: array(SECS) of real          ! Expected yield of securities
  VAR: array(SECS,SECS) of real     ! Variance/covariance matrix of
                                    ! estimated returns

  frac: array(SECS) of mpvar        ! Fraction of capital used per security
  buy: array(SECS) of mpvar         ! 1 if asset is in portfolio, 0 otherwise
 end-declarations
```

```
 initializations from 'h7qportf.dat'
  RET VAR
 end-initializations

! Objective: mean variance
 Variance:= sum(s,t in SECS) VAR(s,t)*frac(s)*frac(t)

! Spend all the capital
  sum(s in SECS)  frac(s) = 1

! Target yield
 sum(s in SECS) RET(s)*frac(s) >=  TARGET

! Solve the problem
 minimize(Variance)
end-model
```

13.7.3 Results for question 1

Running the model thus, so as to specify a target yield of 8.5%

```
 mosel -s -c "exec h7qportf 'TARGET=8.5'"
```

we get a minimum variance of 0.72476. We should invest 15.14% into hardware, 4.32% into software, 25.24% into show-biz, and 55.29% into T-bills.

13.7.4 Model formulation for question 2

For the second problem we need to express the fact that no more than a certain number *MAXASSETS* of different securities may be bought. The previous model may be extended by adding binary variables buy_s to represent whether we do ($buy_s = 1$) or do not ($buy_s = 0$) buy any of the asset for our portfolio. If we add to the model above the constraints

$$\sum_{s \in SECS} buy_s \leq MAXASSETS \tag{13.7.5}$$

$$\forall s \in SECS : frac_s \leq buy_s \tag{13.7.6}$$

$$\forall s \in SECS : buy_s \in \{0, 1\} \tag{13.7.7}$$

then we can see that for a $frac_s$ to be bigger than 0, buy_s must be bigger than 0 (constraints (13.7.6)) and so must be 1 as it is binary (13.7.7). The constraint (13.7.5) establishes the limit on the number of different securities bought.

This problem is actually a **Mixed Integer Quadratic Program** (MIQP) as it has elements of Quadratic Programming and also Integer Programming.

13.7.5 Implementation for question 2

The Mosel implementation of the second problem is similar to the first, but since there are changes at several places in the model, we show the entire program again.

```
model "H-7 QP Portfolio (2)"
 uses "mmxprs", "mmquad"

 parameters
  TARGET = 7.0                    ! Minimum target yield
  MAXASSETS = 4                   ! Maximum number of assets in portfolio
 end-parameters

 declarations
  SECS = 1..4                     ! Set of securities

  RET: array(SECS) of real        ! Expected yield of securities
  VAR: array(SECS,SECS) of real   ! Variance/covariance matrix of
                                  ! estimated returns

  frac: array(SECS) of mpvar      ! Fraction of capital used per security
  buy: array(SECS) of mpvar       ! 1 if asset is in portfolio, 0 otherwise
 end-declarations
```

```
 initializations from 'h7qportf.dat'
  RET VAR
 end-initializations

! Objective: mean variance
 Variance:= sum(s,t in SECS) VAR(s,t)*frac(s)*frac(t)

! Spend all the capital
  sum(s in SECS)  frac(s) = 1

! Target yield
 sum(s in SECS) RET(s)*frac(s) >=  TARGET

! Limit the total number of assets
 sum(s in SECS) buy(s) <= MAXASSETS

 forall(s in SECS) do
  buy(s) is_binary
  frac(s) <= buy(s)
 end-do

! Solve the problem
 minimize(Variance)
end-model
```

13.7.6 Results for question 2

Running the model with the following parameter settings,

```
mosel -s -c "exec h7qportf 'TARGET=8.5, MAXASSETS=2'"
```

that is, a target yield of 8.5% and at most two assets in the portfolio, we get a minimum variance of 0.9. We need to invest 30% into show-biz and 70% into T-bills. The results for questions 1 and 2 are summarized in Table 13.13.

Table 13.13: Results with *TARGET* = 8. 5

	Percentage of total investment	
	Question 1	**Question 2**
Hardware	15.14%	0%
Software	4.33%	0%
Show-biz	25.24%	30%
T-bills	55.29%	70%
Variance	0.72	0.9

Notice that because we are restricting ourselves to holding at most two securities, the variance for a given yield has gone up compared with the case where we can hold any number of securities.

The second model is probably the more realistic one: in practical applications, there are many thousands of possible securities, and constraints that specify that no more than a certain number of different securities may be bought stop us having the managerial problem of being forced to look after too many securities.

13.7.7 Extension

Finally, we should note that in practice deciding on a reasonable value for an acceptable *TARGET* / *Variance* pair is not easy, and the decision maker might want to study a large range of possible combinations. In the implementation we may simply add a loop around the optimization (and solution output). `TARGET` is no longer a parameter and is therefore now declared in the `declarations` block. The possible values of `TARGET` range between the smallest and the largest value of the mean returns of all securities, enumerated in steps of 0.1. Note that in this case the constraint on the target yield must be named (`Rmean`), so it gets replaced each time through the loop.

```
 declarations
  TARGET: real
 end-declarations

 TARGET:= min(s in SECS) RET(s)
```

```
repeat
 Rmean:= sum(s in SECS) RET(s)*frac(s) >=  TARGET
 minimize(Variance)
 writeln("Target: ", TARGET, " minimum variance: ", getobjval)
 TARGET+=0.1
until(TARGET>max(s in SECS) RET(s) )
```

The results of such a study may be plotted as a graph, for instance using the graphing capabilities of Xpress-IVE.

13.8 References and further material

The problems presented here are relatively simple, but they bring into play uncertainity in the data. Being able to solve a model quickly to optimality allows us to use the model in simulations.

Often the players in the world of finance (mainly bankers) do not worry much about optimization problems. They use the software and procedures established by their management. The programs developed internally usually do not perform any optimization, but more likely a simple simulation based on priority rules. These rules are based on the principle that one needs to invest into the product(s) with the best ratio of benefit to cost. The problem hence resides in the reliability of the data and in the interpretation of the results that is made afterwards. The events on the stock markets over the last few years, especially for technology values, are an additional proof of the need to consider data subject to changes.

The Quadratic Programming (QP) problem in Section 13.7 stems from Manne [Man86]. In general, QP problems occur much less frequently than LP or MIP problems, but many of these occurrences are problems related to finance.

Operations Research techniques have become widespread in the financial sector. For an overview, see the excellent book by Winston [Win98] and the one by Zenios [Zen96].

Chapter 14
Timetabling and personnel planning

Personnel management is a very sensitive subject in the life of a company. It is determined by a large number of qualitative and psychological factors that, a priori, do not make this field well suited for quantitative modeling through Mathematical Programming. One may even put forward two major reasons that put Operations Research on hold in the 70s, until its renewed flowering in the 90s: serious mistakes were committed by treating humans as a resource just like others in the models, and furthermore, the researchers at that time did not have the computational means to satisfy their ambitions.

Optimization is currently considered as providing useful help for timetabling of personnel, which can be a real headache, and also for staffing, that is, the planning of the requirements for human resources. The key to preventing negative reactions/rejections is never to impose a solution found by the computer by force. Indeed, many subtle sociological and psychological constraints are very difficult to take into account. This makes interactive decision making tools a preferred choice, where the mathematical model makes suggestions and the human being plans by interactively adjusting the solution to his exact needs.

The first problem consists of assigning persons to machines to maximize the productivity of a workshop. We shall then see three timetabling problems: for nurses in a hospital (Section 14.2), for college courses (Section 14.3), and for exams in a technical university (Section 14.4). The problem in Section 14.5 shows how the benefit from a production unit may be optimized by moving personnel from one line to another. The chapter ends with the planning of recruitment and leaving personnel at a construction site.

14.1 Assigning personnel to machines

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 14.1 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 14.1: 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 $p \rightarrow m$ 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 problem should then be solved to optimality using Mathematical Programming. And finally, solve the same problem to optimality, but for machines working in series.

14.1.1 Model formulation

14.1.1.1 Parallel machines

This problem type is well known under the name of the **assignment problem**. It also occurs in Chapter 11 (flight connections). 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*. A first idea for encoding an assignment would be to 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. With these variables it is unfortunately not possible to formulate in a linear form the constraints that a person has to be on a single machine and a machine only takes one operator. These constraints become easy to express if we instead use N^2 binary variables $assign_{pm}$ that are 1 if and only if person *p* is assigned to machine *m* (14.1.1).

$$\forall p \in PERS, m \in MACH : assign_{pm} \in \{0, 1\} \tag{14.1.1}$$

The fact that a person *p* is assigned to a single machine *m* is by (14.1.2). Similarly, to express that a machine *m* is operated by a single person *p*, we have constraints (14.1.3).

$$\forall p \in PERS : \sum_{m \in MACH} assign_{pm} = 1 \tag{14.1.2}$$

$$\forall m \in MACH : \sum_{p \in PERS} assign_{pm} = 1 \tag{14.1.3}$$

The objective function to be maximized sums the $OUTP_{pm}$ for the variables $assign_{pm}$ that are at 1 (14.1.4). The resulting mathematical model is formed by the lines (14.1.1) to (14.1.4).

$$\text{maximize} \sum_{p \in PERS} \sum_{m \in MACH} OUTP_{pm} \cdot assign_{pm} \tag{14.1.4}$$

The assignment problem may be looked at as a flow problem. It is sufficient to define a bipartite labeled graph *G* = (*PERS, MACH, ARCS, OUTP*), where *PERS* is a set of nodes representing the personnel, *MACH* a set of nodes for the machines, and *ARCS* the set of arcs describing the possible assignments of workers to machines. In our case, there are N^2 arcs, but it would be possible to prohibit certain assignments in the general case. Every arc (*p, m*) has infinite capacity and is labeled with the cost (productivity) $OUTP_{pm}$. We create a source node *SOURCE* that is connected to every person-node *p* by an arc (*SOURCE, p*) of capacity 1. We then also create a sink node *SINK* to which are connected all machine-nodes *m* by arcs (*m, SINK*), also of capacity 1. An optimal assignment corresponds to a flow in *G* of throughput *N* with maximum total cost. Due to the unit capacities, the flow will trace *N* disjunctive paths from *SOURCE* to *SINK* that indicate the assignments.

The linear solution to the assignment problem is automatically integer. The constraints (14.1.1) may therefore be replaced by simple non-negativity conditions. It is not necessary to specify that the variables $assign_{pm}$ must not be larger than 1 because this is guaranteed through the constraints (14.1.2) and (14.1.3). The model can be modified to deal with sets of personnel and machines of different size. If, for instance, we have more workers than machines, we keep the constraints (14.1.3) so that every machine receives an operator, but we replace the constraints (14.1.2) by (14.1.5).

$$\forall p \in PERS : \sum_{m \in MACH} assign_{pm} \leq 1 \tag{14.1.5}$$

Certain assignments may be infeasible. In such a case the value of the corresponding variable $assign_{pm}$ will be forced to 0, or a highly negative number (that is, with a very large absolute value) is used for $OUTP_{pm}$. If the graph of possible assignments is given instead of the productivity matrix, it is also possible to generate the variables $assign_{pm}$ merely for the arcs (*p, m*) that are part of the graph.

14.1.1.2 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^2 variables $assign_{pm}$ (14.1.1) and the constraints (14.1.2) and (14.1.3). To this we add a non-negative 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**. To obtain a linear formulation of such problem, the procedure is always the same:

- A variable bounding the productivities on every machine from below is defined (here *pmin*).
- Constraints are added that bound every productivity by *pmin*.
- We maximize *pmin* which implies that in the optimum at least one productivity is equal to *pmin* and hence defines the bottleneck.

Two groups of constraints are possible to bound the productivities by *pmin*: the N^2 constraints (14.1.6) or the N constraints (14.1.7). In the latter, a single variable $assign_{pm}$ has the value 1 due to the constraints (14.1.2). The resulting mathematical model is given by the lines (14.1.1) to (14.1.3), the constraints (14.1.6) or (14.1.7), the non-negativity condition for *pmin* and the new objective function: maximize *pmin*. Due to the constraints (14.1.6) or (14.1.7), the mathematical model is no longer a classical assignment problem and integrality of the LP solution values is no longer guaranteed. We therefore have to keep the constraints (14.1.1).

$$\forall p \in PERS, m \in MACH : OUTP_{pm} \cdot assign_{pm} \geq pmin \tag{14.1.6}$$

$$\forall p \in PERS : \sum_{m \in MACH} OUTP_{pm} \cdot assign_{pm} \geq pmin \tag{14.1.7}$$

14.1.2 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. We also add the necessary bounding constraints (choosing constraints 14.1.7) for the variables $assign_{pm}$ and turn these variables into binaries.

```
model "I-1 Personnel assignment"
 uses "mmxprs"

 declarations
  PERS = 1..6                        ! Personnel
  MACH = 1..6                        ! Machines

  OUTP: array(PERS,MACH) of integer  ! Productivity
 end-declarations

 initializations from 'i1assign.dat'
  OUTP
 end-initializations

! **** Exact solution for parallel assignment ****

 declarations
  assign: array(PERS,MACH) of mpvar  ! 1 if person assigned to machine,
                                     ! 0 otherwise
 end-declarations

! Objective: total productivity
 TotalProd:= sum(p in PERS, m in MACH) OUTP(p,m)*assign(p,m)

! One machine per person
  forall(p in PERS) sum(m in MACH) assign(p,m) = 1

! One person per machine
  forall(m in MACH) sum(p in PERS) assign(p,m) = 1

! Solve the problem
 maximize(TotalProd)
 writeln("Parallel machines: ", getobjval)

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

 declarations
  pmin: mpvar                        ! Minimum productivity
 end-declarations

! Calculate minimum productivity
 forall(p in PERS) sum(m in MACH) OUTP(p,m)*assign(p,m) >= pmin

 forall(p in PERS, m in MACH) assign(p,m) is_binary
```

```
! Solve the problem
 maximize(pmin)
 writeln("Serial machines: ", getobjval)

end-model
```

The following procedure `parallel_heur` may be added to the above program. If called after the data has been initialized, it heuristically calculates a (non-optimal) solution to the parallel assignment problem using the intuitive procedure described in the introduction to Section 14.1.

```
procedure parallel_heur
 declarations
  ALLP, ALLM: set of integer         ! Copies of sets PERS and MACH
  HProd: integer                     ! Total productivity value
  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

  HProd+=omax                        ! Add to total productivity
  ALLP-={pmax}; ALLM-={mmax}         ! Remove person and machine from sets

  writeln("  ",pmax, " operates machine ", mmax, " (", omax, ")")
 end-do

 writeln("  Total productivity: ", HProd)
end-procedure
```

14.1.3 Results

The following table summarizes the results found with the different solution methods for the two problems of parallel and serial machines. The very intuitive heuristic method is likely to be employed by people who do not know of Mathematical Programming. However, there is a notable difference between the heuristic and the exact solution to the problem with parallel machines.

Table 14.2: Optimal assignment found

		Person						
	Algorithm	**1**	**2**	**3**	**4**	**5**	**6**	**Productivity**
Parallel Machines	**Heuristic**	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**	6 (29)	3 (30)	5 (34)	2 (26)	1 (28)	4 (27)	26

14.2 Scheduling nurses

Mr. Schedule has been asked to organize the schedule of nurses for the Cardiology service at St. Joseph's hospital. A working day in this service is subdivided into twelve periods of two hours. The personnel requirement changes from period to period: for instance, only a few nurses are required during the night, but the total number must be relatively high during the morning to provide specific care to the patients. The following table lists the personnel requirement for every time period.

Question 1: Determine the minimum number of nurses required to cover all the requirements, knowing that a nurse works eight hours per day and that she is entitled to a break of two hours after she has worked for four hours.

Table 14.3: Personnel requirement per time period

Number	Time interval	Minimum number of nurses
0	00am – 02am	15
1	02am – 04am	15
2	04am – 06am	15
3	06am – 08am	35
4	08am – 10am	40
5	10am – 12pm	40
6	12pm – 02pm	40
7	02pm – 04pm	30
8	04pm – 06pm	31
9	06pm – 08pm	35
10	08pm – 10pm	30
11	10pm – 12am	20

Question 2: The service only has 80 nurses, which is not sufficient with the given requirements. Mr. Schedule therefore proposes that part of the personnel works two additional hours per day. These two additional hours follow immediately after the last four hours, without any break. Determine the schedule of the nurses in this service that minimizes the number of nurses working overtime.

14.2.1 Model formulation for question 1

Let $start_t$ be the number of nurses starting work in time period t (first period = time period from 0am–2am, second period = time period from 2am–4am *etc.*), NT be the number of time periods, and $TIME = \{0, \ldots, NT-1\}$ the set of time periods. The objective function is given by (14.2.1).

$$\text{minimize} \sum_{t \in TIME} start_t \tag{14.2.1}$$

We need to make sure that a sufficiently large number of nurses is available during every time period. For instance, in the period $t = 6$ (12pm–2pm) of the day 40 nurses are required. The number of nurses working during this time period is equal to the total of the number of nurses starting work in this period, the immediately preceding time period, and in the two periods 3 and 4 intervals earlier. That is, during time period 6, the nurses starting in the intervals 2, 3, 5, and 6 are working. For instance, a nurse starting in interval 3 will work in the intervals 3 and 4, have her break in interval 5, and work again in the intervals 6 to 7. If we write REQ_t for the personnel requirement in interval t, we obtain the following constraint for the time period 6:

$$start_2 + start_3 + start_5 + start_6 \geq REQ_6 \tag{14.2.2}$$

Similarly, we can formulate the personnel requirement constraints for all other time periods. If $WORK$ denotes the set of intervals that have an impact on the time period under consideration (that is $WORK = \{0, -1, -3, -4\}$), we may write these constraints in the generic form (14.2.3).

$$\forall t \in TIME : \sum_{i \in WORK} start_{(t+i+NT) \bmod NT} \geq REQ_t \tag{14.2.3}$$

For the first four time periods every day, we need to take into account the nurses who started working in the last four time intervals of the previous day. The expression $(t + 1 + NT) \bmod NT$ transposes the indices correctly, so that for instance for the first interval of a day, the periods 8, 9, 11, and 0 are taken into account by the corresponding constraint ($t = 0$).

The constraints (14.2.4) indicate that all variables $start_t$ must be integer.

$$\forall t \in TIME : start_t \in \mathbb{N} \tag{14.2.4}$$

14.2.2 Implementation of question 1

The following Mosel program is a straightforward implementation of the mathematical model. `WORK` is an example of sets in Mosel that contain negative values and also the value 0.

```
model "I-2 Scheduling nurses 1"
 uses "mmxprs"
```

```
 declarations
  NT = 12                          ! Number of time periods
  TIME = 0..NT-1
  WORK: set of integer             ! Nurses started in other time periods
                                   ! that are working during a period
  REQ: array(TIME) of integer      ! Required number of nurses per time period

  start: array(TIME) of mpvar      ! Nurses starting work in a period
 end-declarations

 initializations from 'i2nurse.dat'
  REQ
 end-initializations

 WORK:= {0, -1, -3, -4}

! Objective: total personnel required
 Total:= sum(t in TIME) start(t)

! Nurses working per time period
 forall(t in TIME) sum(i in WORK) start((t+i+NT) mod NT) >= REQ(t)

 forall(t in TIME) start(t) is_integer

! Solve the problem
 minimize(Total)

end-model
```

14.2.3 Results for question 1

100 nurses are required in total for the Cardiology service. Their working periods may be scheduled as shown in Table 14.4 (there are several equivalent solutions).

Table 14.4: Nurse schedule covering all requirements

Time interval	Starting work	Total
00am – 02am	6	30
02am – 04am	17	40
04am – 06am	12	29
06am – 08am	17	35
08am – 10am	0	40
10am – 12pm	11	40
12pm – 02pm	0	40
02pm – 04pm	13	30
04pm – 06pm	7	31
06pm – 08pm	17	35
08pm – 10pm	0	30
10pm – 12am	0	20

14.2.4 Model formulation for question 2

The optimal solution involves employing 100 nurses. Unfortunately, the service currently only has 80 nurses. They will therefore have to work extra hours.

We retain the variables $start_t$ introduced for the answer to Question 1. In addition, we introduce variables $overt_t$ corresponding to the number of nurses starting to work in interval t and working two overtime hours at the end of their normal period of duty.

Our objective now is to minimize the number of nurses that have to work overtime. We thus obtain the objective function (14.2.5).

$$\text{minimize} \sum_{t \in TIME} overt_t \tag{14.2.5}$$

The number of nurses may not exceed the available total staff of $NUM = 80$. This constraint is expressed

by the relation (14.2.6).

$$\sum_{t \in TIME} start_t \leq NUM \tag{14.2.6}$$

For every time interval t, the number of nurses $overt_t$ starting in this period who work overtime may at most be as large as the total number of nurses $start_t$ starting their work during this time period. We therefore obtain the constraints (14.2.7)

$$\forall t \in TIME : overt_t \leq start_t \tag{14.2.7}$$

As before, the personnel requirement of every time slice needs to be covered. We obtain the constraints (14.2.8) that are similar to the constraints (14.2.3) of Question 1, but in which we also take into account the nurses working extra hours. For example, the nurses on duty in interval 6 started work either in the intervals 2, 3, 5, or 6 as in the first question, or in the interval 1 if they are working overtime.

$$\forall t \in TIME : overt_{(t-5+NT) \bmod NT} + \sum_{i \in WORK} start_{(t+i+NT) mod NT} \geq REQ_t \tag{14.2.8}$$

To complete the model for Question 2, the constraints (14.2.9) and (14.2.10) specify that all variables $start_t$ and $overt_t$ must be integer.

$$\forall t \in TIME : start_t \in \mathbb{N} \tag{14.2.9}$$

$$\forall t \in TIME : overt_t \in \mathbb{N} \tag{14.2.10}$$

14.2.5 Implementation of question 2

The translation of the mathematical model to Mosel is again immediate.

```
model "I-2 Scheduling nurses 2"
 uses "mmxprs"

 declarations
  NT = 12                          ! Number of time periods
  TIME = 0..NT-1
  WORK: set of integer             ! Nurses started in other time periods
                                   ! that are working during a period

  REQ: array(TIME) of integer      ! Required number of nurses per time period
  NUM: integer                     ! Available total staff

  start: array(TIME) of mpvar      ! Nurses starting work in a period
  overt: array(TIME) of mpvar      ! Nurses working overtime
 end-declarations

 initializations from 'i2nurse.dat'
  REQ NUM
 end-initializations

 WORK:= {0, -1, -3, -4}

! Objective: total overtime worked
 TotalOvert:= sum(t in TIME) overt(t)

! Nurses working per time period
 forall(t in TIME)
  overt((t-5+NT) mod NT) + sum(i in WORK) start((t+i+NT) mod NT) >= REQ(t)

! Limit on total number of nurses
 Total <= NUM

 forall(t in TIME) do
  start(t) is_integer
  overt(t) is_integer
  overt(t) <= start(t)
 end-do

! Solve the problem
 minimize(TotalOvert)

end-model
```

14.2.6 Results for question 2

Forty of the eighty available nurses will have to work overtime to satisfy the personnel requirements of the service. Table 14.5 lists a detailed schedule (there are several equivalent solutions).

Table 14.5: Nurse schedule with overtime work

Time interval	Starting work	Total	Overtime
00am – 02am	13	15	0
02am – 04am	1	17	0
04am – 06am	10	16	0
06am – 08am	7	35	3
08am – 10am	17	40	2
10am – 12pm	4	40	8
12pm – 02pm	14	35	0
02pm – 04pm	9	47	0
04pm – 06pm	0	30	0
06pm – 08pm	0	35	17
08pm – 10pm	3	30	4
10pm – 12am	2	20	6

14.3 Establishing a college timetable

Mr. Miller is in charge of establishing the weekly timetable for two classes of the last year in a college. The two classes have the same teachers, except for mathematics and sport. In the college all lessons have a duration of two hours. Furthermore, all students of the same class attend exactly the same courses. From Monday to Friday, the slots for courses are the following: 8:00–10:00, 10:15–12:15, 14:00–16:00, and 16:15–18:15. The following table lists the number of two-hour lessons that every teacher has to teach the students of the two classes per week.

Table 14.6: Number of 2-hour lessons per teacher and class

Teacher	Subject	Lessons for class 1	Lessons for class 2
Mr Cheese	English	1	1
Mrs Insulin	Biology	3	3
Mr Map	History-Geography	2	2
Mr Effofecks	Mathematics	0	4
Mrs Derivate	Mathematics	4	0
Mrs Electron	Physics	3	3
Mr Wise	Philosophy	1	1
Mr Muscle	Sport	1	0
Mrs Biceps	Sport	0	1

The sport lessons have to take place on Thursday afternoon from 14:00 to 16:00. Furthermore, the first time slot on Monday morning is reserved for supervised homework. Mr Effofecks is absent every Monday morning because he teaches some courses at another college. Mrs Insulin does not work on Wednesday. And finally, to prevent students from getting bored, every class may only have one two-hour lesson per subject on a single day. Write a mathematical program that allows Mr Miller to determine the weekly timetable for the two classes.

14.3.1 Model formulation

Let *TEACHERS*, and *CLASS* respectively be the sets of teachers and classes. Let *SLOTS* denote the set of time slots per week, numbered consecutively from 1 to $NP \cdot ND$, where *NP* denotes the number of time slots per day and *ND* the number of days per week. $COURSE_{tc}$ denotes the number of two-hour lessons that the teacher *t* has to teach class *c* every week.

We define variables $teach_{tcl}$ that are 1 if the teacher *t* gives a lesson to class *c* in time slot *l*. We obtain

the following mathematical model:

$$\text{minimize} \sum_{t\in TEACHERS} \sum_{c\in CLASS} \sum_{d=0}^{ND-1} (teach_{t,c,d\cdot NP+1} + teach_{t,c,(d+1)\cdot NP}) \tag{14.3.1}$$

$$\forall t \in TEACHERS, c \in CLASS : \sum_{l\in SLOTS} teach_{tcl} = COURSE_{tc} \tag{14.3.2}$$

$$\forall c \in CLASS, l \in SLOTS : \sum_{t\in TEACHERS} teach_{tcl} \leq 1 \tag{14.3.3}$$

$$\forall t \in TEACHERS, l \in SLOTS : \sum_{c\in CLASS} teach_{tcl} \leq 1 \tag{14.3.4}$$

$$\forall t \in TEACHERS, c \in CLASS, d = 0, \ldots, ND-1 : \sum_{l=d\cdot NP+1}^{(d+1)\cdot NP} teach_{tcl} \leq 1 \tag{14.3.5}$$

$$teach_{MrMuscle,1,15} = 1 \tag{14.3.6}$$

$$teach_{MrsBiceps,2,15} = 1 \tag{14.3.7}$$

$$\forall t \in TEACHERS, c \in CLASS : teach_{tc1} = 0 \tag{14.3.8}$$

$$\forall l = 1, 2 : teach_{MrEffofecks,2,l} = 0 \tag{14.3.9}$$

$$\forall c \in CLASS, l = 2 \cdot NP + 1, \ldots, 3 \cdot NP : teach_{MrsInsuline,c,l} = 0 \tag{14.3.10}$$

$$\forall t \in TEACHERS, c \in CLASS, l \in SLOTS : teach_{tcl} \in \{0, 1\} \tag{14.3.11}$$

The aim of this problem is simply to find a timetable that fulfills all constraints. We may however give ourselves the objective to minimize the **holes** in the timetable. With this aim, the courses should preferably be placed in the time slots from 10:15–12:15 and 14:00–16:00. If these time slices remain unused but lessons are placed either at the beginning or at the end of the day, there will necessarily be holes in the middle of the day. We therefore penalize lessons placed at the beginning or the end of the day, which means we minimize the sum of courses taught during slots 1 and 4 of every day; hence the objective function (14.3.1).

The constraints (14.3.2) indicate that all lessons taught by the teacher t to class c must be scheduled. The constraints (14.3.3) specify that a class has at most one course at any time, and similarly the constraints (14.3.4) that a teacher must not teach more than one course at a time. The constraints (14.3.5) say that at most one two-hour lesson per subject is taught on the same day.

The constraints (14.3.6) to (14.3.10) translate the specific conditions of this problem: the constraints (14.3.6) and (14.3.7) indicate that the sport lessons by Mr Muscle and Mrs Biceps have to take place in the beginning of Thursday afternoon (time slice 15). The first time interval of the week being reserved for supervised homework, the constraints (14.3.8) specify that no course may be scheduled during this period. The constraints (14.3.9) and (14.3.10) respectively prohibit Mr Effofecks teaching on Monday morning and Mrs Insulin teaching on Wednesday (time slices 9-12). The last set of constraints specifies that all variables are binaries.

14.3.2 Implementation

The mathematical model of the previous section is implemented by the following Mosel program.

```
model "I-3 School timetable"
 uses "mmxprs"

 declarations
  TEACHERS: set of string         ! Set of teachers
  CLASS = 1..2                    ! Set of classes
  NP = 4                          ! Number of time periods for courses
  ND = 5                          ! Days per week
  SLOTS=1..NP*ND                  ! Set of time slots for the entire week

  COURSE: array(TEACHERS,CLASS) of integer ! Lessons per teacher and class
 end-declarations

 initializations from 'i3school.dat'
  COURSE
 end-initializations

 finalize(TEACHERS)
```

```
 declarations
  teach: array(TEACHERS,CLASS,SLOTS) of mpvar
                                  ! teach(t,c,l) = 1 if teacher t gives a
                                  ! lesson to class c during time period l
 end-declarations

! Objective: number of "holes" in the class timetables
 Hole:=
  sum(t in TEACHERS, c in CLASS, d in 0..ND-1) (teach(t,c,d*NP+1) +
      teach(t,c,(d+1)*NP))

! Plan all courses
 forall(t in TEACHERS, c in CLASS) sum(l in SLOTS) teach(t,c,l) = COURSE(t,c)

! For every class, one course at a time
 forall(c in CLASS, l in SLOTS) sum(t in TEACHERS) teach(t,c,l) <= 1

! Teacher teaches one course at a time
 forall(t in TEACHERS, l in SLOTS) sum(c in CLASS) teach(t,c,l) <= 1

! Every subject only once per day
 forall(t in TEACHERS, c in CLASS, d in 0..ND-1)
  sum(l in d*NP+1..(d+1)*NP) teach(t,c,l) <= 1

! Sport Thursday afternoon (slot 15)
 teach("Mr Muscle",1,15) = 1
 teach("Mrs Biceps",2,15) = 1

! No course during first period of Monday morning
 forall(t in TEACHERS, c in CLASS) teach(t,c,1) = 0

! No course by Mr Effofecks Monday morning
 forall(l in 1..2) teach("Mr Effofecks",2,l) = 0

! No Biology on Wednesday
 forall(c in CLASS, l in 2*NP+1..3*NP) teach("Mrs Insulin",c,l) = 0

 forall(t in TEACHERS, c in CLASS, l in SLOTS) teach(t,c,l) is_binary

! Solve the problem
 minimize(Hole)

end-model
```

14.3.3 Results

The optimal solution found has the value 10, indicating that 10 courses are scheduled in the slots at the beginning or the end of a day. The resulting timetables for the two classes displayed in Tables 14.7 and 14.8 have no holes. Note that there are many equivalent solutions to this problem.

Table 14.7: Timetable for class 1

	8:00–10:00	10:15–12:15	14:00–16:00	16:15–18:15
Mon	Homework	Maths (Mrs Derivate)	Biology (Mrs Insulin)	Physics (Mrs Electron)
Tue	English (Mr Cheese)	Maths (Mrs Derivate)	Biology (Mrs Insulin)	Physics (Mrs Electron)
Wed	—	Maths (Mrs Derivate)	Hist.-Geogr. (Mr Map)	—
Thu	—	Hist.-Geogr. (Mr Map)	Sport (Mr Muscle)	Physics (Mrs Electron)
Fri	—	Biology (Mrs Insulin)	Maths (Mrs Derivate)	Philosophy (Mr Wise)

Table 14.8: Timetable for class 2

	8:00–10:00	10:15–12:15	14:00–16:00	16:15–18:15
Mon	Homework	Hist.-Geogr. (Mr Map)	Maths (Mr Effofecks)	Biology (Mrs Insulin)
Tue	—	Maths (Mr Effofecks)	Hist.-Geogr. (Mr Map)	Biology (Mrs Insulin)
Wed	—	Physics (Mrs Electron)	Maths (Mr Effofecks)	—
Thu	Physics (Mrs Electron)	English (Mr Cheese)	Sport (Mrs Biceps)	Biology (Mrs Insulin)
Fri	Physics (Mrs Electron)	Philosophy (Mr Wise)	Maths (Mr Effofecks)	—

14.4 Exam scheduling

At a technical university every term the third-year students choose eight modules from the eleven modules that are taught, depending on the option they wish to choose in the fourth year (there are two possible choices: 'Production planning' and 'Quality and security management'). In the current term, certain modules are obligatory for students who wish to continue with one of these options. The obligatory courses are Statistics (S), Graph models and algorithms (GMA), Production management (PM), and Discrete systems and events (DSE). The optional modules are: Data analysis (DA), Numerical analysis (NA), Mathematical programming (MP), C++, Java (J), Logic programming (LP), and Software engineering (SE).

Table 14.9: 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	–

Mrs Edeetee needs to schedule the exams at the end of the term. Every exam lasts two hours. Two days have been reserved for the exams with the following time slices: 8:00–10:00, 10:15–12:15, 14:00–16:00, and 16:15–18:15. For every exam she knows the set of incompatible exams that may not take place at the same time because they have to be taken by the same students. These incompatibilities are summarized in Table 14.4.1.

Help Mrs Edeetee construct a timetable so that no student has more than one exam at a time.

14.4.1 Model formulation

Let $plan_{et}$ be binary variables that are 1 if the exam e is scheduled in time slice t and 0 otherwise. We shall write *EXAM* for the set of exams and *TIME* for the set of time slices. The mathematical model associated with the problem is the following:

$$\forall e \in EXAM : \sum_{t \in TIME} plan_{et} = 1 \tag{14.4.1}$$

$$\forall d, e \in EXAM, d < e \wedge INCOMP_{de} = 1, \forall t \in TIME : plan_{et} + plan_{dt} \leq 1 \tag{14.4.2}$$

$$\forall e \in EXAM, t \in TIME : plan_{et} \in \{0, 1\} \tag{14.4.3}$$

This mathematical model does not contain any objective function since we simply wish to find a solution that satisfies all incompatibility constraints. This does not cause any problem for the solution algorithm that will generate an arbitrary feasible solution (if one exists). We might however add an objective function, for instance by associating costs or preferences with the assignment of certain exams to certain time slices.

The constraints (14.4.1) indicate that every exam needs to be scheduled exactly once. The constraints (14.4.2) require that two incompatible exams may not be scheduled at the same time. The incompatibilities are stored in a Boolean matrix *INCOMP*. An element $INCOMP_{de}$ of this matrix is 1 if the exams d and e are incompatible and 0 otherwise. Since the incompatibility matrix is symmetric for this problem, we only need to define the constraints (14.4.2) for pairs of exams d and e with $d < e$. The last set of constraints (14.4.3) specifies that all variables are binaries.

The problem is completely stated by the constraints (14.4.1) to (14.4.3). However, if this problem has a solution then it has an enormous number of equivalent (symmetric) solutions that are obtained by permutating the time slots assigned to compatible sets of exams. We may try to 'help' the solution algorithm by adding constraints breaking some of these symmetries and hence reducing the size of the search space. For instance, we can assign DA to slot 1, and NA to slot 2 (since it is incompatible with DA). In fact, we can go further as PM is incompatible with DA and NA, so it can be assigned to slot 3. Similarly, GMA, S, and DSE must be given unique slots (4, 5, and 6) since they are incompatible with everything.

How useful these observations are in general is debatable, but breaking symmetry, if possible, is always a good idea.

14.4.2 Implementation

The mathematical model may be implemented with Mosel as shown below. The problem has no objective function but we wish to invoke the optimization algorithm for finding a feasible solution. We therefore simply indicate a constant (dummy) objective in the call to the optimizer — instead of `minimize` we may also use `maximize` since we want to solve the problem without optimizing anything. We have not introduced the symmetry-breaking ideas discussed above.

```
model "I-4 Scheduling exams"
 uses "mmxprs"

 declarations
  EXAM = {"DA","NA","C++","SE","PM","J","GMA","LP","MP","S","DSE"}
                                    ! Set of exams
  TIME = 1..8                       ! Set of time slots

  INCOMP: array(EXAM,EXAM) of integer ! Incompatibility between exams

  plan: array(EXAM,TIME) of mpvar     ! 1 if exam in a time slot, 0 otherwise
 end-declarations

 initializations from 'i4exam.dat'
  INCOMP
 end-initializations

! Schedule all exams
 forall(e in EXAM) sum(t in TIME) plan(e,t) = 1

! Respect incompatibilities
 forall(d,e in EXAM, t in TIME | d<e and INCOMP(d,e)=1)
  plan(e,t) + plan(d,t) <= 1

 forall(e in EXAM, t in TIME) plan(e,t) is_binary

! Solve the problem (no objective)
 minimize(0)

end-model
```

14.4.3 Results

The linear relaxation of this problem gives an integer solution (this is not true in general). In the schedule displayed below only seven out of the eight available time intervals are used. This problem has a large number of different feasible solutions.

Table 14.10: Exam schedule

	8:00–10:00	10:15–12:15	14:00–16:00	16:15–18:15
Day 1	NA, SE, LP, MP	PM	GMA	DA, C++
Day 2	—	S	J	DSE

14.5 Production planning with personnel assignment

The company Line Production decides to plan the production of four of its products (P1, P2, P3, P4) on its five production lines (L1 to L5). The company gains net profits of €7 for the products P1 and P4, €8 for P2, and €9 for P3. The maximum times during which the five production lines may operate are different during the planning period. The maximum capacities for L1 to L5 are 4500 hours, 5000 hours, 4500 hours, 1500 hours, and 2500 hours respectively. Table 14.12 lists the processing time in hours necessary for the production of one unit of every product on every production line. Which quantities of P1 to P4 should be produced to maximize the total profit?

If subsequently a transfer of personnel (and hence of working hours) is authorized between production

Table 14.11: Unitary processing times

	Lines				
Products	**L1**	**L2**	**L3**	**L4**	**L5**
P1	1.3	0.9	2.0	0.3	0.9
P2	1.8	1.7	1.4	0.6	1.1
P3	1.3	1.2	1.3	1.0	1.4
P4	0.9	1.1	1.0	0.9	1.0

Table 14.12: Possible transfers of personnel

	Destination					
Origin	**L1**	**L2**	**L3**	**L4**	**L5**	**Maximum number of transferable hours**
L1	–	yes	yes	yes	no	400
L2	no	–	yes	no	yes	800
L3	yes	yes	–	yes	no	500
L4	no	no	no	–	yes	200
L5	yes	yes	yes	no	–	300

lines during the planning period as shown in Table 14.12, which is the maximum profit? How many hours are transfered and under what conditions?

14.5.1 Model formulation

Let *PRODS* be the set of products and *LINES* the set of production lines. The per unit profit of product p is given as $PROFIT_p$ and the processing duration of product p on line l as DUR_{pl}. The capacity of every production line l in working hours is CAP_l. As usual in production planning problems, a variables $produce_p$ indicates the quantity of product p that is produced. The model corresponding to the first question is fairly simple. The objective is to maximize the total profit (14.5.1) whilst remaining within the capacity limits of every production line (constraints (14.5.2)). We obtain the following model:

$$\text{maximize} \sum_{p \in PRODS} PROFIT_p \cdot produce_p \tag{14.5.1}$$

$$\forall l \in LINES : \sum_{p \in PRODS} DUR_{pl} \cdot produce_p \leq CAP_l \tag{14.5.2}$$

$$\forall p \in PRODS : produce_p \geq 0 \tag{14.5.3}$$

If it is now possible to transfer personnel from one line to another, variables $hours_l$ are required that correspond to the working hours used on production line l. We also introduce variables $transfer_{lk}$ that represent the number of hours transfered from line l to line k. The constraints (14.5.5) that replace the constraints (14.5.2) of the previous model specify that the number of hours worked is equal to the production multiplied by the per unit duration of the work.

The constraints (14.5.7) establish the balance between the working hours carried out on a line, the hours transfered and the maximum number of working hours on this line. On one line, $hours_l$ hours are worked that correspond to the available number of hours for this line, increased by the hours that are transfered to this line from other and decreased by the hours transfered to other lines ($TRANSF_{lk}$ is 1 if it is possible to transfer hours from line l to line k, 0 otherwise). The constraints (14.5.8) establish the limits $TMAX_l$ on the number of hours that may be transfered from a line to the others. The last two sets of constraints (14.5.9) and (14.5.10) are the non-negativity conditions for variables $transfer_{kl}$ and $hours_l$.

$$\text{maximize} \sum_{p \in PRODS} PROFIT_p \cdot produce_p \tag{14.5.4}$$

$$\forall l \in LINES : \sum_{p \in PRODS} DUR_{pl} \cdot produce_p \leq hours_l \tag{14.5.5}$$

$$\forall p \in PRODS : produce_p \geq 0 \tag{14.5.6}$$

$$\forall l \in LINES : hours_l = CAP_l + \sum_{\substack{k \in LINES \\ TRANSF_{kl}=1}} transfer_{kl} - \sum_{\substack{k \in LINES \\ TRANSF_{lk}=1}} transfer_{lk} \tag{14.5.7}$$

$$\forall l \in LINES : \sum_{\substack{k \in LINES \\ TRANSF_{lk}=1}} transfer_{lk} \leq TMAX_l \tag{14.5.8}$$

$$\forall l, k \in LINES : transfer_{lk} \geq 0 \quad (14.5.9)$$
$$\forall l \in LINES : hours_l \geq 0 \quad (14.5.10)$$

14.5.2 Implementation

The following Mosel program implements and solves the models for both questions. First the model for question 1 is defined and solved. Then follow the additional declarations of data and variables required by the second model. For the second model, the capacity constraints `Load` of the first model (14.5.2) are re-defined with the modified version required by the second model (14.5.5).

```
model "I-5 Production planning with personnel"
 uses "mmxprs"

 declarations
  PRODS = 1..4                        ! Set of products
  LINES = 1..5                        ! Set of production lines

  PROFIT: array(PRODS) of integer     ! Profit per product
  DUR: array(PRODS,LINES) of real     ! Duration of production per line
  CAP: array(LINES) of integer        ! Working hours available per line

  Load: array(LINES) of linctr        ! Workload constraints
  produce: array(PRODS) of mpvar      ! Quantity produced
 end-declarations

 initializations from 'i5pplan.dat'
  PROFIT DUR CAP
 end-initializations

! Objective: Total profit
 Profit:= sum(p in PRODS) PROFIT(p)*produce(p)

! Capacity constraints on lines
 forall(l in LINES) Load(l):=sum(p in PRODS) DUR(p,l)*produce(p) <= CAP(l)

! Solve the problem
 maximize(Profit)
 writeln("Total profit: ", getobjval)

! **** Allow transfer of working hours between lines ****

 declarations
  TRANSF: dynamic array(LINES,LINES) of integer ! 1 if transfer is allowed,
                                                ! 0 otherwise
  TMAX: array(LINES) of integer         ! Maximum no. of hours to transfer

  hours: array(LINES) of mpvar          ! Initial working hours per line
  transfer: dynamic array(LINES,LINES) of mpvar ! Hours transferred
 end-declarations

 initializations from 'i5pplan.dat'
  TRANSF TMAX
 end-initializations

 forall(k,l in LINES | exists(TRANSF(k,l))) create(transfer(k,l))

! Re-define capacity constraints on lines
 forall(l in LINES) Load(l):=sum(p in PRODS) DUR(p,l)*produce(p) <= hours(l)

! Balance constraints
 forall(l in LINES)
  hours(l) = CAP(l) + sum(k in LINES) transfer(k,l) -
                      sum(k in LINES) transfer(l,k)

! Limit on transfer
 forall(l in LINES) sum(k in LINES) transfer(l,k) <= TMAX(l)

! Solve the problem
 maximize(Profit)
 writeln("Total profit allowing transfer: ", getobjval)

end-model
```

Note that the variables `transfer` in the second model are defined as a dynamic array, and only after the data indicating the allowable transfers (array `TRANSF`) is known do we create the `transfer` variables actually required. By proceeding in this way, we may remove the tests for the values of the entries of `TRANSF` from the various constraints since Mosel will automatically sum only those `transfer` variables that have been created.

14.5.3 Results

If no transfer of personnel (that is, working hours) between production lines is possible, the maximum profit is €18,883. 1542.55 units of product P1 and 1010.64 units of P2 are produced, and nothing of the other two products. When examining the capacity constraints for the production lines, we find that only line 3 and 5 work at their maximum capacity. This indicates that a transfer of working hours may be profitable.

When the transfer of working hours between lines is allowed, the total profit increases to €23,431.10. The quantities to produce are now 702.35 units of P1, 942.82 of P2, 554.83 of P3, and 854.84 of P4. On production line 1 a total of 4100 hours are used and 400 hours are transfered to line 4. Production line 2 works for 3840.3 hours and 800 hours are transfered to line 5. Line 3 works during 4300 hours and 200 hours are transfered to line 4. On line 4, 600 hours are added to the original capacity of 1500 hours and for L5 the 800 hours transfered from line 2 result in a total working time of 3300 hours.

14.6 Planning the personnel at a construction site

Construction workers who erect the metal skeleton of skyscrapers are called steel erectors. The following table lists the requirements for steel erectors at a construction site during a period of six months. Transfers from other sites to this one are possible on the first day of every month and cost $100 per person. At the end of every month workers may leave to other sites at a transfer cost of $160 per person. It is estimated that understaffing as well as overstaffing cost $200 per month per post (in the case of unoccupied posts the missing hours have to be filled through overtime work).

Table 14.13: Monthly requirement for steel erectors

March	April	May	June	July	August
4	6	7	4	6	2

Overtime work is limited to 25% of the hours worked normally. Every month, at most three workers may arrive at the site. The departure to other sites is limited by agreements with labor unions to 1/3 of the total personnel of the month. We suppose that three steel erectors are already present on site at the end of February, that nobody leaves at the end of February and that three workers need to remain on-site at the end of August. Which are the number of arrivals and departures every month to minimize the total cost?

14.6.1 Model formulation

Let *MONTHS* be the set of months in the time period, numbered consecutively from *FIRST* to *LAST*. The different basic costs are denoted as follows: *CARR* and *CLEAVE* the cost of a person arriving and leaving respectively, *COVER* and *CUNDER* the cost of over- and understaffing per person. *NSTART* and *NFINAL* respectively are the number of workers on-site at the beginning and end of the planning period. For every month m the personnel requirement REQ_m is given.

Although it would be possible to formulate this model with fewer variables, we describe a formulation that uses five types of variables indexed by the months to facilitate the modeling and provide all the details for the recruitment plan. In every month m, $onsite_m$ steel erectors are present (between the arrivals at the beginning of the month and the departures at its end). There are $arrive_m$ arrivals, $leave_m$ workers leaving, $over_m$ persons more than the required personnel and $under_m$ persons missing (14.6.1).

$$\forall m \in MONTHS : onsite_m, arrive_m, leave_m, over_m, under_m \in \mathbb{N} \tag{14.6.1}$$

For the first month *FIRST* the total staff equals the initial staff plus the arrivals at the beginning of the month (14.6.2). After the last month *LAST*, the remaining personnel onsite equals the personnel present in the last month minus the departures at the end of this month (14.6.3).

$$onsite_{FIRST} = NSTART + arrive_{FIRST} \tag{14.6.2}$$

$$NFINAL = onsite_{LAST} - leave_{LAST} \tag{14.6.3}$$

For the intermediate months, the personnel on-site, the arrivals and departures are linked by a relation comparable to the stock balance constraints used for the production planning problems in Chapter 8 (14.6.4).

$$\forall m \in \{FIRST + 1, \ldots, LAST - 1\} : onsite_m = onsite_{m-1} - leave_{m-1} + arrive_m \quad (14.6.4)$$

The number of workers present on-site may be different from the number of personnel actually required. But it becomes equal to the requirement if one deducts the overstaffing and adds the missing persons (14.6.5).

$$\forall m \in MONTHS : onsite_m - over_m + under_m = REQ_m \quad (14.6.5)$$

For every month m, the constraints (14.6.6) formulate the limit on the hours worked overtime, the constraints (14.6.7) limit the number of arrivals and the constraints (14.6.8) the number of persons leaving the construction site.

$$\forall m \in MONTHS : over_m \leq onsite_m \ / \ 4 \quad (14.6.6)$$

$$\forall m \in MONTHS : arrive_m \leq 3 \quad (14.6.7)$$

$$\forall m \in MONTHS : leave_m \leq onsite_m \ / \ 3 \quad (14.6.8)$$

The objective function (14.6.9) accumulates the costs of arrivals, departures, over- and understaffing. Due to the minimization of the cost, there will never be over- and understaffing at the same time in any month.

$$\text{minimize} \sum_{m \in MONTHS} (CARR \cdot arrive_m + CLEAVE \cdot leave_m + COVER \cdot over_m + CUNDER \cdot under_m) \quad (14.6.9)$$

14.6.2 Implementation

In the following Mosel implementation of the mathematical model, the balance constraints (14.6.2) – (14.6.4) are grouped into a single constraint expression using the inline `if`. Note further the index set `MONTHS` corresponding to the order numbers of the months (and not starting with 1) used for all arrays.

```
model "I-6 Construction site personnel"
 uses "mmxprs"

 declarations
  FIRST = 3; LAST = 8
  MONTHS = FIRST..LAST                ! Set of time periods (months)

  CARR, CLEAVE: integer               ! Cost per arrival/departure
  COVER, CUNDER: integer              ! Cost of over-/understaffing
  NSTART, NFINAL: integer             ! No. of workers at begin/end of plan
  REQ: array(MONTHS) of integer       ! Requirement of workers per month

  onsite: array(MONTHS) of mpvar      ! Workers on site
  arrive,leave: array(MONTHS) of mpvar ! Workers arriving/leaving
  over,under: array(MONTHS) of mpvar  ! Over-/understaffing
 end-declarations

 initializations from 'i6build.dat'
  CARR CLEAVE COVER CUNDER NSTART NFINAL REQ
 end-initializations

! Objective: total cost
 Cost:= sum(m in MONTHS) (CARR*arrive(m) + CLEAVE*leave(m) +
                          COVER*over(m) + CUNDER*under(m))

! Satisfy monthly need of workers
 forall(m in MONTHS) onsite(m) - over(m) + under(m) = REQ(m)

! Balances
 forall(m in MONTHS)
  onsite(m) = if(m>FIRST, onsite(m-1) - leave(m-1), NSTART) + arrive(m)
 NFINAL = onsite(LAST) - leave(LAST)

! Limits on departures, understaffing, arrivals; integrality constraints
 forall(m in MONTHS) do
  leave(m) <= 1/3*onsite(m)
  under(m) <= 1/4*onsite(m)
  arrive(m) <= 3
```

```
  arrive(m) is_integer; leave(m) is_integer; onsite(m) is_integer
  under(m) is_integer; over(m) is_integer
 end-do

! Solve the problem
 minimize(Cost)

end-model
```

14.6.3 Results

The optimal solution to this problem has a cost of $1,780. The following table contains the detailed recruitment plan (there are several equivalent solutions).

Table 14.14: Optimal plan for the construction site personnel

Month	Initial	March	April	May	June	July	August	Final
Requirement	–	4	6	7	4	6	2	–
On site	3	4	6	6	6	6	4	3
Arrive	0	1	2	0	0	0	0	0
Leave	0	0	0	0	0	2	1	0
Overstaffing	0	0	0	0	2	0	2	0
Understaffing	0	0	0	1	0	0	0	0

14.7 References and further material

The assignment problem seen for the allocation of personnel to machines applies to other situations, as for example the recruitment of candidates for jobs according to their preferences and the perhaps more surprising problem of flight connections in Chapter 11. Since the simplex algorithm automatically finds integer-valued solutions, cases of considerable size (100×100 matrices for example) are easily dealt with.

However, even faster specialized algorithms exist, like the Hungarian algorithm in $O(n^3)$ described by Papadimitriou [PS98] or the techniques for finding the maximum flow presented by Ahuja et al. [AMO93] and Prins [Pri94a]. For serial workstations, the Mathematical Programming approach works less well, but fast algorithms based on graphs also exist for this case [DZ78].

The nurse scheduling problem in Section 14.2 is a simplified version of personnel planning problems in the hospital environment. A more complex problem for determining an individual plan for every nurse and taking into account the rules imposed by law and the preferences of the nurses, whilst minimizing the cost of the personnel is studied by Jaumard et al. [JSV98]. The authors suggest a column generation technique as the solution method. Personnel scheduling problems of this and similar types are among the most successful application areas of Constraint Programming (for instance the application described in [HC99]). Typically the aim is 'simply' to generate a feasible plan, without any optimization. For nurse scheduling, specialized CP software is available ([CHW98]).

The problems in Section 14.3 and 14.4 are classical timetabling problems frequently encountered in secondary schools, colleges and universities. They have in general many, sometimes difficult, constraints that have not been taken into account here (assignment of class rooms, courses or exams of different durations, balanced distribution of courses or exams over the days of the week). These problems are NP-hard and usually solved by heuristics, metaheuristics or Constraint Programming techniques. Carter and Laporte [CL96] give an overview of the approaches used to solve this type of problem. In [Tri84] the author suggests a model formulation with Mathematical Programming. Graph coloring models are used by de Werra [DW97]. Boufflet and Nègre describe in [BN96] a tabu search method for planning the exams at the Technical University of Compiègne. In Boizumault [BDGP95] a Constraint Programming approach for solving a timetabling problem and the problem of scheduling exams at the Institute of Applied Mathematics of Angers may be found.

Section 14.5 presents a particular application of personnel management. In the production planning problems studied in Chapter 8, certain aspects of resource management are often ignored or deliberately not taken into account. The notion of personnel/human resource management in the form of working hours transfered between production lines is nevertheless important and must not be forgotten.

The personnel planning problem in Section 14.6 has been studied by Clark and Hastings [CH77] who solve it with a dynamic programming method. It resembles the production planning problems in Chapter 8. It may be adapted to problems that consist of filling time slices by periods of work like contracts of limited

duration, shift planning, *etc.* It also has certain similarities with the problem of planning a fleet of vans in Chapter 10.

Chapter 15
Local authorities and public services

The military was the first to be interested in Operations Research, followed by industry. In the most developed countries, the public sector uses these techniques more and more to provide the best possible service to consumers for a given budget.

The provision of drinking water will be a major problem for humanity in the 21st century. It will become necessary to find new sources of water and to enlarge the old sewage systems and networks of pipes that might create unforeseen bottlenecks. In the classical maximum flow problem of Section 15.1 we show how to calculate the maximum throughput of a water transport network. The methods may be applied to other liquids, and also to road or telecommunications networks.

Section 15.2 is a study of closed circuit TV surveillance of streets. The problem consist of covering all the streets with a minimum number of pivoting cameras. A political application, still quite relevant in certain countries, is presented in Section 15.3: a party wishes to re-organize the electoral districts to maximize its number of seats at the next elections. More seriously, this type of problem occurs when one tries to establish a fair subdivision into geographical sectors.

The problem in Section 15.4 is gritting the streets of a village in case of ice, with a truck that drives a tour of minimum total length. As opposed to the delivery of heating oil in Section 10.4, the tasks to be executed are not placed on the nodes of the network but on its arcs. Section 15.5 deals with a location problem for public services, in this case of income tax offices. The optimization criterion employed is the average distance that an inhabitant of the area has to travel to the closest income tax office. The chapter terminates with the evaluation of the performances of four hospitals using the DEA method.

15.1 Water conveyance / water supply management

The graph in Figure 15.1 shows a water transport network. The nodes, numbered from 1 to 10, represent the cities, the reservoirs, and the pumping stations connected by a network of pipes. The three cities Gotham City, Metropolis, and Spider Ville are supplied from two reservoirs. The availabilities of water from these reservoirs in thousands of m^3/h are 35 for reservoir 1 and 25 for reservoir 2. The capacity of each pipe connection is indicated in thousands of m^3/h in the graph.

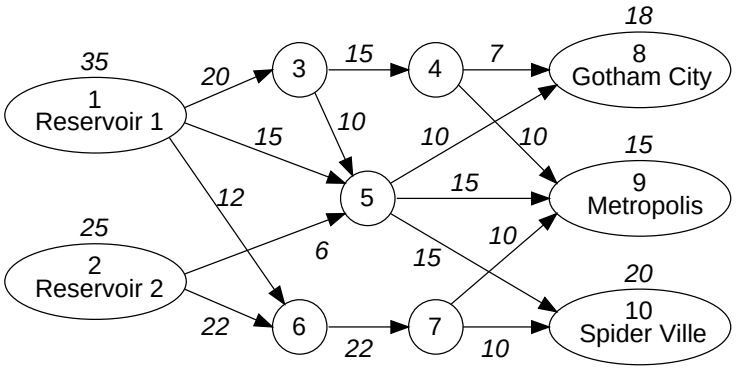

Figure 15.1: Water transport network

A study is undertaken to find out whether the existing network will be able to satisfy the demands of the

cities in ten years time, that is 18, 15, and 20 thousand m^3/h. Determine the maximum flow in the current network. Will it be sufficient in ten years from now?

15.1.1 Model formulation

The problem consists of finding a flow through the given graph that best satisfies the needs of the three cities (nodes 8, 9, and 10) taking into account the availabilities at the reservoirs 1 and 2. This problem is a classical maximum flow problem.

We start by transforming the graph by creating:

- a first fictitious node called *SOURCE* (node 11 in Figure 15.2) connected to the reservoirs by two arcs (11,1) and (11,2) with capacities corresponding to the availability of water from the two reservoirs (35 and 25 thousand m^3/h);
- a second fictitious node called *SINK* (node 12 in Figure 15.2) to which the three cities are connected by three arcs (8,12), (9,12), and (10,12) with capacities corresponding to the cities' requirement for water (18, 15, and 20 thousand m^3/h).

The resulting graph (Figure 15.2) is a **transport network** G = (*NODES, ARCS, CAP, SOURCE, SINK*) in which:

- *NODES* is the set of nodes;
- *ARCS* is the set of arcs;
- CAP_{nm} denotes the capacity of the arc $a = (n, m)$;
- *SOURCE* is the source (node 11);
- *SINK* is the sink (node 12).

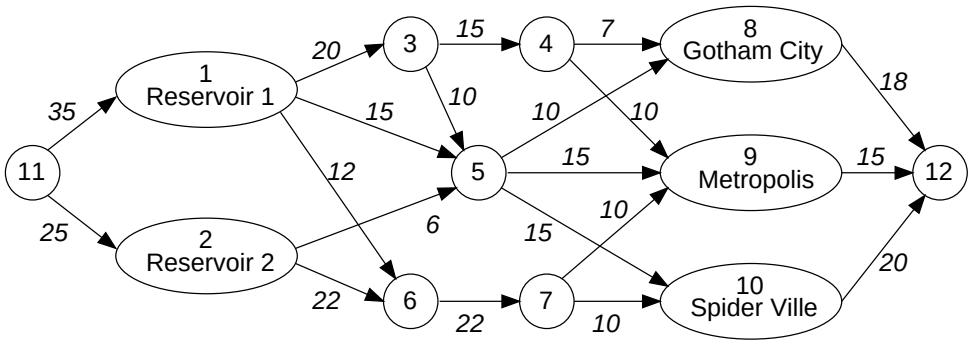

Figure 15.2: Water transport network after addition of source and sink

$SUCC_n$ denotes the set of immediate successors of a node n and $PRED_n$ the set of its immediate predecessors. A **flow** of a total throughput *TotalFlow* in this transport network fulfills the following constraints.

$$\forall (n, m) \in ARCS : flow_{nm} \leq CAP_{nm} \tag{15.1.1}$$

$$\forall n \neq SOURCE, SINK : \sum_{m \in SUCC_n} flow_{nm} = \sum_{m \in PRED_n} flow_{mn} \tag{15.1.2}$$

$$TotalFlow = \sum_{n \in PRED_{SINK}} flow_{n,SINK} \tag{15.1.3}$$

$$\forall (n, m) \in ARCS : flow_{nm} \geq 0 \tag{15.1.4}$$

The constraints (15.1.1) indicate that the flow $flow_{nm}$ on every arc (n, m) must not exceed the capacity CAP_{nm} of this arc. The constraints (15.1.2) specify that the total flow arriving at any node n also has to leave this node (with the exception of the source and sink nodes). This condition is called **Kirchhoff's law**. The constraint (15.1.3) indicates that the total flow *TotalFlow* in the network equals the flow arriving at the sink (it is also equal to the flow leaving the source). Finally, the non-negativity constraints for the variables are given in (15.1.4). The problem that we need to solve corresponds to searching for the maximum flow between the nodes *SOURCE* and *SINK*, that is, a flow maximizing *TotalFlow*. Hence the very simple objective function (15.1.5):

$$\text{maximize } TotalFlow \tag{15.1.5}$$

It is quite obviously possible to let non-integer flows pass through the arcs. However, in this type of problem, the simplex algorithm automatically finds integer solution values in the optimal solution to the linear problem if all capacities are integer.

15.1.2 Implementation

In the Mosel implementation of this problem we represent the set of arcs as a list *PIPE* that defines every arc $a \in ARCS$ in the form $a = (PIPE_{a1}, PIPE_{a2})$. For efficiency reasons, the arrays *CAP* and *flow* are indexed by the sequence number *a* of the arc, and not with the double index (*n, m*) indicating the two nodes connected by the arc *a* as in the mathematical formulation. This indexation allows us to define these arrays exactly with the required size.

```
model "J-1 Water supply"
 uses "mmxprs"

 declarations
  ARCS: range                        ! Set of arcs
  NODES=1..12

  PIPE: array(ARCS,1..2) of integer ! Definition of arcs (= pipes)
  CAP: array(ARCS) of integer       ! Capacity of arcs
  SOURCE,SINK: integer              ! Number of source and sink nodes
 end-declarations

 initializations from 'j1water.dat'
  PIPE CAP SOURCE SINK
 end-initializations

 finalize(ARCS)

 declarations
  flow: array(ARCS) of mpvar       ! Flow on arcs
 end-declarations

! Objective: total flow
 TotalFlow:= sum(a in ARCS | PIPE(a,2)=SINK) flow(a)

! Flow balances in nodes
 forall(n in NODES | n<>SOURCE and n<>SINK)
  sum(a in ARCS | PIPE(a,1)=n) flow(a) = sum(a in ARCS | PIPE(a,2)=n) flow(a)

! Capacity limits
 forall(a in ARCS) flow(a) <= CAP(a)

! Solve the problem
 maximize(TotalFlow)

end-model
```

15.1.3 Results

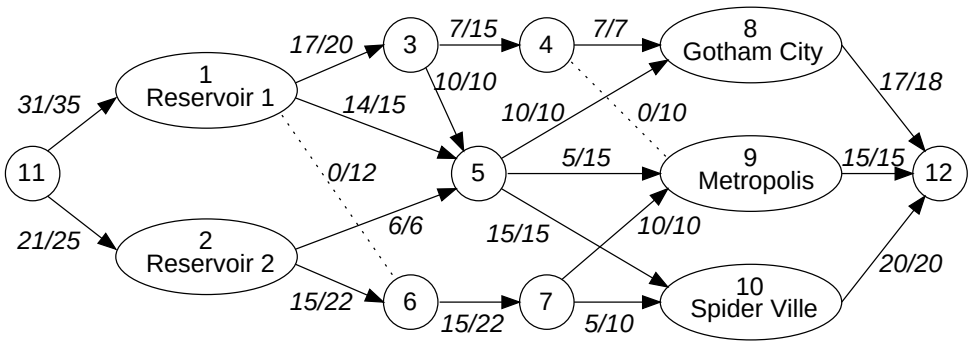

Figure 15.3: Maximum flow

The maximum flow that may pass through the network is 52,000 m^3 per hour. Figure 15.3 shows an

example how this flow may spread through the network (there are other equivalent solutions). In this figure, every arc is labeled with the flow passing through the arc followed by its capacity (both in m^3/h). for instance, the flow on the arc (1,3) is 17 and its capacity is 20.

The total flow is not sufficient to satisfy all the demands in ten years time: with the given network layout and capacities, Gotham City will only receive 17,000 m^3 of the required 18,000 m^3.

15.2 CCTV surveillance

In the course of the last few months, the industrial zone of Billston has suffered from a series of break-ins and thefts during the night. The zone is watched by security men but there are too few of them. The town council in charge of security in this zone decides to install surveillance cameras to aid the security men with their task. These cameras can be directed and pivot through 360°. By installing a camera at the intersection of several streets, it is possible to survey all adjoining streets. The map in Figure 15.4 shows the industrial zone with the limits of the zone to be covered by closed circuit TV (CCTV) surveillance and the 49 possible locations where to install the cameras. What is the minimum number of cameras that have to be installed to survey all the streets of this zone and where should they be placed?

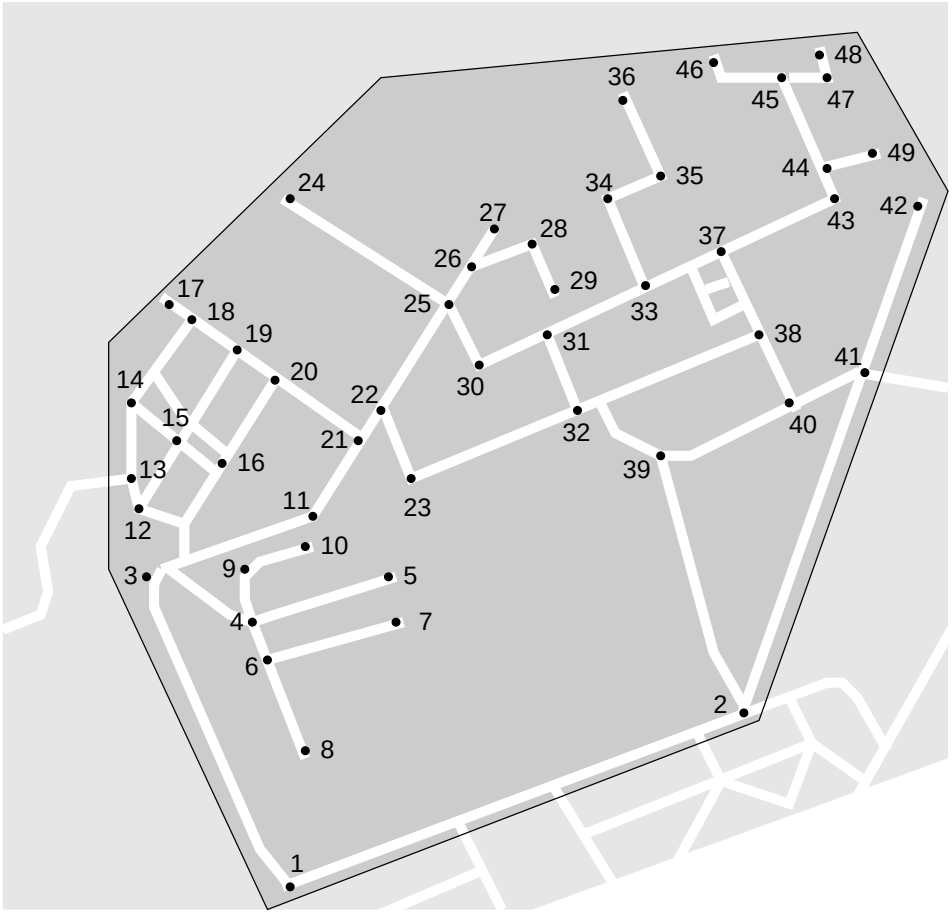

Figure 15.4: Industrial zone in Billston

15.2.1 Model formulation

The most fastidious task in this problem is the encoding of the map of the zone, but this step is necessary for any computerized solving of this problem. The set of possible locations of cameras (mostly street intersections) is denoted by *NODES*. The street network is modeled through a graph *G* = (*NODES, STREETS*), where the set *STREETS* corresponds to the links (streets) between the possible locations *NODES* of the cameras.

We define binary variables $place_n$ that are 1 if a camera is put up at location *n* and 0 otherwise. The

unique set of constraints (15.2.2) indicates that every street needs to be surveyed by at least one camera. Therefore, if a street exists between the two locations n and m, we need to have a camera in n ($place_n = 1$) or m ($place_m = 1$) or in both places. It is possible to cover a street by two cameras and this may pay in certain cases: in Figure 15.4 two cameras in locations 4 and 6 seem to be too much for the street but they also cover the three cul-de-sacs leading to 5, 7, and 8. The objective function (15.2.1) minimizes the total number of cameras to install. Once the graph is defined, we obtain a remarkably simple 0-1 integer program.

$$\text{minimize} \sum_{n \in NODES} place_n \tag{15.2.1}$$

$$\forall (n, m) \in STREETS : place_n + place_m \geq 1 \tag{15.2.2}$$

$$\forall n \in NODES : place_n \in \{0, 1\} \tag{15.2.3}$$

15.2.2 Implementation

The undirected graph is encoded by a symmetric adjacency matrix $STREET$. $STREET_{nm} = 1$ if the street (n, m) exists and 0 otherwise. This matrix is read from the data file in sparse format, that is, only the entries that are 1 are defined. To reduce the amount of data in the file, only the arcs (n, m) with $n < m$ are given. The other half of the adjacency matrix is calculated in the Mosel program.

```
model "J-2 Surveillance"
 uses "mmxprs"

 declarations
  NODES=1..49
  STREET: dynamic array(NODES,NODES) of integer  ! 1 if a street connects
                                                 ! two nodes, 0 otherwise

  place: array(NODES) of mpvar     ! 1 if camera at node, 0 otherwise
 end-declarations

 initializations from 'j2bigbro.dat'
  STREET
 end-initializations

 forall(n,m in NODES | exists(STREET(n,m)) and n<m )
  STREET(m,n):= STREET(n,m)

! Objective: number of cameras to install
 Total:= sum(n in NODES) place(n)

! Flow balances in nodes
 forall(n,m in NODES | exists(STREET(n,m)) ) place(n)+place(m) >= 1

 forall(n in NODES) place(n) is_binary

! Solve the problem
 minimize(Total)

end-model
```

15.2.3 Results

The optimization algorithm finds an integer solution with 24 cameras, the locations of which are marked with circles in Figure 15.5 (there are several equivalent solutions to this problem).

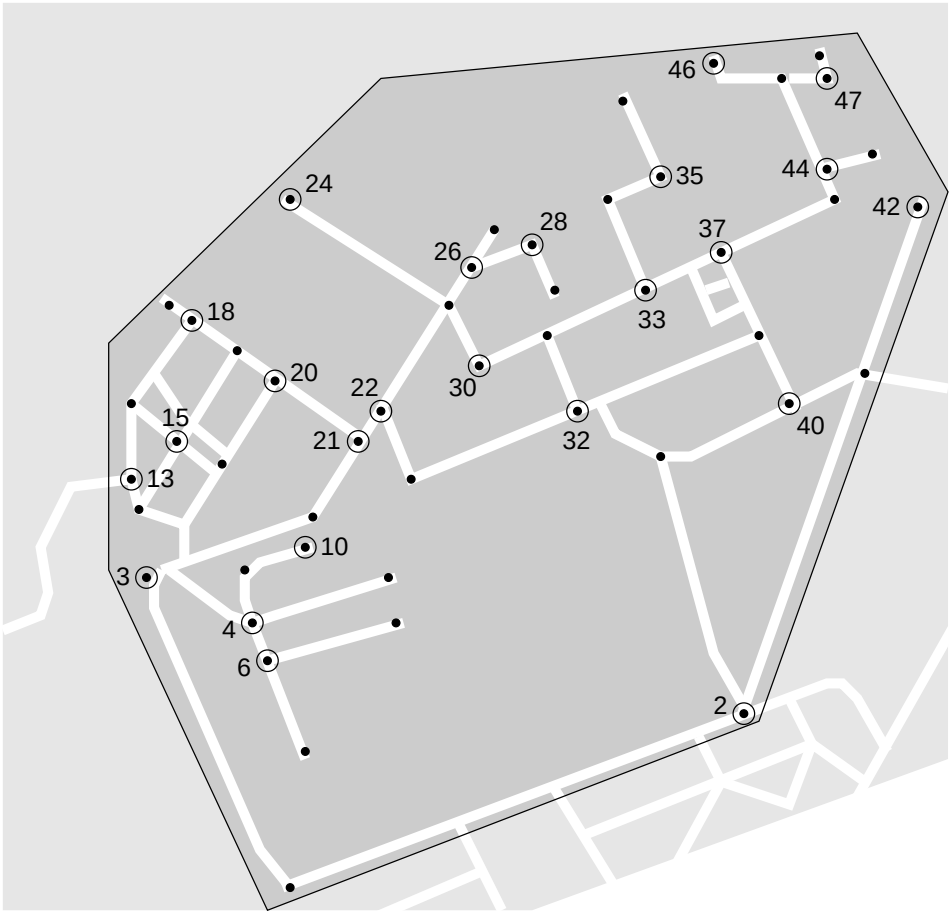

Figure 15.5: Location of cameras

15.3 Rigging elections

In a country far away, the party of the duke Sark Mevo has finally beaten the ruling coalition headed by the princess Reguel Tekris. Mevo wishes to consolidate his position in the capital, the fourteen quarters of which need to be grouped to electoral districts. A schematic map of the capital is given in Figure 15.6. The quarters are numbered from 1 to 14 (bold print). The two other numbers are the forecast number of favorable votes for Mevo and the total number of electors per quarter. All electors must vote and the winner needs to have the absolute majority. A valid electoral district is formed by several adjacent quarters and must have between 30,000 and 100,000 voters. Two quarters that touch each other just at a point like 12 and 13 are not considered adjacent. Electoral districts consisting of a single quarter are permitted if it has at least 50,000 voters. Nevertheless, Mevo may not decently define an electoral district solely with quarter 10, since this contains his residence.

Determine a partitioning into five electoral districts that maximizes the number of seats for Mevo. Should this cause any difficulties, try a partitioning into six districts. Snirp, the mathematical jester, suggests Mevo uses Mathematical Programming...

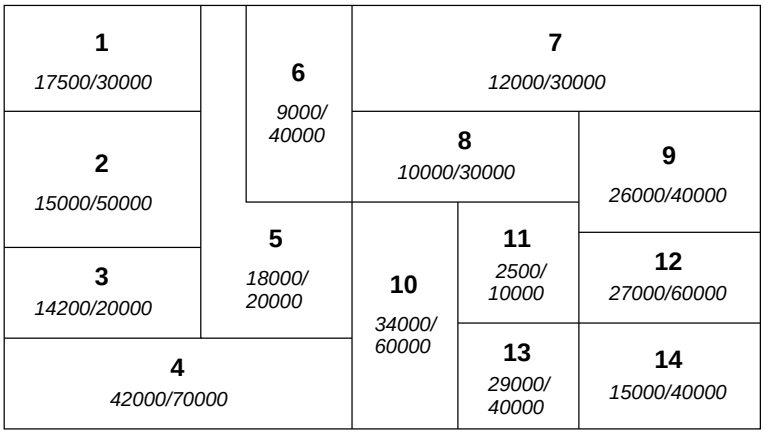

Figure 15.6: Map of the capital and its quarters. Legend: **quarter number**, *supporters/electorate*

15.3.1 Model formulation

The problem we are concerned with here is a **partitioning problem**: given the set of all possible electoral districts, we have to choose a subset such that every quarter appears in a single chosen electoral district.

Let *QUARTERS* be the set of all quarters of the capital, *MINSINGLE* the minimum size for an electoral district consisting of a single quarter and *MINPOP* and *MAXPOP* respectively the minimum and maximum sizes for districts formed from several quarters. The number of electors per quarter is given in *POP*. We may then calculate the complete set of electoral districts using the following algorithm:

- forall q in *QUARTERS*:
  - if (electorate of $q \geq$ *MINSINGLE* and $q \neq 10$): add q to the list of districts
  - forall neighbors p of q:
    if electorate of $(p + q) \leq$ *MAXPOP*: call *add_neighbor*(p, $\{q\}$)
- procedure *add_neighbor*(*toadd*, *set*)
  - add *toadd* to *set*
  - if electorate of *set* $\geq$ *MINPOP*: add *set* to the list of districts
  - forall neighbors p of *toadd*:
    if electorate of $(set \cup \{q\}) \leq$ *MAXPOP*: call *add_neighbor*(p, *set*)

If the set of neighbors of every quarter is defined in such a way that it only contains those adjacent quarters with sequence numbers larger than the quarter itself, the algorithm calculates every possible electoral district exactly once.

In the following we shall assume that the set *RDIST* of possible districts (there are 48 different ones in this example) has been saved in the form of an array *DISTR*. An entry $DISTR_{dq}$ of this array has the value

1 if quarter q is contained in district d and 0 otherwise. Based on the given forecasts of favorable votes per quarter, we can calculate the majority indicator MAJ_d for every district d. MAJ_d is 1 if the sum of favorable votes is at least 50% of the electorate of all quarters in this district (that is, Mevo has the absolute majority), and 0 otherwise.

The formulation of this problem requires a binary variable $choose_d$ per possible district d that takes the value 1 if the district is chosen for the partitioning (15.3.3). The constraint (15.3.2) means that exactly $REQD$ districts are formed, where $REQD$ indicates the required number of districts. The objective function (15.3.1) consists of maximizing the number of seats in the chosen partitioning.

$$\text{maximize} \sum_{d \in RDIST} MAJ_d \cdot choose_d \tag{15.3.1}$$

$$\sum_{d \in RDIST} choose_d = REQD \tag{15.3.2}$$

$$\forall d \in RDIST : choose_d \in \{0, 1\} \tag{15.3.3}$$

Every quarter has to appear in exactly one district of the chosen partitioning. This constraint is expressed by (15.3.4).

$$\forall q \in QUARTERS : \sum_{d \in RDIST} DISTR_{dq} \cdot choose_d = 1 \tag{15.3.4}$$

After the corresponding data preprocessing, the mathematical model thus has a very simple form. It has the particularity of an entirely binary coefficient matrix and right hand side (constant terms).

15.3.2 Implementation

For clarity's sake, the Mosel implementation of this problem is split into two files separating the data preprocessing from the model itself. In the following, we first list the code implementing the mathematical model. At the beginning of this model, after the declaration of the data arrays, we include the file `j3elect_calc.mos` that generates the data in the form required for the model. Any code included into a Mosel program using the `include` statement is treated as if it were printed at this place in the program itself. In particular, the included code must **not** contain `model "..."` and `end-model`. The required number of districts $REQD$ is defined as a parameter so as to enable the user to run this model for different values of $REQD$ without having to modify the code.

```
model "J-3 Election districts"
 uses "mmxprs"

 parameters
  REQD = 5                          ! Required number of districts
 end-parameters

 declarations
  QUARTERS = 1..14                  ! Number of quarters
  RDIST: range                      ! Set of election districts

  MAJ: array(RDIST) of integer      ! 1 if majority of votes, 0 otherwise
  DISTR: array(RDIST,QUARTERS) of integer ! 1 if quarter is in district,
                                          ! 0 otherwise
 end-declarations

 include "j3elect_calc.mos"

 declarations
  choose: array(RDIST) of mpvar     ! 1 if district chosen, 0 otherwise
 end-declarations

! Objective: number of votes
 Votes:= sum(d in RDIST) MAJ(d)*choose(d)

! Partitioning
 forall(q in QUARTERS) sum(d in RDIST) DISTR(d,q)*choose(d) = 1

! Desired number of districts
 sum(d in RDIST) choose(d) = REQD

 forall(d in RDIST) choose(d) is_binary
```

```
! Solve the problem
 maximize(Votes)

! Solution printing
 if(getprobstat<>XPRS_OPT) then
  writeln("Problem is infeasible")
 else
  writeln("Total number of votes: ", getobjval)
 end-if

end-model
```

In this model, we test the status of the optimizer (using function `getprobstat`) to decide whether to try printing a solution or not. If no optimal solution is available (`XPRS_OPT`), the problem may be infeasible (`XPRS_INF`), unbounded (`XPRS_UNB`) or the algorithm has not terminated (`XPRS_UNF`), for instance due to a time limit.

The following Mosel procedures implement the algorithm described in Section 15.3.1. For every quarter of the capital, the data file contains the numbers of the voters `POP` and of the expected favorable votes `VOTES`, and the set of neighboring quarters with order numbers greater than the quarter itself. From this input, the procedure `calculate_distr` calculates the list `DISTR` of possible electoral districts. This procedure is called at the end of this piece of code to start the whole data preprocessing algorithm. After the list of possible electoral districts `DISTR` has been established, the majority indicator `MAJ(d)` for every entry `d` in this list is calculated.

```
 declarations
  NUMD: integer                          ! Number of possible districts
  RN: range                              ! Neighboring quarters

  NEIGHB: array(QUARTERS,RN) of integer ! Set of neighboring quarters
  POP: array(QUARTERS) of integer        ! Number of electors (in 1000)
  VOTES: array(QUARTERS) of real         ! Number of favorable votes (in 1000)
  MINPOP,MAXPOP,MINSINGLE: integer       ! Limits on electors per district
 end-declarations

 initializations from 'j3elect_rev.dat'
  NEIGHB POP VOTES MINPOP MAXPOP MINSINGLE
 end-initializations

!**** Save a new entry in the list of districts ****
 procedure save_distr(sQ: set of integer)
  NUMD+=1
  forall(q in sQ) DISTR(NUMD,q):=1
 end-procedure

!**** Add a neighboring quarter to the current set of quarters ****
 procedure add_neighb(toadd:integer, sQ:set of integer)
  declarations
   nQ: set of integer
  end-declarations

  nQ:=sQ+{toadd}
  if(sum(q in nQ) POP(q) >= MINPOP) then      ! Large enough to form distr.
   save_distr(nQ)
  end-if
  forall(p in RN | exists(NEIGHB(toadd,p)))   ! Try adding every neighbor
   if(sum(q in nQ) POP(q)+POP(NEIGHB(toadd,p))<=MAXPOP) then
    add_neighb(NEIGHB(toadd,p), nQ)
   end-if
 end-procedure

!**** Calculate the list of possible districts ****
 procedure calculate_distr
  NUMD:=0
  forall(q in QUARTERS) do
   if (POP(q) >= MINSINGLE and q<>10) then    ! Single quarter districts
    save_distr({q})
   end-if
   forall(p in RN | exists(NEIGHB(q,p)))      ! Try adding every neighbor
    if(POP(q)+POP(NEIGHB(q,p))<=MAXPOP) then
     add_neighb(NEIGHB(q,p),{q})
    end-if
```

```
  end-do

  forall(d in 1..NUMD)                          ! Calculate majorities
   MAJ(d):= if(((sum(q in QUARTERS | DISTR(d,q)>0) VOTES(q)) /
                (sum(q in QUARTERS | DISTR(d,q)>0) POP(q))    >= 0.5), 1, 0)
  finalize(RDIST)
 end-procedure

!**** Start the calculation of the list of possible districts ****
 calculate_distr
```

15.3.3 Results

The problem has no solution for *REQD* = 5. Snirp should have noticed this, as the total population is 540,000, and no district may have more than 100,000 electors; so obviously we need at least 6 districts. Mevo executes Snirp, and appoints Che Pike as Court Mathematician.

We run the Mosel program a second time with *REQD* = 6. For this value we find the partitioning represented in Figure 15.7: all but one district (grey shaded area) is favorable to Mevo.

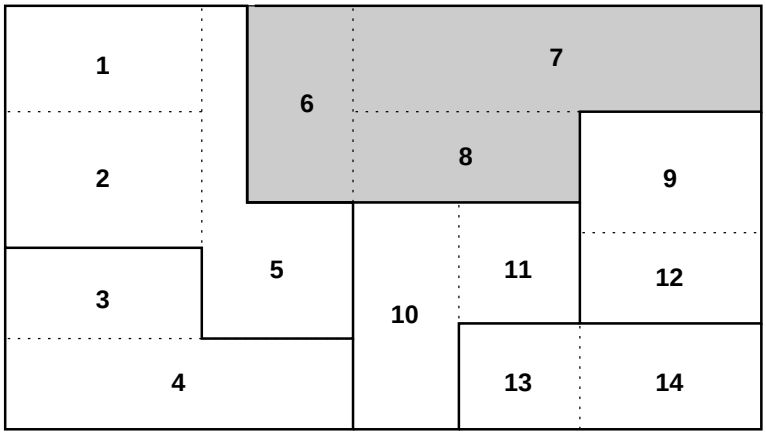

Figure 15.7: Electoral districts for *REQD* = 6

15.4 Gritting roads

In the case of ice, all the streets of a village need to be gritted. A schematic map of the streets is given in Figure 15.8. The nodes correspond to street intersections, the arcs to the streets that need to be gritted. The arcs are labeled with the length of the street in meters.

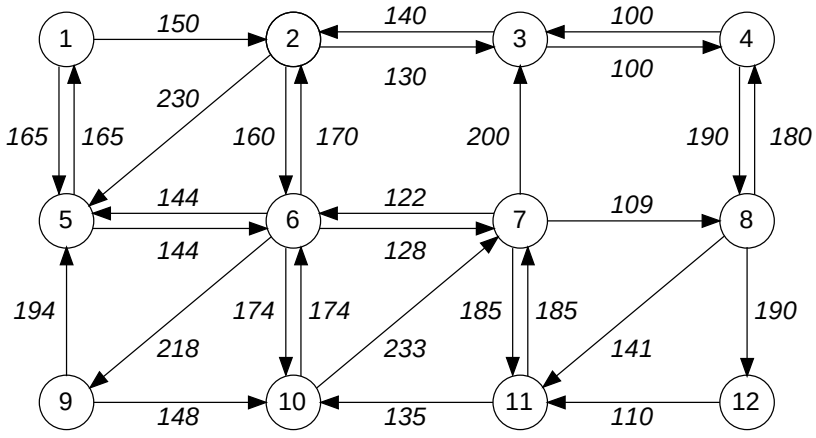

Figure 15.8: Graph of the village streets

The highway maintenance depot with the gritting truck is located at intersection number 1. The truck has a sufficiently large capacity to grit all the streets during a single tour. Due to the one-way streets, it may be forced to pass several times through the same street that has already been gritted. Determine a tour for the gritting truck of minimum total length that passes through all streets. For bidirectional streets, both directions need to be gritted separately.

15.4.1 Model formulation

Let G = (*ISEC, ARCS, LEN*) be the directed graph of the street network. *ISEC* denotes the set of street intersections, *ARCS* the set of arcs, and LEN_{ij} the length LEN_{ij} of arc (i, j) in meters. The tour we are searching for corresponds to a circuit that starts and finishes at the depot node 1, passes at least once over every arc and has a minimum total length. This classical problem is called the **Chinese postman problem** because it was first solved by a Chinese mathematician, Mei-Ko Kuan [Kua62].

A **Eulerian circuit** in a directed graph is a circuit that visits every arc exactly once. If a Eulerian circuit exists, then this is an optimal tour since it is equal to the sum of all arc lengths. A simple test exists to find out whether G may contain a Eulerian circuit: every node i needs to be **equilibrated**, that is the number of arcs arriving at i and leaving i must be equal, see for instance Gibbons [Gib85]. This test must not be confused with the better known case of undirected graphs in which a Eulerian circuit exists if and only if either 0 or 2 nodes are of odd degree.

The graph G of this problem does not have a Eulerian circuit because among others the node 1 is not equilibrated. The circuit will thus re-use streets that are already gritted. The objective is to minimize the total length of such unproductive stretches. Formulating a mathematical model that directly deals with this problem is possible, but difficult. We are going to overcome this difficulty by dealing with a simpler problem, namely the problem of transforming G into a Eulerian graph G' by adding **copies of arcs** to equilibrate the nodes, whilst minimizing the total length of these copies. It then suffices to extract a Eulerian circuit from G', a task for which simple algorithms exist.

The problem of equilibrating the graph at minimum cost is modeled by a fairly simple mathematical model. For every arc (i, j) in *ARCS*, an integer variable use_{ij} is defined that indicates the number of times the tour uses this arc (15.4.3). This number includes one pass for gritting the arc, plus possible other unproductive passages (copies of arcs). Since the gritting of the arc is obligatory, the use_{ij} are at least 1. The constraints (15.4.2) imply that the transformed graph G' must have equilibrated nodes. These constraints are similar to Kirchhoff's law used in flow problems, such as water conveyance in Section 15.2. The objective function (15.4.1) is to minimize the total length driven.

$$\text{minimize} \sum_{(i,j)\in ARCS} LEN_{ij} \cdot use_{ij} \tag{15.4.1}$$

$$\forall i \in ISEC : \sum_{(j,i)\in ARCS} use_{ji} = \sum_{(i,j)\in ARCS} use_{ij} \tag{15.4.2}$$

$$\forall (i, j) \in ARCS : use_{ij} \geq 1 \wedge use_{i,j} \in \mathbb{N} \tag{15.4.3}$$

It is in fact not necessary to specify that the variables are integer because the matrix of this model is a node-arc incidence matrix. The simplex algorithm will automatically find an integer solution. The optimization provides the optimal length and the number of times use_{ij} an arc is used. We could stop there if the exact composition of the tour was not part of the question to answer. We obtain the Eulerian graph G' by adding $use_{ij} - 1$ copies to every arc (i, j). The first pass for gritting the street is already represented by the existing example of the arc.

To obtain a Eulerian circuit, a procedure `add_path(i,tour)` is used. Starting from a node i, it consists of using arcs that have not yet been used as far as possible. An arc that has been visited is marked to avoid re-using it. At the end, the procedure returns the obtained tour *tour*. It is possible to show that *tour* always stops at the end in i and forms a circuit. The procedure `add_path` is first applied at the depot node 1. If *tour* visits all the arcs of G', this is a Eulerian circuit. Otherwise, we increase *tour* by executing `add_path` from a node in *tour* that still has free arcs to visit. The implementation of this algorithm is given after the Mosel implementation of the mathematical model in the following section. The following is pseudo-code for the algorithm implemented with Mosel:

- Call *add_path*(1,tour)
- As long as the *tour* does not visit all arcs:
  - Search the first node i in *tour* from which non-visited arcs leave
  - Call *add_path*(i,tour)

- procedure *add_path*(node,tour)
  - Determine the insertion position *pos*
  - Find an unused path *subtour* starting from and returning to *node*
  - Add the new path *subtour* to the main tour *tour* at position *pos*

15.4.2 Implementation

The mathematical model translates into the following Mosel program. As we have seen, a graph may be encoded in two different ways: in the form of a list of arcs or as an adjacency matrix. The first method is used in this book, for instance, for the opencast mining problem in Section 6.5, the assembly line balancing problem in Section 7.6, and the composition of flight crews in Section 11.2. The second matrix-based representation that is employed here also finds use in the network reliability problem in Section 12.1 and the CCTV surveillance problem of Section 15.2.

The graph is defined by the matrix of arc lengths *LEN*, with $LEN_{ij} = 0$ if arc (i, j) does not exist. The graph is given in sparse format in the data file (that is, it contains only the values of non-zero entries with the corresponding index tuples). By defining *LEN* as a dynamic array, only the entries that are listed in the data file will actually be created so that we can use the function `exists` on this array to test the existence of a street (and at the same time, to enumerate the defined entries of *LEN* efficiently). The variables *use* are created after reading the data, only for the existing entries of *LEN*. It is not required to repeat the filter `exists(LEN(i,j))` every time we define sums with these variables: Mosel automatically skips the undefined entries.

```
model "J-4 Gritting circuit"
 uses "mmxprs"

 declarations
  ISEC = 1..12                              ! Set of street intersections
  LEN: dynamic array(ISEC,ISEC) of integer  ! Length of streets

  use: dynamic array(ISEC,ISEC) of mpvar    ! Frequency of use of streets
 end-declarations

 initializations from 'j4grit.dat'
  LEN
 end-initializations

 forall(i,j in ISEC | exists(LEN(i,j))) create(use(i,j))

! Objective: length of circuit
 Length:= sum(i,j in ISEC | exists(LEN(i,j))) LEN(i,j)*use(i,j)

! Balance traffic flow through intersections
 forall(i in ISEC) sum(j in ISEC) use(i,j) = sum(j in ISEC) use(j,i)

! Grit every street
 forall(i,j in ISEC | exists(LEN(i,j))) use(i,j) >= 1

! Solve the problem
 minimize(Length)

end-model
```

The variables use_{ij} indicate the number of times the truck passes through a street, but what we really want to know is the complete tour it has to drive to grit all streets. To print the result for this problem in a useful form, we need to calculate a Eulerian circuit. This may be done by adding the following lines to the program above. This code fragment implements the algorithm outlined in the previous section. The procedure `add_path` has been turned into a function that returns the number of arcs in the new (sub)tour. We define a counter `ct` to control when all arcs of the Eulerian graph G' calculated by the optimization have been used. Its value is initially set to the result of the optimization and decreased by the number of arcs added as subtours to the main tour. The algorithm stops when the counter is at 0.

```
 forward function find_unused(J: array(range) of integer):integer
 forward function add_path(node:integer, J: array(range) of integer):integer

 ct:=round(getsol(sum(i,j in ISEC) use(i,j)))
```

```
 declarations
  TOUR: array(1..ct+1) of integer
  COUNT: array(ISEC,ISEC) of integer
 end-declarations

! Main loop of the Eulerian circuit algorithm
 forall(i,j in ISEC | exists(LEN(i,j))) COUNT(i,j):=round(getsol(use(i,j)))
 TOUR(1):=1
 ct-=add_path(1,TOUR)
 while(ct>0)
  ct-=add_path(find_unused(TOUR),TOUR)
 writeln("Tour: ", TOUR)

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

! Find first node in list with free path(s)
 function find_unused(J: array(range) of integer):integer
  i:=1
  returned:=1
  while(J(i)>0 and i<getsize(J))
   if (sum(j in ISEC) COUNT(J(i),j) > 0) then
    returned:=J(i)
    break
   else
    i+=1
   end-if
 end-function

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

! Add a subtour to the current tour
 function add_path(node:integer, J: array(range) of integer):integer
  declarations
   NEWJ: array(1..getsize(J)) of integer
  end-declarations

  ! Insertion position
  pos:=1
  while(J(pos)<>node and pos<getsize(J)) pos+=1

  ! Find a new path
  cur:=node; newct:=0
  while(sum(j in ISEC) COUNT(cur,j) > 0) do
   forall(j in ISEC) if(COUNT(cur,j) > 0) then
     COUNT(cur,j)-=1
     newct+=1; NEWJ(newct):=j
     cur:=j
     break
    end-if
  end-do

  ! Add the new path to main journey
  i:=getsize(J)-newct
  while(i>pos) do
   J(i+newct):=J(i)
   i-=1
  end-do
  forall(j in 1..newct) J(pos+j):=NEWJ(j)
  returned:=newct
 end-function
```

15.4.3 Results

The optimization finds a tour of length 5990 meters. All arcs are used once except for the following that are used twice: (3,4), (4,8), (5,1), (5,6), (6,9), (10,6), (11,7).

Figure 15.9 shows the resulting Eulerian graph G'; arcs that are doubled (used twice) are printed with thick lines. A Eulerian circuit in this graph is the tour 1 → 2 → 6 → 5 → 6 → 7 → 8 → 12 → 11 → 7 → 11 → 10 → 7 → 3 → 4 → 3 → 4 → 8 → 4 → 8 → 11 → 7 → 6 → 9 → 5 → 6 → 9 → 10 → 6 → 10 → 6 → 2 → 3 → 2 → 5 → 1 → 5 → 1 (there are several equivalent tours).

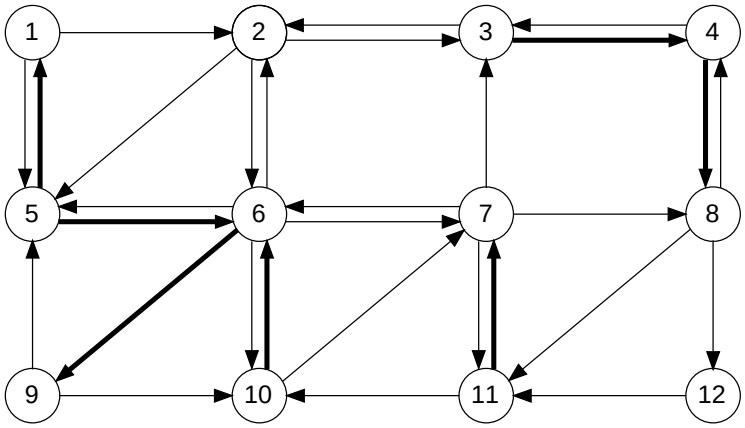

Figure 15.9: Eulerian graph for the extraction of the circuit

15.5 Location of income tax offices

The income tax administration is planning to restructure the network of income tax offices in a region. The graph in Figure 15.10 shows the cities in the region and the major roads. The bold numbers printed close to the cities indicate the population in thousands of inhabitants. The arcs are labeled with the distances in kilometers (printed in italics). 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?

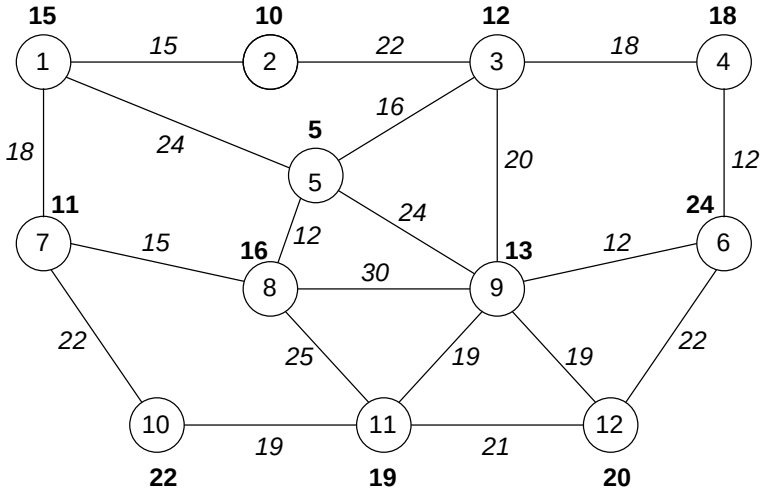

Figure 15.10: Graph of towns and roads of the region

15.5.1 Model formulation

We are dealing here with a classical problem called the **p-median problem**. Let *CITIES* denote the set of cities in the region and *NUMLOC* the number of offices to install. We write POP_c for the population of city *c*. The lengths of the shortest paths between cities are given by a distance matrix *DIST* shown in the following table.

The distance matrix can be calculated from the graph in Figure 15.10 by an all-pairs shortest path algorithm like the Floyd-Warshall algorithm [AMO93]:

- For all node pairs (*c, d*): initialize the distance label $DIST_{cd}$ with plus infinity (a sufficiently large positive value)
- For all nodes *c*: set $DIST_{cc}$:= 0
- For all arcs *a* = (*c, d*): set $DIST_{cd}$ to the length of the arc

Table 15.1: Distance Matrix

	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

- For all nodes b, c, d:
  if $DIST_{cd} > DIST_{cb} + DIST_{bd}$: set $DIST_{cd} := DIST_{cb} + DIST_{bd}$

For the formulation of the problem two groups of binary variables are necessary: a variable $build_c$ that is 1 if and only if a tax office is established in city c (15.5.1), and a variable $depend_{cd}$ that equals 1 if the city c depends on the office in city d (15.5.2). The variables $depend_{cd}$ are required for calculating the average distance per inhabitant and to find out which city depends on which office.

$$\forall c \in CITIES : build_c \in \{0, 1\} \tag{15.5.1}$$

$$\forall c, d \in CITIES : depend_{cd} \in \{0, 1\} \tag{15.5.2}$$

NUMLOC offices should be opened, which is expressed by the constraint (15.5.3). The constraints (15.5.4) indicate that every city depends on a single office.

$$\sum_{c \in CITIES} build_c = NUMLOC \tag{15.5.3}$$

$$\forall c \in CITIES : \sum_{d \in CITIES} depend_{cd} = 1 \tag{15.5.4}$$

The objective function (15.5.5) 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.

$$\text{minimize} \sum_{c \in CITIES} \sum_{d \in CITIES} POP_c \cdot DIST_{cd} \cdot depend_{cd} \tag{15.5.5}$$

If we stop here, we risk assigning cities to tax offices that do not exist! We make this impossible by translating the implication $build_c = 0 \Rightarrow depend_{cd} = 0$ into a linear constraint. From this result the constraints (15.5.6): $build_c$ at 0 forces $depend_{cd}$ to be 0.

$$\forall c, d \in CITIES : depend_{cd} \leq build_d \tag{15.5.6}$$

15.5.2 Implementation

The following Mosel program implements the mathematical model of the previous section. In theory it is necessary to define all the variables as binaries, but one may try to solve the problem by omitting these constraints for the $depend_{cd}$ variables to reduce the total number of integer variables.

```
model "J-5 Tax office location"
 uses "mmxprs"

 forward procedure calculate_dist

 declarations
  CITIES = 1..12                        ! 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 network
  NUMLOC: integer                       ! Desired number of tax offices

  build: array(CITIES) of mpvar         ! 1 if office in city, 0 otherwise
  depend: array(CITIES,CITIES) of mpvar ! (c,d) 1 if city c depends on office
                                        ! in city d, 0 otherwise
 end-declarations

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

! Calculate the distance matrix
 calculate_dist

! Objective: weighted total distance
 TotDist:= sum(c,d in CITIES) POP(c)*DIST(c,d)*depend(c,d)

! Assign cities to offices
 forall(c in CITIES) sum(d in CITIES) depend(c,d) = 1

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

! Relations between dependencies and offices built
 forall(c,d in CITIES) depend(c,d) <= build(d)

 forall(c in CITIES) build(c) is_binary

! Solve the problem
 minimize(TotDist)

end-model
```

The Floyd-Warshall algorithm for finding the shortest distance between every pair of nodes is implemented with procedure `calculate_dist`.

```
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

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

15.5.3 Results

Without specifying that the variables $depend_{cd}$ are binary, luckily the optimizer finds a solution with all variables taking integer values and 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 6 cities 3,4,6,9, and the office in 11 cities 8,10,11,12.

15.6 Efficiency of hospitals

The administration of the hospitals in Paris decides to measure the efficiency of the surgery departments in four major hospitals with a desire to improve the service to the public. To keep this study anonymous, the hospitals are named H1 to H4. The method suggested to measure the efficiency is DEA (Data Envelopment Analysis). This method compares the performance of a fictitious hospital with the performances of the four hospitals.

Three initial indicators (resources) are taken into account: the number of non-medical personnel, the general expenses, and the available number of beds. In addition, four final indicators (services) are analyzed: the number of hospital admissions per day, the number of consultations in the outpatients' clinic, the number of nurses on duty every day, and the number of interns and doctors on duty every day. The corresponding data have been analyzed over a period of two years and the numbers representing a day of average activity in every hospital are given in the following two tables.

Table 15.2: Resource indicators

	H1	H2	H3	H4
Non-medical personnel	90	87	51	66
General expenses (in k€)	38.89	109.48	40.43	48.41
Number of beds	34	33	20	33

Table 15.3: Service indicators

	H1	H2	H3	H4
Admissions	30.12	18.54	20.88	10.42
Consultations	13.54	14.45	8.52	17.74
Interns and doctors	13	7	8	26
Nurses on duty	79	55	47	50

Justify through the DEA method how hospital H2 is performing compared to the others.

15.6.1 Model formulation

15.6.1.1 General idea of the DEA method

To use the DEA method and measure the efficiency of a hospital, we need to develop a mathematical model for every hospital that is to be evaluated. Hospital H2 is used here as an example, but the procedure for measuring the performance of the other hospitals would be similar.

By using Mathematical Programming we are going to construct a fictitious hospital based on the data from the four hospitals. The service indicators of the fictitious hospital will be weighted sums of the service indicators of the hospitals H1 to H4. Similarly, the resource indicators will be weighted sums of the resource indicators of the four hospitals, always using the same coefficients.

The service indicators of the fictitious hospital must be larger than those of the hospital H2. If they are smaller, then the fictitious hospital requires less resources than H2 for a service that is at least equivalent. In other words, hospital H2 is less performing than the fictitious hospital and hence less performing than the other hospitals.

15.6.1.2 Modeling our problem

Let *HOSP* be the set of hospitals in this study and h' the hospital the performance of which we wish to measure. Let $coef_h$ be the DEA coefficient (decision variable) associated with hospital h. The DEA method imposes a requirement that the sum of these coefficients is equal to 1 (15.6.1).

$$\sum_{h \in HOSP} coeff_h = 1 \tag{15.6.1}$$

The constants $INDSERV_{sh}$ represent the service indicator s of every hospital h (given in Table 15.3). The index s takes its value in the set *SERV* of service indicator types. Similarly, $INDRES_{rh}$ represents the indicator for resource r of hospital h (data given in Table 15.2), where r ranges over the set *RES* of resource indicator types. To simplify the model formulation, we introduce variables $fserv_s$ and $fres_r$ for the sums

weighted by the coefficients $coef_h$ for every service and resource indicator type, constraints (15.6.2) and (15.6.3).

$$\forall s \in SERV : \sum_{h \in HOSP} INDSERV_{sh} \cdot coef_h = fserv_s \tag{15.6.2}$$

$$\forall r \in RES : \sum_{h \in HOSP} INDRES_{rh} \cdot coef_h = fres_r \tag{15.6.3}$$

The variables $fserv_s$ (service indicators of the fictitious hospital) must be larger than the service indicators of the hospital h' for which we wish to evaluate the performance. The constraints (15.6.4) establish these relations for all service types in *SERV*.

$$\forall s \in SERV : fserv_s \geq INDSERV_{sh'} \tag{15.6.4}$$

The weighted sum of the resource indicators $fres_r$ (that is, the resource indicators of the fictitious hospital) will not be compared directly with the value of the resource indicators of hospital h', but with a fraction *eff* of these indicators. The constraints (15.6.5) link the coefficients $coef_h$ and this new variable *eff*. The sense of the inequality has to be "less than or equal to" since we are trying to show that the hospital h' uses less resources than the fictitious hospital. To give a numerical example, the number of beds in hospital h' is 33. In this case $33 \cdot eff$ corresponds to the number of beds in the fictitious hospital. If $eff = 1$ then the number of beds available in the fictitious hospital is identical to that of hospital h'. If *eff* is larger than 1, then the fictitious hospital uses more beds than the hospital h' and in the opposite case, the hospital h' is performing worse than the fictitious hospital for this resource.

$$\forall r \in RES : fres_r \leq INDRES_{rh'} \cdot eff \tag{15.6.5}$$

If a solution exists with $eff < 1$, this means that the fictitious hospital needs less resources than the hospital h. The objective function (15.6.6) for the DEA method minimizes the value of *eff*, that is, the resources required by the fictitious hospital. To complete the mathematical model, we need to add the non-negativity constraints (15.6.7) to (15.6.10) for all variables.

$$\text{minimize } eff \tag{15.6.6}$$

$$\forall h \in HOSP : coef_h \geq 0 \tag{15.6.7}$$

$$eff \geq 0 \tag{15.6.8}$$

$$\forall s \in SERV : fserv_s \geq 0 \tag{15.6.9}$$

$$\forall r \in RES : fres_r \geq 0 \tag{15.6.10}$$

15.6.2 Implementation

The following Mosel implementation of the mathematical model solves the problem for every hospital in the set *HOSP*. The constraint on the sum of the DEA coefficients (15.6.1) and the relations defining the service and resource indicators of the fictitious hospital (constraints (15.6.2) and (15.6.3)) are stated only once. The definition of all constraints referring to a specific hospital (namely constraints (15.6.4) and 15.6.5) is replaced in every new execution of the loop by the corresponding constraints for the hospital that is to be evaluated. To be able to replace (and hence delete) a constraint in the problem held in Mosel, it needs to be **named** as shown in the following program for `LimServ` and `LimRes`. (The declaration of constraints is optional, but it is recommended for efficiency reasons if the array sizes are known like in this example).

```
model "J-6 Hospital efficiency"
 uses "mmxprs"

 declarations
  HOSP = 1..4                          ! Set of hospitals
  SERV: set of string                  ! Service indicators
  RES: set of string                   ! Resource indicators

  INDSERV: array(SERV,HOSP) of real    ! Service indicator values
  INDRES: array(RES,HOSP) of real      ! Resource indicator values
 end-declarations

 initializations from 'j6hospit.dat'
  INDSERV INDRES
 end-initializations
```

```
 finalize(SERV); finalize(RES)

 declarations
  eff: mpvar                            ! Efficiency value
  coef: array(HOSP) of mpvar            ! Coefficients for DEA method
  fserv: array(SERV) of mpvar           ! Service indicator of fict. hospital
  fres: array(RES) of mpvar             ! Resource indicator of fict. hospit.
  LimServ: array(SERV) of linctr        ! Hospital-specific service constr.
  LimRes: array(RES) of linctr          ! Hospital-specific resource constr.
 end-declarations

! DEA coefficients
 sum(h in HOSP) coef(h) = 1

! Relations between service and resource indicators
 forall(s in SERV) fserv(s) = sum(h in HOSP) INDSERV(s,h)*coef(h)
 forall(r in RES) fres(r) = sum(h in HOSP) INDRES(r,h)*coef(h)

! Solve the problem for every hospital
 forall(h in HOSP) do
 ! Limits on services and resources for the hospital currently looked at
  forall(s in SERV) LimServ(s):= fserv(s) >= INDSERV(s,h)
  forall(r in RES) LimRes(r):= fres(r) <= INDRES(r,h)*eff

 ! Minimize efficiency index
  minimize(eff)
  writeln("Evaluation of hospital ", h, ": ", getobjval)
 end-do

end-model
```

15.6.3 Results

The program finds efficiency values 1 for the hospitals 1, 3, and 4, so these hospitals have the same performance as the fictitious hospital. For hospital 2, the algorithm returns a value of 0.921. This means that the fictitious hospital reaches the same level of services with only approximately 92% of the resources used by hospital 2. In other words, hospital 2 performs less well than the other hospitals in the study. For every evaluation run, the coefficients $coef_h$ represent the proportions of the four hospitals that form the fictitious hospital.

15.7 References and further material

The mathematical model in Section 15.1 allows us to deal with large maximum flow problems. However, even more efficient graph algorithms exist. Historically the first one was suggested by Ford and Fulkerson [FF67]. Its complexity is $O(nm^2)$, where n denotes the number of nodes in the graph and m the number of arcs. An even faster algorithm in $O(n^2m)$ is due to Ahuja and Orlin [AMO93]. In the latter reference an improved version in $O(mn \cdot \log U)$ is also described, where U denotes the maximum value of the arc capacities.

The problem in Section 15.2 is a problem of covering the edges by the nodes, **node cover** or **vertex cover** in graph theory. This problem is NP-hard [CLR90]. Heuristics are described by Papadimitriou and Steiglitz [PS98].

Partitioning problems are defined over a set S of m elements and a set U of n pre-selected subsets of S. They consist of choosing in U the subsets forming a partitioning of S to optimize the total cost of the selection. Every subset may be encoded as a binary vector and from these vectors results a binary matrix A ($m \times n$). The partitioning constraints may then be written as $A \cdot x = s$, s denoting a vector formed by m entries of 1. **Covering problems** are quite similar, but an element of S may belong to several chosen subsets: the constraints are therefore written as $A \cdot x \geq s$. These problems frequently occur for subdividing geographical regions or for the coverage of zones by services.

Besides the partitioning problem in Section 15.3, this book contains two other covering problems: the cutting of sheet metal in Chapter 9 and the location of telephone transmitters in Chapter 12. Covering problems are easier, since they always have a trivial solution, namely the one using all the available subsets. Partitioning problems may be infeasible like the subdivision into five electoral districts. In both problem types, the simplex algorithm finds a majority of the variables with integer values in the optimum which facilitates the subsequent tree search for the remaining fractional variables, see for instance

Beasley for the covering problem [Bea87] and Fisher [FK90] for both types of problem.

The book by Syslo describes heuristics and exact tree search methods for both problem types, with Pascal source code [SDK83]. Very efficient heuristics guided by the Linear Programming relaxations have recently been suggested for very large size covering problems [CFT99].

The Chinese Postman problem seen in Section 15.4 for gritting the streets of a village also exists for undirected graphs. Edmonds and Johnson give efficient polynomial algorithms for the directed and undirected case [EJ73]. Pascal source code for the extraction of Eulerian circuits and the directed Chinese Postman are provided in the book by Prins [Pri94a]. The Chinese Postman problem becomes a **Rural Postman problem** if only a subset of the arcs needs to be visited.

If quantities are added that need to be distributed over the arcs and if trucks of limited capacity are available and several tours are necessary we obtain a **Capacitated Arc Routing Problem** (CARP). The rural postman and the CARP are NP-hard problems. Numerical examples and heuristics are given in the book by Evans and Minieka [EM92]. Hertz et al. describe local search procedures for the rural postman [HLNH99] and a tabu search methods for the CARP [HLM00].

Location problems like the placement of income tax offices (Section 15.5) form a rich class of combinatorial problems to which the book by Daskin is entirely dedicated [Das95]. The problem of p-centers is another classical one that differs from the p-median problem in its objective that consists of minimizing the maximum distance between an office and a city. This book contains other problems of this type, like the depot location one in Chapter 10 and the choice of locations for mobile phone transmitters in Chapter 12.

The DEA (Data Envelopment Analysis) method from Section 15.6 is widely used for comparing the performances related to several almost identical environments. The first applications were indeed to measure the efficiency of a hospital. Other applications and examples for this methods may be found in Sherman [She84] and Lewin et al. [LM81]. A recent book by Cooper et al. gives a very complete overview of the DEA method and its numerous applications [CST99].

Bibliography

[AA95] E. D. Andersen and K. D. Andersen. Presolving in Linear Programming. *Mathematical Programming*, 71(2):221–245, 1995.

[AC91] D. Applegate and W. Cook. A Computational Study of the Job-Shop Scheduling Problem. *ORSA Journal on Computing*, 3(2):149–156, 1991.

[AD93] R. W. Ashford and R. C. Daniel. Mixed-Integer Programming in Production Scheduling: A Case Study. In T. Ciriani and R. C. Leachman, editors, *Optimization in Industry*, pages 231–239, New York, 1993. John Wiley & Sons.

[AMO93] R. K.. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows. Theory, Algorithms and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[Arb93] A. Arbel. *Exploring Interior-Point Linear Programming: Algorithms and Software*. The MIT Press, 1993.

[ARVK89] I. Adler, M. Resende, G. Veiga, and N. Karmarkar. An Implementation of Karmarkar's Algorithm for Linear Programming. *Mathematical Programming*, 44:297–335, 1989.

[Baa88] S. Baase. *Computer Algorithms*. Addison-Wesley, 1988.

[BC96] J. E. Beasley and B. Cao. A Tree Search Algorithm for the Crew cheduling Problem. *European Journal of Operational Research*, 94:517–526, 1996.

[BDB95] H. Beringer and B. De Backer. Combinatorial Problem Solving in Constraint Logic Programming with Cooperating Solvers. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, pages 245–272, Amsterdam, 1995. Elsevier Science B. V./North-Holland.

[BDGP95] P. Boizumault, Y Delon, C. Guéret, and L. Péridy. Résolution de problèmes en Programmation Logique avec Contraintes. *Revue d'Intelligence Artificielle*, 9(3):383–406, 1995.

[Bea59] E. M. L. Beale. On Quadratic Programming. *Naval. Res. logist. Q.*, 6:227–243, 1959.

[Bea87] J. E. Beasley. An Algorithm for Set Covering Problems. *European Journal of Operational Research*, 31:85–93, 1987.

[BESW93] J. Błazewicz, K. Ecker, G. Schmidt, and J. Węglarz. *Scheduling in Computer and Manufacturing Systems*. Springer, Berlin , Heidelberg, 1993.

[BF76] E. M. L. Beale and J. J. H. Forrest. Global Optimization Using Special Ordered Sets. *Mathematical Programming*, 10(1):52–69, 1976.

[BHJS95] C. Barnhart, C. A. Hane, E. L. Johnson, and G. Sigismondi. A Column Generation and Partitioning Approach for Multi-commodity Flow Problems. *Telecommunication Systems*, 3:239–258, 1995.

[BJ90] M. S. Bazaraa and J. J. Jarvis. *Linear Programming and Network Flows*. Wiley, 1990.

[BJN+98] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance. Branch-and-Price: Column Generation for Solving Huge Integer Programs. *Operations Research*, 46(3):316–329, 1998.

[BK98] A. Bockmayr and T. Kasper. Branch and Infer: A Unifying Framework for Integer and Finite Domain Constraint Programming. *INFORMS Journal on Computing*, 10(3):287–300, 1998.

[BKSA00] J. E. Beasley, M. Krishnamoorthy, Y. M. Sharaiha, and D. Acramson. Scheduling Aircraft Landings – the Static Case. *Transportation Science*, 34:180–197, 2000.

[BL83] E. Balas and P. R. Landweer. Traffic Assignment in Communication Satellites. *Operation Research Letters*, 2(4):141–147, 1983.

[BMR94] L. Bianco, A. Mingozzi, and S. Ricciardelli. A Set Partitioning Approach to the Multiple Depot Vehicle Scheduling Problem. *Optimization Methods and Software*, 3:163–194, 1994.

[BN96] J. P. Boufflet and S. Nègre. Three Methods Used to Solve an Examination Timetabling Problem. In E.K. Burke and P. Ross, editors, *Practice and Theory of Automated Timetabling*, pages 327–344. Springer-Verlag Lecture Notes in Computer Science, 1996.

[Bow56] E. H. Bowman. Production Scheduling by the Transportation Method of Linear Programming. *Operations Research*, 4(1):100–103, 1956.

[BS87] E. S. Buffa and R. K. Sarin. *Modern Production/Operations Management*. Wiley, 1987.

[BT70] E. M. L. Beale and J. A. Tomlin. Special Facilities in a General Mathematical Programming System for Nonconvex Problems Using Ordered Sets of Variables. In J. Lawrence, editor, *Operational Research 69*, pages 447–454. Tavistock Publishing, London, 1970.

[BT79] E. S. Buffa and W. Taubert. *Production-Inventory Systems: Planning and control (3rd Ed.)*. Homewood III, R.D. Irwin, 1979.

[CA99] C. Chu and J. Antonio. Approximation Algorithms to Solve Real-Life Multicriteria Cutting Stock Problems. *Operations Research*, 47(4):495–508, 1999.

[Cam96] J. F. Campbell. Hub Location and the p-hub Median Problem. *Operations Research*, 44(6):923–935, 1996.

[Can77] D. Candea. Issues of Hierarchical Planning in Multistage Production Systems. Technical Report 134, Operations Research Center, MIT, Cambridge, Mass., USA, 1977.

[CC88] J. Carlier and P. Chrétienne. *Problèmes d'ordonnancement : modélisation / complexité / algorithmes*. Masson, 1988.

[CCE+93] J. Carlier, Chrétienne, J. Erschler, C. Hanen, P. Lopez, A. Munier, E. Pinson, M.-C. Portmann, C. Prins, C. Proust, and P. Villon. Les problèmes d'ordonnancement. *RAIRO-Recherche Opérationnelle*, 27(1):77–150, 1993.

[CD99] G. Cornuejols and M. Dawande. A Class of Hard Small 0-1 Programs. *INFORMS Journal on Computing*, 11(2):205–210, 1999.

[CFT99] A. Caprara, M. Fischetti, and P. Toth. A Heuristic Method for the Set Covering Problem. *Operations Research*, 47(5):730–743, 1999.

[CGJ96] E. G. Coffman, M. R. Garey, and D. S. Johnson. Approximation Algorithms for Bin-packing: a Survey. In D. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, pages 46–93. PWS Publishing, Boston, 1996.

[CH77] J. A. Clark and N. A. J. Hastings. Decision Networks. *Operations Research Quaterly*, 20:51–68, 1977.

[CH02] Y. Colombani and S. Heipcke. Mosel: An Extensible Environment for Modeling and Programming Solutions. In N. Jussien and F. Laburthe, editors, *Proceedings of CP-AI-OR'02*, pages 277–290, Le Croisic, March 2002.

[Chv83] V. Chvàtal. *Linear Programming*. W.H. Freeman, 1983.

[CHW98] P. Chan, K. Heus, and G. Weil. Nurse Scheduling with Global Constraints in CHIP: GYMNASTE. In *Proceedings of the Fourth International Conference on the Practical Application of Constraint Technology, PACT98*, pages 157–169, London, UK, March 1998. The Practical Application Company.

[CL96] M. W. Carter and G. Laporte. Recent Development in Practical Examination Timetabling. In E.K. Burke and P. Ross, editors, *Practice and Theory of Automated Timetabling*, pages 3–21. Springer-Verlag Lecture Notes in Computer Science, 1996.

[CLR90] T. H. Cormen, C. L. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill, New York, 1990.

[CMT79a] N. Christofides, A. Mingozzi, and P. Toth. Loading Problems. In N. Christofides and al., editors, *Combinatorial Optimization*, pages 339–369. Wiley, 1979.

[CMT79b] N. Christofides, A. Mingozzi, and P. Toth. The Vehicle Routing Problem. In N. Christofides and al., editors, *Combinatorial Optimization*, pages 315–338. Wiley, 1979.

[CST99] W. W. Cooper, L. M. Seiford, and K. Tone. *Data Envelopment Analysis*. Kluwer, 1999.

[CW64] G. Clarke and J. W. Wright. Xheduling of Vehicles from a Central Depot to a Number of Delivery Points. *Operations Research*, 12(4):568–581, 1964.

[Dan63] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, NJ, 1963.

[Das95] M. S. Daskin. *Network and Discrete Location*. Wiley, 1995.

[DDS92] M. Desrochers, J. Desrosiers, and M. Solomon. A New Optimization Algorithm for the Vehicle Routing Problem with Time Windows. *Operations Research*, 40(2):342–354, 1992.

[DE73] E. M. Dar-El. MALB, a Heuristic Technique for Balancing Large Scale Single-Model Assembly Lines. *AIIE Transactions*, 5(4):–, 1973.

[DK99] A. Dutta and P. Kubat. Design of Partially Survivable Networks for Cellular Telecommunications Systems. *European Journal of Operational Research*, 118(1):52–64, 1999.

[DPL94] S. Dauzère-Pérès and J. B. Lasserre. An Integrated Approach in Production Planning and Schedulin. In *Lecture Notes in Economics and Mathematical Systems 411*. Springer-Verlag, 1994.

[DW97] D. De Werra. The Combinatorics of Timetabling. *European Journal of Operational Research*, 96(3):504–513, 1997.

[DZ78] U. Derigs and U. Zimmerman. An Augmenting Path Method for Solving Linear Bottleneck Assignment Problems. *Computing*, 19:285–295, 1978.

[EJ73] J. Edmonds and E. L. Johnson. Matching, Euler Tours, and the Chinese Postman. *Mathematical Programming*, 5:88–124, 1973.

[EM92] J. E. Evans and E. Minieka. *Optimization Algorithms for Networks and Graphs*. Marcel Dekker Inc., 1992.

[Erl78] D. Erlenkotter. A Dual-Based Procedure for Uncapacited Facility Location. *Operations Research*, 26:992–1009, 1978.

[Eva64] H. F. Evart. *Introduction to PERT*. Allyn & Bacon, 1964.

[FF67] L. Ford and D. Fulkerson. *Flots dans les graphes*. Gauthier-Villars, Paris, 1967.

[FJVW86] M. L. Fisher, R. Jaikumar, and L. N. Van Wassenhove. A Multiplier Adjustment Method for the Generalized Assignment Problem. *Management Science*, 39(2):1095–1103, 1986.

[FK90] M. L. Fisher and P. Kedia. Optimal Solutions of Set Covering/Partitioning Problems Using Dual Heuristics. *Management Science*, 36:674–688, 1990.

[Fle87] R. Fletcher. *Practical Methods of Optimization (2nd ed.)*. John Wiley, Chichester UK, 1987.

[Fre82] S. French. *Sequencing and Scheduling. An Introduction to the Mathematics of the Job-Shop*. John Wiley and Sons, New York, 1982.

[Gar63] L. L. Garver. Power Scheduling by Integer Programming. *IEEE Transactions on Power Apparatus and Systems*, 81:730–735, 1963.

[Gar00] D. T. Gardner. Efficient Formulation of Electric Utility Resource Planning Models. *Journal of the Operational Society*, 51(2):231–236, 2000.

[GG61] P. C. Gilmore and R. E. Gomory. A Linear Programming Approach to the Cutting Stock Problem. Part I. *Operations Research*, 9:849–859, 1961.

[GG63] P. C. Gilmore and R. E. Gomory. A Linear Programming Approach to the Cutting Stock Problem. Part II. *Operations Research*, 11:863–888, 1963.

[GH91] M. Grötschel and O. Holland. Solution of Large-Scale Symmetric Traveling Salesman Problems. *Mathematical Programming*, 51:141–202, 1991.

[GHL94] M. Gendreau, A. Hertz, and G. Laporte. A Tabu Search Algorithm for the Vehicle Routing Problem. *Management Science*, 40:1276–1290, 1994.

[Gib85] A. Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, Cambridge, New York, 1985.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W.H. Freeman & Company, San Francisco, 1979.

[Gle80] J. J. Glen. A Parametric Programming Method for Beef Cattle Ration Formulation. *Journal of the Operational Research Society*, 31:689–690, 1980.

[GM90] M. Gondran and M. Minous. *Graphes et algorithmes (2nd Ed.)*. Eyrolles, 1990.

[GP98] C. Guéret and C. Prins. Classical and New Heuristics for the Open-Shop Scheduling Problem: a Computational Evaluation. *European Journal of Operational Research*, 107(2):306–314, 1998.

[GP01] Gunluk and Y. Pochet. Mixing mixed-integer inequalities. *Mathematical Programming*, 90:429–457, 2001.

[Gra69] R. L. Graham. Bounds on Multiprocessing Timing Anomalies. *Siam Journal on Applied Mathematics*, 17:416–429, 1969.

[GS76] T. Gonzalez and S. Sahni. Open-Shop Scheduling to Minimize Finish Time. *Journal of the ACM*, 23:665–679, 1976.

[GS87] A. Gersht and A. Shulman. A New Algorithm for the Solution of the Minimum Cost Multicommodity Flow Problem. In *IEEE 26th Conference on Decision and Control (CDC'87)*, pages 748–758, Los Angeles, December 1987.

[GTdW93] F. Glover, E. Taillard, and D. de Werra. A User's Guide to Tabu Search. *Annals of Operations Research*, 41(1–4):3–28, 1993.

[HC84] A. C. Hax and D. Candea. *Production and Inventory Management*. Prentice Hall, 1984.

[HC99] S. Heipcke and Y. Colombani. Julian's Problem. Work Assignment at a Disabled Care Centre. *ORInsight*, 12(1):16–20, 1999.

[Hei99a] S. Heipcke. *Combined Modelling and Problem Solving in Constraint Programming and Mathematical Programming*. PhD thesis, University of Buckingham, 1999.

[Hei99b] S. Heipcke. Comparing Constraint Programming and Mathematical Programming Approaches to Discrete Optimisation. The Change Problem. *Journal of the Operational Research Society*, 50(6):581–595, 1999.

[HH60] F. Hanssmann and S. W. Hess. A Linear Programming Approach to Production and Employment Scheduling. *Management Technology*, 1:–, 1960.

[Hit41] F. L. Hitchcock. The Distribution of a Product from Several Sources to Numerous Localities. *J. Math. Phys.*, 20(2):224–230, 1941.

[HK70] M. Held and R. Karp. The Traveling Salesman Problem and Minimum Spanning Trees. *Operations Research*, 18:1138–1162, 1970.

[HLM00] A. Hertz, G. Laporte, and M. Mittaz. A Tabu Search Heuristic for the Capacited Arc Routing Problem. *Operations Research*, 48(1):129–135, 2000.

[HLNH99] A. Hertz, G. Laporte, and P. Nanchen Hugo. Improvement Procedures for the Undirected Rural Postman Problem. *INFORMS Journal on Computing*, 11(1):53–62, 1999.

[HW95] J. L. Ho and J. S. Wong. Makespan Minimization for m Parallel Identical Processors. *Naval Research Logistics*, 42(6):935–948, 1995.

[HZ96] M. Hifi and V. Zissimopoulos. A Recursive Exact Algorithm for Weighted Two-Dimensional Cutting. *European Journal of Operational Research*, 91:553–564, 1996.

[Jak96] S. Jakobs. On Genetic Algorithms for the Packing of Polygons. *European Journal of Operational Research*, 88:165–181, 1996.

[JM74] L. A. Johnson and D. C. Montgomery. *Operations Research in Production Planning, Scheduling, and Inventory Control*. Wiley, New York, 1974.

[JNS00] E. L. Johnson, G. L. Nemhauser, and M. W. P. Savelsbergh. Progress in Linear Programming Based Algorithms for Integer Programming: An Exposition. *INFORMS Journal on Computing*, 12(1):2–23, 2000.

[JSV98] B. Jaumard, F. Semet, and T. Vovor. A Generalized Linear Programming Model for Nurse Scheduling. *European Journal of Operational Research*, 107(1):1–18, 1998.

[Kao79] E. P. C. Kao. A Multi-Product Dynamic Lot-Size Problem with Individual and Joint Set-up Costs. *Operations Research*, 27(2):279–289, 1979.

[Kar84] N. Karmarkar. A New Polynomial Time Algorithm for Linear Programming. *Combinatorica*, 4:373–395, 1984.

[Kas98] R. G. Kasilingam. *Logistics and Transportation*. Kluwer Academic Publishers, 1998.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–680, 1983.

[Kua62] M. K. Kuan. Graphic Programming Using Odd and Even Points. *Chinese Mathematics*, 1:273–277, 1962.

[LD60] A. H. Land and A. G. Doig. An Automatic Method for Solving Discrete Programming Problems. *Econometrica*, 28:497–520, 1960.

[LDN84] G. Laporte, M. Desrochers, and Y. Nobert. Two Exact Algorithms for the Distance-Constrained Vehicle Routing Problem. *Networks*, 14:161–172, 1984.

[LM81] A. Y. Lewin and R. C. Morey. Measuring the Relative Efficiency and Output Potential of Public Sector Organizations: an Application of Data Envelopment Analysis. *International Journal of Policy Analysis and Information Systems*, 5(4):267–285, 1981.

[LMV99] A. Lodi, S. Martello, and D. Vigo. Heuristics and Metaheuristics for a class of two-Dimensional Bin Packing Problems. *INFORMS Journal on Computing*, 11(4):345–357, 1999.

[LQ92] J. B. Lasserre and M. Queyranne. Generic Scheduling Polyedra and a new Mixed-Integer Formulation for a Single Machine Scheduling. In *Second IPCO Conference*, Pittsburg, 1992. Carnegie-Mellon University.

[LR99] P. Lopez and F. Roubellat. *Ordonnancement*. Economica, 1999.

[LS98] J. T. Linderoth and M. W. P. Savelsbergh. A Computational Study of Search Strategies for Mixed Integer Programming. INFORMS Journal on Computing (to appear), 1998.

[LT71] L. S. Lasdon and R. C. Terjung. An Efficient Algorithm for Multi-Item Scheduling. *Operations Research*, 19(4):946–969, 1971.

[Lue84] D. G. Luenberger. *Linear and Non-Linear Programming*. Addison-Wesley, 1984.

[Man86] A. S. Manne. GAMS/MINOS: Three Examples. Technical report, Department of Operations Research, Stanford University, 1986.

[MB86] M. Minoux and G. Bartnik. *Graphes, algorithmes, logiciels*. Dunod, 1986.

[McC69] W. H. S. McColl. Management and Operations in an Oil Company. *Operations Research Quaterly*, 20:64–65, 1969.

[Min75] M. Minoux. Résolution des problèmes de multiflots en nombres entiers dans les grands réseaux. *RAIRO - Recherche Opérationnelle*, 3:–, 1975.

[MS98] K. Marriott and P. J. Stuckey. *Programming with Constraints*. MIT Press, Cambridge MA, 1998.

[MT79] S. Martello and P. Toth. The 0-1 Knapsack Problem. In N. Christofides and al., editors, *Combinatorial Optimization*. Wiley, 1979.

[MT90] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley, New York, 1990.

[MW01] H. Marchand and L. Wolsey. Aggregation and mixed-integer rounding to solve MIPs. *Operations Research*, 49:363–371, 2001.

[NEH83] M. Nawaz, E.E. Enscore, and I. Ham. A Heuristic Algorithm for the m-Machine, n-Job Flow-Shop Sequencing Problem. *Omega*, 11(1):91–95, 1983.

[Orl75] J. A. Orlicky. *Material Requirement Planning: the New Way of Life in Production and Inventory Management*. McGraw-Hill, New York, 1975.

[PFTVed] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C – The Art of Scientific Computing*. Cambridge University Press, Cambridge, 1992 (2nd ed.).

[Pin95a] M. Pinedo. *Scheduling: Theory, Algorithms and Systems*. Prentice Hall, 1995.

[Pin95b] E. Pinson. The Job Shop Scheduling Problem: A Concise Survey and Some Recent Developments. In Ph. Chrétienne, E. G. Coffman, J. K. Lenstra, and Z. Liu, editors, *Scheduling Theory and its Applications*, pages 277–294, Chichester, New York, 1995. John Wiley.

[Pri94a] C. Prins. *Algorithmes de graphes avec programmes en Pascal*. Eyrolles, 1994.

[Pri94b] C. Prins. An Overview of Scheduling Problems Arising in Satellite Communications. *J. Opl. Res. Soc.*, 45(6):611–623, 1994.

[PS98] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization*. Dover, 1998.

[Roo97] K. Roos. *Theory and Algorithms for Linear Optimization. An Interior Point Approach*. John Wiley, Chichester, New York, 1997.

[Ros85] Roseaux. *Exercices et problèmes résolus de Recherche Opérationnelle – Tome 3*. Masson, 1985.

[RWH99] R. Rodošek, M. G. Wallace, and M. T. Hajian. A New Approach to Integrating Mixed Integer Programming and CLP. *Annals of Operations Research*, 86:63–87, 1999.

[Sav94] M. W. P. Savelsbergh. Preprocessing and Probing Techniques for Mixed Integer Programming Problems. *ORSA Journal on Computing*, 6(4):445–454, 1994.

[SC81] D. W. Sutton and P. A. Coates. On-Line Mixture Calculation System for Stainless Steel Production by BSC Stainless: the Least Through Cost Mix System (LTCM). *Journal of the Operational Research Society*, 32:165–169, 1981.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1986.

[Sch97] L. Schrage. *Optimization Modeling with LINDO (5th ed.)*. Duxbury, 1997.

[Sch99] A. Scholl. *Balancing and Sequencing of Assembly Lines*. Physica-Verlag, 1999.

[SDK83] M. M. Syslo, N. Deo, and J. S. Kowalik. *Discrete Optimization Algorithms with Pascal Programs*. Prentice Hall, 1983.

[Sev98] M. Sevaux. *Étude de deux problèmes d'optimisation en planification et ordonnancement*. PhD thesis, Thèse de l'université Pierre-et-Marie-Curie, Paris VI, 1998.

[She84] H. D. Sherman. Hospital Efficiency Measurement and Evaluation. *Medical Care*, 22(10):922–938, 1984.

[SM74] A. M. Sasson and H. M. Merrill. Some Applications of Optimization Techniques to Power Systems Problems. In *IEEE,62*, pages 959–972, 1974.

[SN96] A. Sofer and S. G. Nash. *Linear and Non-Linear Programming*. McGraw-Hill, 1996.

[SPP98] E. Silver, D. F. Pyke, and R. Peterson. *Inventory Management and Production Planning and Scheduling*. Wiley, 1998.

[SRP92] P. E. Sweeney and E. Ridenour Paternoster. Cutting and Packing Problems: a Categorized, Application-Oriented Research Bibliography. *Journal of the Operational Research Society*, 43(7):691–706, 1992.

[SS98] B. Sansò and P. Soriano. *Telecommunication Network Planning*. Kluwer, 1998.

[SSR00] C. H. Scott, O. G. Skelton, and E. Rolland. Tactical and Strategic Models for Satellite Customer Assignment. *Journal of the Operational Research Society*, 51(1):61–71, 2000.

[Tri84] A. Tripathy. School Timetabling: a Case in Large Binary Integer Linear Programming. *Management Science*, 30(12):1473–1489, 1984.

[VH98] P. Van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, Cambridge, MA, 1998.

[VW02] S. Voß and D. L. Woodruff. *Introduction to Computational Optimization Methods for Production Planning in a Supply Chain*. Springer, Berlin, 2002.

[WA88] E. A. Wasil and A. A. Assad. Project Management on the PC: Software, Applications and Trends. *Interfaces*, 18(2):75–84, 1988.

[Wag59] H. M. Wagner. An Integer Programming Model for Machine Scheduling. *Naval Research Logistics Wuaterly*, 6:131–140, 1959.

[Wal76] W. E. Walker. A Heuristic Adjacent Extreme Point Algorithm for the Fixed Charge Problem. *Management Science*, 22(5):587–596, 1976.

[WH89] M. Widmer and A. Hertz. A New Heuristic Method for the Flow Shop Sequencing Problem. *European Journal of Operational Research*, 41:186–193, 1989.

[Wil93] H. P. Williams. *Model Building in Mathematical Programming. (3rd Rev. Ed.)*. John Wiley, Chichester, 1993.

[Win94] W. L. Winston. *Operations Research: Applications and Algorithms (3rd ed.)*. Duxbury Press, Belmont, 1994.

[Win98] W. L. Winston. *Financial Models Using Simulation and Optimization*. Palisade, 1998.

[Wol98] L. A. Wolsey. *Integer Programming*. Wiley Interscience, 1998.

[WW58] H. M. Wagner and T. M. Whitin. A Dynamic Version of the Economic Lot Size Model. *Management Science*, 50(1):89–96, 1958.

[Zen96] Z. A. Zenios. *Financial Optimization*. Cambridge University Press, 1996.

Index