Tutorial for Operations Research: Modeling in GNU MathProg

András Éles, György Dósa

Faculty of Information Technology
University of Pannonia, Veszprém, Hungary
2019

Contents

Chapter 1

Introduction

This Tutorial is intended to give an insight into mathematical programming, and Operations Research in general, through the development of Linear Programming (LP) and Mixed-Integer Linear Programming (MILP) models using the GNU MathProg (GMPL) modeling language.

Optimization means finding the most suitable solution in real life situations. This is often possible using common sense or some simple logic, but there are many cases where advanced approaches are needed. Usually a mathematical model of the problem is considered, and solution methods are designed utilizing the best of the available mathematical theoretical background, computer theory and existing technology.

The problem is generally the vast number of possible solutions to a problem, which are impossible to be all taken into account one by one, even with the fastest computers available. In many situations, instead of a perfect solution, „rules of thumb", also called heuristic methods are used. These simplify the underlying model of the problem, trim the set of possible solutions to be investigated or both. This sacrifices optimality but makes finding a fairly good solution of the problem computationally possible. Nevertheless, even in these cases, finding such a solution usually requires some optimization to be made, for example to find the best solution from a few candidates.

If an optimization problem is faced, a „conventional" method of solving is the design, implementation and application of a specific **algorithm**, which usually require some programming language. The concept of algorithm has a very general meaning, and it also depends on what programming language we choose to implement it, but usually the data structures used and the steps to be made are described. This is what most common programming languages can do, for example C/C++, Java, Python, etc. The common point in these languages is that not only the „What?" is answered by implementing the algorithm, but the „How?" is also addressed by the programmer. That is, the exact steps to be made with the data are described. For example, Dijkstra's algorithm can find the shortest path in a graph, Kruskal's algorithm can find the cheapest spanning tree of a graph, or the Ford-Fulkerson algorithm is capable of finding the optimal flow in a flow network.

Mathematical programming methods are an alternative. Instead of describing the steps to be made, only the mathematical model of the real-world problem is formulated. The mathematical model consists of statements like equations, inequalities, logical expressions and more, which correspond to the rules governing the real-world problem and its solution. Then, the real-world problem is solved by finding solutions to the system of these equations, inequalities or other statements. However, only the formulation of the mathematical model is up to the programmer, not the algorithm of how it is solved. The latter is actually done by some **solver** software, which is designed to solve a particular class of mathematical programming models.

In this Tutorial, a subset of mathematical programming methods, namely **Linear Programming (LP)** and **Mixed-Integer Linear Programming (MILP)** models are in scope, which can be formulated by the GNU MathProg (GMPL) modeling language [1]. Tackling real-world optimization problems by formulating a mathematical programming model and solving it with a solver software is a valuable expertise that is substantially different from programming in the „conventional" way. Nevertheless, an algorithm designed to solve a specific optimization problem may be much faster than a mathematical programming model: many well-studied optimization problems have state-of-the-art algorithmic solutions. However, for some problems, even for some where decent algorithms exist, designing a mathematical programming model instead can be surprisingly fast and easy. Mathematical programming models are also more flexible in case the real-world problem itself is changed.

The algorithms that the solvers utilize to actually solve our formulated mathematical programming model are on their own an interesting subject, but is not in focus. In short, the way these models can be formulated are more important than how these mathematical models can be solved to optimality.

The modeling language GNU MathProg is selected because, along with its solver `glpsol` in the GNU Linear Programming Kit [2], these are tools relatively easy to be understood, and available under the GNU General Public License. There are many other free, and commercial tools for both formulating and solving not only MILP, but more general mathematical programming models. Many of such tools are much faster than `glpsol`.

There are several other learning materials available in this subject. The basic installation of GLPK also includes a bunch of example models, including complex ones. GLPK also has a Wikibooks project where usage is covered in high detail [3]. We would like to recommend the Linear Programming tutorial of Hegyháti [4]. Moreover, many useful books and lecture materials can be found at the Hungarian Tankönyvtár homepage [5].

This Tutorial presents the basics of the language, and some key problems that can be easily solved by mathematical programming tools, to give an insight into the strength of optimization with mathematical, and particularly MILP models. Some algorithmic solution methods for particular problems are also presented.

The rest of the Tutorial is organized as follows.

- Chapter 2 provides the introduction to the main goals, tools and concepts of optimization.
- Chapter 3 introduces to the basic usage of GNU MathProg and solver `glpsol`. A minimal, „Hello, World!" model file is shown.
- In Chapter 4, linear equation systems are solved using GNU MathProg with gradually improving model implementations. This part aims at demonstrating features of the language that are required to develop concise and general LP/MILP models, most importantly indexing and the separation of model and data sections.
- Chapter 5 introduces the production problem, a simple optimization problem. Incremental development of models is in focus, demonstrated through changes, extensions of the problem definition. Some additional language features, and integer variables are used.
- Chapter 6 shows another simple optimization problem, the transportation problem. Modeling techniques for some cost functions are presented, which can be part of more complex problems.
- Chapter 7 mentions some harder problems requiring MILP model formulations. Advanced modeling and coding techniques, and possible graphical outputs are presented.
- Chapter 8 gives an insight into the LP/MILP solving algorithms, explaining briefly how software tools may work in the background.

This Tutorial contains attachments. All GNU MathProg codes including model and data files, and its outputs mentioned in the text can be found, grouped by chapters and sections. (A more detailed description is available in the attachment itself.)

For learning purposes, it is recommended to manually see, and solve the models with their corresponding data file(s) to investigate the results. Note that most of the codes and more important output file segments are included in the text of the Tutorial itself, full or in part, using a specific format.

```
GNU MathProg code of model files is formatted like this.
```

```
GNU MathProg code of data sections that may go to
separate data files are formatted like this.
```

```
Output generated by the solver is formatted like this.
```

```
Commands to be run are formatted like this.
```

If you have questions, or in case you find any errors in this Tutorial, please feel free to contact us at `eles@dcs.uni-pannon.hu`. We hope you will find this material useful.

Chapter 2

Optimization

2.1 Equations

Equations are one of the easiest and strongest mathematical modeling tools we have. Let's see an example problem that can be solved by an equation.

Problem 1.
Alice's train departures 75 minutes from now. She lives 3 km away from the station and wants to walk, but she and also needs 30 minutes to get ready. How fast shall she walk to the station to catch the train?

We can see that what we actually seek is the *minimal average speed* Alice have to move with. She can be faster at some points of her walk, or slower, but overall, the average speed counts, determining when she will arrive. She can even go with constant speed to achieve the same result. Also, we seek to find the minimal speed, with which she just catches the train. Of course, she can be even faster, then she will have some waiting time at the station. These steps may be trivial, but are also part of the modeling procedure: we simplified the problem into finding a single minimal speed value and we may assume that Alice goes with that constant speed, and just catches the train.

The problem can be solved by expressing the time until she arrives to the station in *two different ways*. Once, the time she arrives is 75 min, by the train schedule. On the other hand, the time she arrives is 30 min for getting ready, plus the time she walks. The latter term is something which is not known, it depends on Alice's speed. Now, we introduce a **variable** x to express Alice's speed, which is in question. Variable x is some value that can be chosen, that Alice has freedom upon. She can choose her speed in order to arrive to the station sooner or later.

With x, we can express the time she spends walking easily by $\frac{3\,\mathrm{km}}{x}$. Now an equation can be formulated based on the fact that Alice's arriving time is expressed in two different ways.

$$30\,\mathrm{min} + \frac{3\,\mathrm{km}}{x} = 75\,\mathrm{min} \tag{1}$$

The task for Alice is to choose an appropriate x so that this equation **holds**. From this point, the equation is formulated, we can forget the real-world problem and apply our mathematical knowledge about solving equations. This is an easy one. We can, for example, subtract 45 minutes from both sides, multiply by x (which, we can assume not to be zero), then dividing by 3 km, in that order. We arrive to the following.

$$4\,\frac{\mathrm{km}}{\mathrm{h}} = x \tag{2}$$

What we actually do by solving an equation is transforming it to easier forms so that the next form is a direct consequence of the previous. The final form is $x = 4\,\frac{\text{km}}{\text{h}}$, which literally means the following: „If $30\,\text{min} + \frac{3\,\text{km}}{x} = 75\,\text{min}$, then its direct consequence is $x = 4\,\frac{\text{km}}{\text{h}}$”. So, if Alice just catches the train, then her average speed is $4\frac{km}{h}$ And at this point we finally got the result and interpreted it in the reality. This is the general way on how equations are used in problem solving. First, we identify the freedom we have at the problem to be solved (x, the speed). Then, we use equations, as modeling tools to express the rules of reality that must be obeyed (the first equation). The model is now ready, we apply our solution techniques which can be quite general for any kinds of equations and have nothing to do with the underlying real-world problems. In this case, the equation is transformed in a chain of direct consequences until the variable is expressed directly as a constant value. Finally, if the result is obtained for the mathematical model, we interpret it as the solution of our initial real-world problem.

Of course, there are much more difficult examples of equation solving (for example, multiple solutions for single equations, fake solutions, multiple variables, multiple equations at the same time, etc.), which are not covered here. But the point is that the scheme is similar each time: we formulate a mathematical model that describes the real-world situation, solve that model with some well-known general techniques and then interpret the solution of the model as the solution of the original problem.

2.2 Finding the „Best” solution

Note that the question about Alice catching the train in Problem 1 was how fast she can go to catch the train. The answer was $x = 4\,\frac{\text{km}}{\text{h}}$, but that is not totally correct. Actually, $x = 5\,\frac{\text{km}}{\text{h}}$, $x = 10\,\frac{\text{km}}{\text{h}}$ or even $x = 15\,\frac{\text{km}}{\text{h}}$ are solutions as well for her, if she can run that fast. However $x = 3\,\frac{\text{km}}{\text{h}}$ is not a solution, as she will miss the train if she is that slow. Of course, for this particular problem, we understand that there is a minimum speed of $x = 4\,\frac{\text{km}}{\text{h}}$, for which Alice *just* catches the train in time. However, the more precise formulation would be the following **inequality** instead.

$$30\,\text{min} + \frac{3\,\text{km}}{x} \leq 75\,\text{min} \tag{3}$$

Two quantities are here: the time Alice needs to reach the station if her speed is x, that is, $30\,\text{min} + \frac{3\,\text{km}}{x}$, and the time the train departs, $75\,\text{min}$. These two quantities are related: Alice must arrive *before or exactly* when the train is due (not later, in other words). This can be expressed as an inequality rather than an equation, but the goal is the same. We must find a speed for Alice x for which the inequality holds. Technically, this can be done exactly the same way as the equation itself was solved, leading to $x \geq 4\,\frac{\text{km}}{\text{h}}$. That means that the solutions are actually all x speeds that are at least $4\,\frac{\text{km}}{\text{h}}$. There are infinitely many solutions for the inequality, so infinitely many speeds Alice can choose.

Now this is more precise, but Alice may be interested in the minimum speed she can choose, so let x be as small as possible. The formulation is as follows.

$$\begin{array}{rl} \text{minimize}: & x \\ \text{subject to}: & 30\,\text{min} + \frac{3\,\text{km}}{x} \leq 75\,\text{min} \end{array} \tag{4}$$

At this point, the formulated problem is an **optimization** problem. The goal is not only finding one suitable solution for a real-world problem, but finding the *most suitable* one in some aspect. In the mathematical point of view, optimization is not only finding a solution for an equation, inequality, or a set of such, but finding the best. Optimization is a sub-field of Operations Research, as it is an inevitable way of supporting business decisions: given a complex real-world situation, what shall be the best response?

2.3 Main concepts

Now that we have seen some optimization from Problem 1, it is time to establish some basic concepts of optimization models.

When solving a real-world problem by defining some variables, equations, inequalities, etc. we **formulate** a mathematical **model** for the problem. That model is then solved with mathematical knowledge, that is usually independent of the original, real-world problem. If the solution procedure is made by some software (usually the case), that software is called a **solver**.

There are **variables** in an optimization model. These represent our freedom of choice: the way the variable can take different values represent our different actions. In Problem 1, there is a single variable, x representing Alice's average speed.

A **solution** of an optimization model is when all the variables take some value. If there are multiple variables in the model, then a single solution consists of values for all of these variables. Two solutions are considered different if any of the variables take a different value. In Problem 1, there are infinitely many solutions, like $x = 4\,\frac{\text{km}}{\text{h}}$, $x = 10\,\frac{\text{km}}{\text{h}}$, and moreover, $x = 1\,\frac{\text{km}}{\text{h}}$ as well (see below), although Alice misses the train.

Usually the variables cannot take arbitrary values. In Problem 1, Alice cannot choose a speed if she would miss the train with it. Such restrictions that must apply, are called **constraints**. So in the example problem, there is a single constraint, which is expressed by Inequality 3.

There are a wide range of constraints that may appear in an optimization problem, some cannot even be expressed as inequalities. The concept of **bounds** shall be mentioned: these are constraints that require a single variable to be not less than, or not more than a given constant limit. For example, $x \geq 0\,\frac{\text{km}}{\text{h}}$ is a bound that must also be ensured. The problem required Alice to move with a positive speed, so it was implicitly enforced and neglected. In many optimization problems, however, we must explicitly consider bounds, otherwise solutions might not reflect reality. Bounds also have a great practical importance in model solution algorithms.

Constraints decide whether solutions are **feasible** or **infeasible**. A feasible solution is when all variables take a value for which all the constraints hold. Otherwise, it is an infeasible solution. In Problem 1, solutions $x = 4\,\frac{\text{km}}{\text{h}}$ and $x = 10\,\frac{\text{km}}{\text{h}}$ are feasible, while $x = 1\,\frac{\text{km}}{\text{h}}$ is infeasible. We could also say that, literally, $x = 4\,\frac{\text{km}}{\text{h}}$ is a solution and $x = 1\,\frac{\text{km}}{\text{h}}$ is not a solution, but we rather use the terminology of feasible and infeasible solutions.

The way we distinguish feasible solutions of a model so that whether they are more or less suitable, is expressed in the model as an **objective** function, which is to be minimized (or maximized). An objective function - like constraints - evaluates an expression based on the model variables. The goal of optimization is to find a feasible solution for which the objective function is minimal (or maximal). A solution with this property is an **optimal solution** of the model. If a solution is optimal, then we can be sure that any other solutions are either infeasible, have an equal or worse objective value, or both. There can be multiple optimal solutions for the same model. In Problem 1, we have a very simple objective function $c(x) = x$, which is minimal at the solution of $x = 4\,\frac{\text{km}}{\text{h}}$. The optimal objective for the problem is therefore $4\,\frac{\text{km}}{\text{h}}$. Sometimes we refer to the objective value as the solution of an optimization problem. Therefore, we can say that the optimal solution of Problem 1 is $4\,\frac{\text{km}}{\text{h}}$.

It may happen that an optimization model does not have a feasible solution at all, hence there are no optimal solutions either. In this case, we say **the model is infeasible**. For example, the following modification of the problem (see Problem 2), the resulting System 5 would be infeasible: Alice is too far away and cannot go fast enough, she will miss the train by 3 min at least.

Problem 2.

Alice's train departures 75 minutes from now. She lives 8 km away from the station and wants to walk, and also needs 30 minutes to get ready. Also, she cannot run faster than $10\,\frac{\text{km}}{\text{h}}$ *in average. How fast shall she walk to the station to catch the train?*

$$\begin{aligned} \text{minimize}: \quad & x \\ \text{subject to}: \quad & 30\,\text{min} + \tfrac{8\,\text{km}}{x} \leq 75\,\text{min} \\ & x \leq 10\,\tfrac{\text{km}}{\text{h}} \end{aligned} \tag{5}$$

In rare cases, it may appear that a model is **unbounded**. This means that there are feasible solutions but none of them are optimal, because there are even better and better feasible solutions as well. For example, if the goal for Alice was to go as fast as possible to catch a train, instead of being as slow as possible, then neither $4\,\frac{\text{km}}{\text{h}}$, $10\,\frac{\text{km}}{\text{h}}$, $1000\,\frac{\text{km}}{\text{h}}$ or the speed of light would be optimal solutions, because there would always be a feasible solution with higher speed value. Of course, this does not make sense. Usually, when our model turns out to be unbounded, then we either made a mistake in solving it, or it does not describe reality well – maybe because it misses some real-world constraint we did not thought about. The latter was the case in our example, because in practice, Alice cannot go as fast as she wishes.

We can also call the set of feasible solutions of the model as the **search space**, emphasizing that our actual goal in optimization is to maximize or minimize an objective function value over the set of feasible solutions. Also, if all the variables together are treated as a mathematical vector, then each solution can be considered a point in that vector space. The set of all feasible solutions is a set of points in space that might have some special properties. For example, in LP problems, it is a convex polytope. Constraints (and bounds) are used to define the search space. Each constraints might exclude some feasible values from the search space that are no longer feasible anymore because of that constraint.

It may appear that there is a constraint that does not exclude solutions from the search space at all. We call those constraints **redundant**. Note that redundant constraints do not affect the solutions of the model, and hence the optimal solution, but may affect the solution procedure itself, either in a positive or a negative way. We may also refer to constraints as redundant if they do actually exclude some solutions, but only those which are not interesting for us (for example, because they cannot be optimal for some reason and therefore will not be reported by the solution algorithm anyways).

Now see a bit more elaborate example for an optimization problem.

Problem 3.

A company wants to design a new kind of chocolate in the form of a rectangular brick of homogeneous material. They want the brick to contain as much mass as possible, however, there are regulations on the faces of the brick, for transporting considerations. One of the faces must have an area not greater than $6\,\text{cm}^2$*, another face cannot be greater than* $8\,\text{cm}^2$*, and the largest face cannot be greater than* $12\,\text{cm}^2$*. What is the largest mass for a piece of chocolate?*

This example can be part of a business decision problem. The company wants to maximize its profit, but has to find a suitable solution for production. Operations Research helps in decision making, for example, with optimization models.

Now it is not even straightforward to define the variables themselves. Though not even mentioned in the problem text, the three edges of the brick, say x, y and z are adequate, for the following reasons. From these we can easily express the other quantities the problem specification refers to: xy, xz and yz are the faces, and xyz is for the total volume which is to be maximized. Note that if the chocolate is homogeneous, then volume is proportional to mass, so we may consider maximizing volume instead of mass, they are equivalent. Second, x, y and z are independent of each other: the selection of x does not directly affect the selection of y, as all possible combinations can yield a brick. If they were not independent, additional constraints must have been formalized.

The most important thing is that if x, y and z are given, then the solution to the real-world problem is well-defined. That means, we can exactly tell whether the constraints are satisfied or not, and what the objective value is. This is the most important requirement for a set of variables to be adequate for a model. Now let us do this work: express the three constraints for the faces and the objective of the volume to formulate the optimization model. Note that x, y and z should also be positive, but these bounds will be implicitly enforced by an optimal solution of the model.

$$\begin{aligned} \text{maximize :} \quad & xyz \\ \text{subject to :} \quad & xy \le 6\,\text{cm}^2 \\ & xz \le 8\,\text{cm}^2 \\ & yz \le 12\,\text{cm}^2 \\ & x, y, z \ge 0 \end{aligned} \tag{6}$$

There are many feasible solutions for this model, for example $x = y = z = 2\,\text{cm}$ is a feasible one, but $x = y = z = 2.5\,\text{cm}$ is an infeasible one. There is a single optimal solution, which is $x = 2\,\text{cm}$, $y = 3\,\text{cm}$ and $z = 4\,\text{cm}$, the total volume in this case is $24\,\text{cm}^3$.

The way of finding this optimal solution and undoubtedly prove that it is indeed optimal, is not trivial, and is out of scope of this Tutorial. There is a hint, if the reader liked to solve it: $(xyz)^2 = xy \cdot xz \cdot yz$.

2.4 Mathematical programming

We have seen two very simple optimization problems so far, however, for the latter one we would have been in trouble if we had to actually solve it and prove that our solution is indeed optimal.

However, the point of **mathematical programming** is that we are not the ones who actually solve the models. We are the ones who translate the real-world problem to an optimization model, then usually a software designed generally for solving models is doing the dirty job for us. Finally, we only need to verify that the solution of the model is actually a suitable solution to the real-world problem.

A mathematical programming model consists of the model variables that represent our freedom of choosing solutions, the constraints that express the requirements for a solution to be feasible, and correspondingly, suitable in the real-world problem, and finally the objective function, which determines the criteria of finding the best among the feasible solutions.

Although the general idea is that we are only formulating a model, we still need to take care about what class of model we choose. The main reason is that it greatly affects what kind of solution methods we can use. Solving a model from a more special class in contrast with a more general one can be drastically cheaper in terms of computational effort. We explicitly mention two model classes that are of great importance in Operations Research.

In **Linear Programming** (LP) models, the variables can take arbitrary real values. The constraints are equations or non-strict inequalities of **linear expressions** of the variables. A linear expression is the sum of some of the variables, each multiplied by a constant. Linear expressions cannot contain multiplication, division of the variables with each other, or any other mathematical functions or operations: only addition, subtraction, and multiplication by a constant is allowed. The objective function itself must be a linear expression of the variables, the minimization or maximization of which is the optimization goal. The following is an example of an LP problem.

$$\begin{aligned} \text{minimize :} \quad & 3x + 4y - z \\ \text{subject to :} \quad & 2x + 5 = z + 2 \\ & x - y \le 0 \\ & x + y + z \le 2\,(x + y) \\ & x, y, z \ge 0 \end{aligned} \tag{7}$$

One important thing about LP models is that only non-strict constraints are allowed ($\le$, $\ge$ or $=$), but strict constraints ($<$, $>$) are not. Also note that Problem 1 about Alice is formulated as an

LP model, while Problem 3 about the chocolate bricks is not LP, because it contains a product of variables, which is a nonlinear term.

An LP model is solvable in polynomial time in the number of constraints and variables. That means, large models can be formulated and solved in acceptable time with a modern computer. LP problems have several interesting properties that make them solvable with efficient algorithms, for example, the Simplex Method (see Chapter 8). One of such properties is that the set of feasible solutions in space is always a closed, convex set.

In **Mixed-Integer Linear Programming** models, we may additionally restrict some variables to only take integer values in feasible solutions, contrary to LP problems. This difference makes solving generally the wider class of MILP models an NP-complete problem in the number of integer variables. That means, if there are many integer variables in an MILP model, then we generally cannot expect a complete solution in acceptable computational time. However, the number of applications the class of MILP models can cover is surprisingly more extended than for LP problems, making MILP models a popular method for solving complex optimization problems. There are effective software tools utilizing heuristic algorithms that usually solve not too complex MILP models very fast.

In this manual, we focus on MILP and LP models. In both cases, there are only linear expressions of the variables appearing in the formulations. There are other problem classes like Nonlinear Programming (NLP), Mixed-Integer Nonlinear Programming (MINLP), with corresponding solvers, but these are not covered here.

Chapter 3

GNU MathProg

This chapter aims at briefly introducing GNU MathProg and how it can be used to implement and solve mathematical programming models in the LP or MILP class. A short example problem is presented with implementation, solution and results, with some of the most frequently used features of the language shown.

3.1 Prerequisites for programming

GNU MathProg, also referred to as the GNU Mathematical Programming Language, or GMPL, is a programming language for designing and solving LP and MILP. The GNU Linear Programming Kit, GLPK provides free software to both parsing implemented models and solving them until the optimal solution is reported. GLPK is available under the General Public License version 3.0 or later [6].

There are other software tools for solving MILP models, some are available with a free license like CBC, or lpsolve. These mentioned can be much faster than GLPK. Commercial software can even be better, some of them are available through academic license. However, we selected GNU MathProg and solver `glpsol` from GLPK because it is relatively easy to use and includes both language and model solving tools.

Installing GLPK depends on the operating system. On Linux, we can head to the official website of GLPK, where installation packages, source codes and even sample GNU MathProg problems can be found. On some distributions, installation can be done with a single command as follows.

```
sudo apt-get install glpk-utils
```

If installation is complete, we shall have the program `glpsol` available in the command line. Throughout this manual, we will use this tool to parse and solve models, in command line.

On Windows, there is a possibility to obtain the command line solver as well. However, a more convenient option is a desktop application called GUSEK [9], which is a simple IDE with text editor for the GNU MathProg programming language. The solver `glpsol` is available with all of its functionalities there. With GUSEK, no command line is needed, although navigating between multiple files requires some attention.

Reference manual for the GNU MathProg modeling language with usages of the `glpsol` software is publicly available, see the references [1].

3.2 „Hello World!" program

After successful installation, we can try the program with a very simple LP problem. This will be our first GNU MathProg program, like a „Hello World!".

```
var x >= 0;
var y >= 0;
var z >= 0;

s.t. Con1: x + y <= 3;
s.t. Con2: x + z <= 5;
s.t. Con3: y + z <= 7;

maximize Sum: x + y + z;

solve;

printf "Optimal (max) sum is: %g.\n", Sum;
printf "x = %g\n", x;
printf "y = %g\n", y;
printf "z = %g\n", z;

end;
```

GNU MathProg is designed to be easy to read, and the mathematical meaning to be easy to understood. The language consists of commands ending with a semicolon (`;`), these are called **statements**. There are six distinct statements in this file: `var`, `s.t.`, `maximize`, `solve`, `printf` and `end`. The meanings of most are self-explanatory, as we will see.

```
var x >= 0;
var y >= 0;
var z >= 0;
```

First of all, this is an LP problem, all three **variables** x, y and z can take real values. Also, they have **bounds** defined: each of them must be nonnegative. Note that this is the most common bound for variables.

```
s.t. Con1: x + y <= 3;
s.t. Con2: x + z <= 5;
s.t. Con3: y + z <= 7;
```

In the problem, there are three constraints, apart from the bounds, which are treated differently in the language. The constraints are named `Con1`, `Con2`, `Con3`, respectively. The names are preceded by „`s.t.`", but it has several aliases, it can be „`subject to`", or can be omitted at all. The constraints state that the sum of any two of the three variables cannot be greater than given constants, which are 3, 5 and 7, respectively.

```
maximize Sum: x + y + z;
```

The goal of the optimization is to maximize the objective entitled as „Sum", which is, of course, the sum of the three variables. There can be at most one `maximize` or `minimize` statement in a model file.

At this point, the LP problem is well defined. This is a relatively easy linear problem, however, its optimal solution might be not obvious for first sight.

It shall be noted that the objective function can be omitted at all from a GNU MathProg model formulation. In this case, the only task is to find any feasible solution for the problem.

```
solve;

printf "Optimal (max) sum is: %g.\n", Sum;
printf "x = %g\n", x;
printf "y = %g\n", y;
printf "z = %g\n", z;

end;
```

There is a `solve` statement in the file. This denotes the point where the solver software should finish reading all the data, then constructing and solving the model. After the `solve` statement, there may still be other statements in the file, to report valuable information about our solution. In this case, the objective value and our three variables are printed with their values obtained by the optimization procedure. The `printf` statement works similarly to the `printf()` function from the C programming language, but has less supported format specifiers. We prefer the `%g` format, because it can print both integers and fractional values in a short form. In some cases, a fixed precision number is more adequate, like `%.3f`.

An `end` statement denotes the end of the model description, although it is optional.

We can use any text editor we like, but some of them have syntax highlight options. GUSEK on Windows has its own text editor which is specifically designed to modeling. On Linux, we can use `gedit`. It does not have a syntax highlight for GNU MathProg by default, but one can be obtained from the internet.

Let us name this exact file as `helloworld.mod`. On some systems, the file extension `.mod` can be recognized as a video format by the file browser. This causes the „double-click" opening method to try to open the file as a video instead of a text file. For this reason, it is recommended to set the opening method of the file to the text editor we prefer, for example `gedit`. Alternatively, we can simply use a different file extension, like `.m`, or simply `.txt`, it does not matter from the viewpoint of `glpsol`, although it may matter if we use some IDE for programming like GUSEK.

File `helloworld.mod` is a **model file**, because it contains the business logic of the problem we want to solve. It is alone enough for defining a problem. Later on, we will split our problem formulations to a single model file and one or more **data files**. The model file encapsulates the logic of the problem, while the data file(s) provide the actual data the problem shall be solved with. This will be useful, because if the data of the real-world problem changes, we do not need to tamper with the model itself. In other words, a user who wants to solve a model for a particular problem does not need to know the model, only a data file must be implemented correctly, which is much easier and less error-prone.

Use `glpsol` to solve the model file with the following command.

```
glpsol -m helloworld.mod
```

Something similar to the following output is obtained.

```
GLPSOL: GLPK LP/MIP Solver, v4.65
Parameter(s) specified in the command line:
 -m helloworld.mod
```

```
Reading model section from helloworld.mod...
20 lines were read
Generating Con1...
Generating Con2...
Generating Con3...
Generating Sum...
Model has been successfully generated
GLPK Simplex Optimizer, v4.65
4 rows, 3 columns, 9 non-zeros
Preprocessing...
3 rows, 3 columns, 6 non-zeros
Scaling...
 A: min|aij| =  1.000e+00  max|aij| =  1.000e+00  ratio =  1.000e+00
Problem data seem to be well scaled
Constructing initial basis...
Size of triangular part is 3
*     0: obj =  -0.000000000e+00 inf =   0.000e+00 (3)
*     3: obj =   7.500000000e+00 inf =   0.000e+00 (0)
OPTIMAL LP SOLUTION FOUND
Time used:   0.0 secs
Memory used: 0.1 Mb (102283 bytes)
Optimal (max) sum is: 7.5.
x = 0.5
y = 2.5
z = 4.5
Model has been successfully processed
```

This is a small problem, therefore the result is almost instantaneous. Let us observe what information can be read from the output.

```
GLPSOL: GLPK LP/MIP Solver, v4.65
Parameter(s) specified in the command line:
 -m helloworld.mod
Reading model section from helloworld.mod...
18 lines were read
Generating Con1...
Generating Con2...
Generating Con3...
Generating Sum...
Model has been successfully generated
```

The first section of the output is printed during the model generation procedure. Note that if the model has bad syntax, it is remarked here. Only the first error is displayed. It can also be observed that the constraints and objective are the model elements that must be „generated”.

```
GLPK Simplex Optimizer, v4.65
4 rows, 3 columns, 9 non-zeros
Preprocessing...
3 rows, 3 columns, 6 non-zeros
Scaling...
```

```
 A: min|aij| =  1.000e+00  max|aij| =  1.000e+00  ratio =  1.000e+00
Problem data seem to be well scaled
Constructing initial basis...
Size of triangular part is 3
*     0: obj =  -0.000000000e+00 inf =   0.000e+00 (3)
*     3: obj =   7.500000000e+00 inf =   0.000e+00 (0)
OPTIMAL LP SOLUTION FOUND
Time used:   0.0 secs
Memory used: 0.1 Mb (102283 bytes)
```

The solution procedure of an LP problem from the constraints themselves to the final, optimal solution is itself an interesting subject of Operations Research. However, we do not focus on the underlying solution algorithms themselves. However, basic knowledge about it would help us debugging in the future. First of all, all LP problems are represented in a matrix of coefficients, the rows of which are corresponding to constraints (and the objective), the columns corresponding to variables, and the entries of the matrix being the coefficients of a given variable in a given constraint. On this matrix, some preprocessing steps are performed, and then the main algorithm is launched.

There are two rows in this output which show a „current" result. The first one shows an objective of 0, the second shows the objective 7.5 which is actually the final optimal solution. The row „OPTIMAL LP SOLUTION FOUND" denotes that the solver succeeded in solving the problem, and is now aware of the optimal solution and the corresponding values of the variables. If there are multiple optimal solutions, one of them is chosen.

With more complex problems, solution can take a considerable amount of time and current best solutions are regularly displayed. The solution procedure may be ended before arriving to the optimal solution, in which case the best feasible solution found so far is reported. This happens, for example, when a time limit is set to `glpsol` and it is exceeded during the procedure.

```
Optimal (max) sum is: 7.5.
x = 0.5
y = 2.5
z = 4.5
Model has been successfully processed
```

Finally, we can see the result of the `printf` statements that we added at the end. We can see that the optimal solution is $x = \frac{1}{2} = 0.5$, $y = \frac{5}{2} = 2.5$, $z = \frac{9}{2} = 4.5$, and then the objective function is $x + y + z = \frac{15}{2} = 7.5$. This result means we cannot choose values for the variables to obtain a larger sum, unless at least one of the constraints (or bounds) are violated. Note that `printf` statements are also valid before the `solve` statement, that is, during the model construction procedure. However, before the `solve` statement, variables and the objective are not available, as they had not been calculated by that point. The final `Model has been successfully processed` message hints us that the solver call was successful.

We shall also note that `glpsol` command line tool has many other options available. For example, we can ourselves print an automated solution file with the `-o` option. We can also save the output of the program with the `--log` option. Also, the constructed model can be exported to formats that other MILP solvers can use. For example, with the `--wlp` option, we can export to `CPLEX-LP` matrix format, which is supported by many other solver software. Perhaps we do want to use `glpsol` to solve the model, we only need it to translate the model to some other format. Then, the `--check` option ensures that the model is only parsed, but not solved. It is also possible to manipulate the solution procedure in many ways.

Now that we have successfully formulated, solved and analyzed the output for our first LP problem implemented in GNU MathProg, let us make a slight change in the model file. Add the `integer` restriction to all three of the variables, as follows. The modified file is now saved as `helloworld-int.mod`.

```
var x >= 0, integer;
var y >= 0, integer;
var z >= 0, integer;
```

From now, this is no more an LP model, but an MILP model, because some (in particular, all) of the variables must take integer values. Solving the model with `glpsol` is the same.

```
glpsol -m helloworld-int.mod
```

The solver identifies the model as MILP, and there are some changes in the generated output.

```
Solving LP relaxation...
GLPK Simplex Optimizer, v4.65
3 rows, 3 columns, 6 non-zeros
*     0: obj =  -0.000000000e+00 inf =   0.000e+00 (3)
*     4: obj =   7.500000000e+00 inf =   0.000e+00 (0)
OPTIMAL LP SOLUTION FOUND
Integer optimization begins...
Long-step dual simplex will be used
+     4: mip =     not found yet <=              +inf        (1; 0)
+     6: >>>>>   7.000000000e+00 <=   7.000000000e+00   0.0% (2; 0)
+     6: mip =   7.000000000e+00 <=     tree is empty   0.0% (0; 3)
INTEGER OPTIMAL SOLUTION FOUND
Time used:   0.0 secs
Memory used: 0.1 Mb (124952 bytes)
Optimal (max) sum is: 7.
x = 1
y = 2
z = 4
Model has been successfully processed
```

We can observe that a so-called **LP relaxation** is solved first, the output for which seems the be the same as the output for our initial LP problem. Actually, this is the nature of the solution algorithm, it is first solved as if all the integer restrictions were removed, as an LP problem. Then, the actual MILP model is solved, and we find that the optimal solution is only 7.0 instead of the optimal solution of the LP, which was 7.5.

This is not surprising. The only difference between the two models is that the integer problem is more restrictive about the variables. That means, any solution to the MILP is a solution to the LP. But the opposite is not true, as the optimal solution of the LP, $x = 0.5$, $y = 2.5$, $z = 4.5$ was eliminated by the integer restrictions. The LP relaxation is an important concept in solving MILP models - for example, it can be used as a good initial solution, trying to make all the variables to obtain integer variables. Also, if the integer variables all turn out to have integer values in the LP relaxation, then it is guaranteed to be an optimal solution for the MILP as well, because the MILP is the more restricted.

In this model, the solver needed a little extra work for the MILP, and found the solution of $x = 1$, $y = 2$ and $z = 4$. Note that it is an optimal solution, but not the only one. $x = 0$, $y = 3$, $z = 4$, and $x = 0$, $y = 2$, $z = 5$ are also feasible and optimal solutions with the same objective value of 7.0.

As we could suspect, the algorithmic procedure behind the MILP can be substantially more complex than that of an LP. However, the difference in the formulation is only the integer restrictions of the variables.

We should keep in mind that solvers can handle a large number of variables and constraints of an LP model, but usually they can only handle a limited number of integer variables of an MILP model. The exact limit for integer variables strongly depends on the model itself, it can range from dozens to thousands.

Chapter 4

Equation systems

In this chapter, basic capabilities of the GNU MathProg language are presented on an example modeling problem, which is the general solution of linear equation systems. The aim is to navigate from the initial point of a straightforward implementation to the usage of indexing and separated model and data sections.

With this knowledge, maintenance of code, and solution of the model with different instances (different data) become very easy, provided that first the model is correctly implemented. This is the standard way we intend to implement models in the future.

Note that solving linear systems of equations is considered a routine task. There are plenty of capable software tools, either standalone or embedded. A well-known algorithm is the Gaussian elimination [10]. But we are now interested in not the solution algorithms, but modeling. Particularly, GNU MathProg implementation techniques, in this chapter.

4.1 Example system

Let us start with the solution of a small system of equations, now in GNU MathProg.

Problem 4.
Solve the following system of equations in the real domain.

$$\begin{aligned} 2\,(x-y) &= -5 \\ y-z &= w-5 \\ 1 &= y+z \\ x+2w &= 7y-x \end{aligned} \tag{8}$$

Although it is not explicitly mentioned in the problem description, there are four variables, x, y, z, w, and the real domain means that all of these can take arbitrary real values, independently. Our goal is to find values for these four variables so that all four equations are true.

If a complete analysis about the problem was in question, then the problem would be not only to find a single solution, but to find all of them - and if there are infinitely many solutions, then characterize this infinite set, moreover, also prove that there are no other solutions. However, for the sake of simplicity, now we only want to find any single solution to the system of equations and we do not want to decide whether there is another or not.

This is therefore a **feasibility problem**, it means that only a single feasible solution is to be found. There is no need to objective function for which we want to optimize, because there are no better or worse solutions.

It can be observed that Problem 4 describes a *linear* system of equations. This is apparent after we arrange the equations into the following format.

$$\begin{array}{rrrrl} 2x & -2y & & & = -5 \\ & y & -z & -w & = -5 \\ & y & +z & & = 1 \\ 2x & -7y & & +2w & = 0 \end{array} \tag{9}$$

The point is that this arrangement can be obtained from the original system by simplifying each equation alone: removing the parentheses, adding and merging terms at both sides. In this way, the arranged system is equivalent to the original. On the **left-hand side (LHS)** of the equation, there is a linear expression of the variables (each multiplied by a constant, then added), and on the **right-hand side (RHS)**, there is a constant, in each equation. Therefore all equations are linear. If we further add zero terms and emphasize coefficients at each column of the arranged format, we get the following, equivalent form of the same system.

$$\begin{array}{rcrcrcrl} (+2)\cdot x & + & (-2)\cdot y & + & 0\cdot z & + & 0\cdot w & = -5 \\ 0\cdot x & + & (+1)\cdot y & + & (-1)\cdot z & + & (-1)\cdot w & = -5 \\ 0\cdot x & + & (+1)\cdot y & + & (+1)\cdot z & + & 0\cdot w & = 1 \\ (+2)\cdot x & + & (-7)\cdot y & + & 0\cdot z & + & (+2)\cdot w & = 0 \end{array} \tag{10}$$

Although this representation is less readable, it represents more generally what a system of linear equations actually mean. For each equation, there is a constant right-hand side, and a linear expression at the left-hand side, for which we only want to know the coefficients for each variable. So this whole problem can be represented in a matrix of coefficients, for which the rows correspond to solutions, and columns correspond to variables – except the rightmost column, which corresponds to the right-hand side.

This is very close to what we call the **standard form** of a Linear Programming problem. The standard form also includes ≥ 0 bounds for all variables, consists of $\leq$ inequalities instead of equations, and of course, has a linear objective function. This matrix-like form is important because it inspires solution algorithms. However, we are now not going into the details of solution algorithms, neither in case of system of equations nor in LP problems.

Instead, the representation shown in Equation system 10 will be useful to understand the general scheme behind systems of linear equations, and to implement a separated GNU MathProg MILP model, for which a data section describing an arbitrary system of linear equations can be provided and solved.

Now let us see how the code looks like in GNU MathProg. First, the most straightforward implementation is shown, without arrangement of the equations.

```
var x;
var y;
var z;
var w;

s.t. Eq1: 2 * (x - y) = -5;
s.t. Eq2: y - z = w - 5;
s.t. Eq3: 1 = y + z;
s.t. Eq4: x + 2 * w = 7 * y - x;

solve;
```

```
printf "x = %g\n", x;
printf "y = %g\n", y;
printf "z = %g\n", z;
printf "w = %g\n", w;

end;
```

As we can see here, GNU MathProg allows usage of parentheses and basic operations, provided that the constraints, now named `Eq1`, `Eq2`, `Eq3` and `Eq4`, simplify into an equality (or non-strict inequality) between two linear expressions.

If we solve the above model file, then the solution $x = 0.5$, $y = 3$, $z = -2$ and $w = 10$ is obtained. This solution is unique, so all solution methods shall end up with this exact result. We can also manually check that all four equations are satisfied with the resulting values of the variables. The code itself helps us by printing the values of the four variables to the solver output.

Now, if the aligned version is to be implemented, we can get the following.

```
s.t. Eq1: 2 * x + (-2) * y +    0 * z +    0 * w = -5;
s.t. Eq2: 0 * x +    1 * y + (-1) * z + (-1) * w = -5;
s.t. Eq3: 0 * x +    1 * y +    1 * z +    0 * w = 1;
s.t. Eq4: 2 * x + (-7) * y +    0 * z +    2 * w = 0;
```

As we can see, white space characters can be freely used in GNU MathProg to help with readability. Also, zero coefficient terms can be eliminated from the expressions and padded with whitespace instead. The rest of the model file can remain unchanged. The result when solving this arranged model file shall be the same.

4.2 Modifications in code

Now that we have a working model file solving a particular system of linear equations, let us try to solve another one, which is obtained by slight modifications from the original one. The problem description is the following.

Problem 5.

Take the equations defined by Equation system 4 as initial, within its aligned form described in either Equation system 9 or 10. Then, obtain another system by applying the following modifications.

1. *Change the coefficient of x everywhere from* 2 *to* 4.
2. *Add z in* `Eq1` *with a coefficient of* 3.
3. *Delete y from* `Eq2` *(zeroize its coefficient).*
4. *Change the RHS of* `Eq3` *to* 1.5.
5. *Add a fifth variable v, with no bounds.*
6. *Add the equation $x + 3v + 0.5 = -2z$ as* `Eq5`.
7. *Introduce v in* `Eq4` *with a coefficient of 5.*
8. *Print out the newly introduced variable v after the* `solve` *statement as the others.*

These modifications can be done one-by-one in a straightforward way until the following model is obtained.

```
var x;
var y;
var z;
var w;
var v;

s.t. Eq1: 4 * x + (-2) * y +    3 * z +    0 * w +   0 * v = -5;
s.t. Eq2: 0 * x +    0 * y + (-1) * z + (-1) * w +   0 * v = -5;
s.t. Eq3: 0 * x +    1 * y +    1 * z +    0 * w +   0 * v = 1.5;
s.t. Eq4: 4 * x + (-7) * y +    0 * z +    2 * w +   5 * v = 0;
s.t. Eq5: 1 * x +    0 * y +    2 * z +    0 * w +   3 * v = -0.5;

solve;

printf "x = %g\n", x;
printf "y = %g\n", y;
printf "z = %g\n", z;
printf "w = %g\n", w;
printf "v = %g\n", v;

end;
```

The solution of the new system is $x = 2$, $y = 3.5$, $z = -2$, $w = 7$, $v = 0.5$. Although not proven here, but this solution is unique.

Now observe which kind of modifications we have to implement in code to obtain the results. The first four tasks were just replacing, introducing and erasing coefficients from the corresponding equations. In these cases, one must find the corresponding constraints, the corresponding variable in it and apply the changes.

However, the introduction of the new variable requires some more work besides declaring v as a variable with no bounds. If we wanted to maintain the arrangement of the equations, then all must be expanded with a single column for coefficients of v. These coefficients are initially zero, for not being used. Then, in `Eq4` and `Eq5`, we can insert a nonzero coefficient for v afterwards. We also need to manually modify the printing code after the `solve` statement to involve v.

Insertion of the new equation `Eq5` would also require some more work. Again, if we wanted to keep the code easy to read and maintain the arrangement, the new code line for `Eq5` must be formatted accordingly.

Although this was still a small problem, we can feel that modification of a system of equations to accommodate another problem might be a bit frustrating. It gets worse if the number of variables and equations increase – note that several thousands of both are still considered a problem with reasonable size.

One problem is that the model code includes some redundancy – we have to maintain arrangement to preserve readability, although it is not possible for very large problems. However, the greatest drawback is that we have to understand and tamper with model code – although we only need to change the actual problem itself, which is still a system of linear equations.

It would be wise to somehow separate the modeling logic of systems of linear equations, and actual data of the linear equations to be solved (that is, the number of variables, equations, coefficients and right-hand side constants). This is exactly what we will do in the next sections. Afterwards, what we only need is to alter the data, and the model can remain unchanged, which helps us in reusing code in the future, and implementation of problems with large data.

4.3 Parameters

The first thing we do to separate problem data from the model logic is the introduction of **parameters**. This case, the modified problem is to be implemented. We only change the code, not the underlying problem.

Parameters can be introduced with the `param` statement. It is similar in syntax to the `var` statement for introducing variables. However, the `param` statement defines constant values that can be used in the model formulation. Parameters are by default numeric values, and are used here as such, but note that they can also represent symbolic values (like strings).

The fundamental difference is that variables are the values to be found by the optimization procedure, to satisfy all constraints and optimize a given objective function. For the implementation point of view, the values of variables are only available after the `solve` statement. Parameters, on the other hand, are constant values that are initially given, and are typically used to formulate the model itself.

Note that parameters defined by the `param` statement, similarly to variables defined in the `var` statement, must have a unique name, and they can only be used in the model file *after* definition. Hence, in any expression, we can only refer to parameters which are defined earlier in the code. For this reason, parameter definitions typically precede constraint and even variable definitions.

First, we only introduce parameters for the right-hand values of the equations, as follows.

```
var x;
var y;
var z;
var w;
var v;

param Rhs1 := -5;
param Rhs2 := -5;
param Rhs3 := 1.5;
param Rhs4 := 0;
param Rhs5 := -0.5;

s.t. Eq1: 4 * x + (-2) * y +    3 * z +    0 * w +   0 * v = Rhs1;
s.t. Eq2: 0 * x +    0 * y + (-1) * z + (-1) * w +   0 * v = Rhs2;
s.t. Eq3: 0 * x +    1 * y +    1 * z +    0 * w +   0 * v = Rhs3;
s.t. Eq4: 4 * x + (-7) * y +    0 * z +    2 * w +   5 * v = Rhs4;
s.t. Eq5: 1 * x +    0 * y +    2 * z +    0 * w +   3 * v = Rhs5;

solve;

printf "x = %g\n", x;
printf "y = %g\n", y;
printf "z = %g\n", z;
printf "w = %g\n", w;
printf "v = %g\n", v;

end;
```

From now on, if we need to modify the RHS of an equation, we do not need to modify the code of the constraint describing that equation. Instead, we only need to replace the definition of the given RHS parameter.

In the same manner, we can also introduce parameters for the coefficients of the equations. Because there are coefficients for all 5 equations and all 5 variables, a total of 25 additional parameters are needed.

```
var x;
var y;
var z;
var w;
var v;

param Rhs1 := -5;
param Rhs2 := -5;
param Rhs3 := 1.5;
param Rhs4 := 0;
param Rhs5 := -0.5;

param c1x:=4; param c1y:=-2; param c1z:= 3; param c1w:= 0; param c1v:= 0;
param c2x:=0; param c2y:= 0; param c2z:=-1; param c2w:=-1; param c2v:= 0;
param c3x:=0; param c3y:= 1; param c3z:= 1; param c3w:= 0; param c3v:= 0;
param c4x:=4; param c4y:=-7; param c4z:= 0; param c4w:= 2; param c4v:= 5;
param c5x:=1; param c5y:= 0; param c5z:= 2; param c5w:= 0; param c5v:= 3;

s.t. Eq1: c1x * x + c1y * y + c1z * z + c1w * w + c1v * v = Rhs1;
s.t. Eq2: c2x * x + c2y * y + c2z * z + c2w * w + c2v * v = Rhs2;
s.t. Eq3: c3x * x + c3y * y + c3z * z + c3w * w + c3v * v = Rhs3;
s.t. Eq4: c4x * x + c4y * y + c4z * z + c4w * w + c4v * v = Rhs4;
s.t. Eq5: c5x * x + c5y * y + c5z * z + c5w * w + c5v * v = Rhs5;

solve;

printf "x = %g\n", x;
printf "y = %g\n", y;
printf "z = %g\n", z;
printf "w = %g\n", w;
printf "v = %g\n", v;

end;
```

Note that `param` statements can be put in a single line for the sake of readability, and to preserve the arrangement of the equations. After the total of 30 parameters are introduced, the implementation of the equations can be considered final. With the sole modification of the parameter statements in the model file, we can solve arbitrary systems of linear equations with 5 variables and 5 equations.

Note that the solution of any of these two models are exactly the same as before, as the system of equations itself did not change, only its implementation.

4.4 Data blocks

We had separated the coefficients and RHS values from the equations they are in, but still, we need to modify the `param` statements in the model file to alter the problem. We used a single model file each time so far. Now we introduce the concept of the **model section** and the **data section**, with which we can separate the model logic with the problem data entirely.

Parameter values in GNU MathProg can be determined in two ways.

- A parameter value is explicitly given where it is defined: as a constant value, or some expression of other constant values, other parameters, and built-in functions and operators of the GNU MathProg language.
- A parameter value in not given in the model section, but in a separate data section.

Now let us see how we can modify the model file to obtain a separate model section and data section. In case where only the RHS parameters were introduced, the complete separation is the following.

```
var x;
var y;
var z;
var w;
var v;

param Rhs1;
param Rhs2;
param Rhs3;
param Rhs4;
param Rhs5;

s.t. Eq1: 4 * x + (-2) * y +    3 * z +    0 * w +   0 * v = Rhs1;
s.t. Eq2: 0 * x +    0 * y + (-1) * z + (-1) * w +   0 * v = Rhs2;
s.t. Eq3: 0 * x +    1 * y +    1 * z +    0 * w +   0 * v = Rhs3;
s.t. Eq4: 4 * x + (-7) * y +    0 * z +    2 * w +   5 * v = Rhs4;
s.t. Eq5: 1 * x +    0 * y +    2 * z +    0 * w +   3 * v = Rhs5;

solve;

printf "x = %g\n", x;
printf "y = %g\n", y;
printf "z = %g\n", z;
printf "w = %g\n", w;
printf "v = %g\n", v;

data;

param Rhs1 := -5;
param Rhs2 := -5;
param Rhs3 := 1.5;
param Rhs4 := 0;
param Rhs5 := -0.5;

end;
```

We can observe that the parameters are only defined, without a value given in the model section. Then, after the model construction, and the post-processing procedures (here, printing the variables out), there comes a `data` statement. There can be a single `data` statement at the end of a model file, which denotes the end of the model section and the start of the data section. The data section contains the actual values of parameters which were defined in the model without a value provided there.

This can be done for the full parameterization example as well.

```
var x;
var y;
var z;
var w;
var v;

param Rhs1;
param Rhs2;
param Rhs3;
param Rhs4;
param Rhs5;

param c1x; param c1y; param c1z; param c1w; param c1v;
param c2x; param c2y; param c2z; param c2w; param c2v;
param c3x; param c3y; param c3z; param c3w; param c3v;
param c4x; param c4y; param c4z; param c4w; param c4v;
param c5x; param c5y; param c5z; param c5w; param c5v;

s.t. Eq1: c1x * x + c1y * y + c1z * z + c1w * w + c1v * v = Rhs1;
s.t. Eq2: c2x * x + c2y * y + c2z * z + c2w * w + c2v * v = Rhs2;
s.t. Eq3: c3x * x + c3y * y + c3z * z + c3w * w + c3v * v = Rhs3;
s.t. Eq4: c4x * x + c4y * y + c4z * z + c4w * w + c4v * v = Rhs4;
s.t. Eq5: c5x * x + c5y * y + c5z * z + c5w * w + c5v * v = Rhs5;

solve;

printf "x = %g\n", x;
printf "y = %g\n", y;
printf "z = %g\n", z;
printf "w = %g\n", w;
printf "v = %g\n", v;

data;

param Rhs1 := -5;
param Rhs2 := -5;
param Rhs3 := 1.5;
param Rhs4 := 0;
param Rhs5 := -0.5;

param c1x:=4; param c1y:=-2; param c1z:= 3; param c1w:= 0; param c1v:= 0;
param c2x:=0; param c2y:= 0; param c2z:=-1; param c2w:=-1; param c2v:= 0;
param c3x:=0; param c3y:= 1; param c3z:= 1; param c3w:= 0; param c3v:= 0;
param c4x:=4; param c4y:=-7; param c4z:= 0; param c4w:= 2; param c4v:= 5;
param c5x:=1; param c5y:= 0; param c5z:= 2; param c5w:= 0; param c5v:= 3;

end;
```

The solution obtained by solving these model files is exactly the same in all the parameterization examples, because the system of linear equations is still the same. But now, if one wants to solve an arbitrary system of 5 linear equations on 5 variables, then only the data section of this model file must be modified accordingly.

There are more complex data sections required for more complex models. However, the syntax for data sections is generally more permissive than in the model section. That means, even if we only want to implement a single model and only solve it with a single problem, it is still a good choice to write its data to a separate data section rather than hard-coding parameter values into the model section. The more permissive syntax also makes shorter and more readable data representation possible.

One thing that makes data sections useful is that they can be implemented in a **separate data file**. A data file is a data section written in a separate file. Its content shall be simply all the code between the `data` and `end` statements. Note that both statements are optional in this case, but increase readability.

The model file shall only contain the full model section and no data sections. Suppose that we had a model file named `equations.mod`, and we write the data section of the actual system of equations into the `eq-example.dat` file. Then, solution with the `glpsol` command line tool can be done with the following command.

```
glpsol -m equations.mod -d eq-example.dat
```

Unlike in the model section where referring to model elements defined afterwards is illegal, the statements in the data section can be in arbitrary order. Moreover, multiple data files are allowed to be solved with a single model file, therefore we can further separate problem data. It does not mean that we are solving different problems at the same time, but a single problem instance where its different parameters are separated into different data files.

For example, if we have n different variations for the value of a given parameter, and k different variables for the value of another parameter, then we can implement these as $n + k$ data files. We actually have nk different problem instances, all can be solved with the same single model file, by providing `glpsol` the corresponding two variations for the two parameters.

One important property of parameters in GNU MathProg that should be kept in mind is that *parameter values are only calculated if they are actually required*, for example to define model elements (like constraints, variable bounds, objectives), or printing them out, or, if they are required in the calculation of other parameters which are themselves required. This might cause, for example that an erroneously defined parameter value is only reported as a modeling error when the constraint it first appears in is reached in the model file. Note that, basic syntax check is done by `glpsol` even in this case, just the substitution of parameter values is „lazy".

One particular example for this is when we do not give a value for a parameter, and use it afterwards. It is legal in GNU MathProg to define a parameter without a value – for example, when we want to give value to it in the data section after the whole model section, or even in a separate file. If the parameter is not actually used at all in the model, then it is not an error. However, if the parameter is used and there is no value at definition, or any data sections, an error is generated. To demonstrate this, erase the `param c3z := 1;` statement from the data section of the model and try to solve it with `glpsol` as before. The following error message is shown.

```
Generating Eq1...
Generating Eq2...
Generating Eq3...
eqsystem-data-blocks-full.mod:21: no value for c3z
MathProg model processing error
```

We can see that the first two equations `Eq1` and `Eq2` were generated successfully. However, `Eq3` utilized the parameter `c3z` in its definition, but there was no value given for that particular parameter: neither in the model section, nor in the data section (or any data sections). The model could not be generated and solved.

Note that, it is also illegal to provide a parameter value twice (in the model and in a data section, or in any two data sections). However, there is a way we can „optionally" provide parameters values in data files, by providing **default values** at the model file. More on that later.

4.5 Indexing

At this point, we are able to completely separate the model logic and the problem data – at least for the case of systems of linear equations. However, one might argue that it did not actually help much. The model section including all the constraints are not needed to be touched, but the parameter definition part of the model, and also their values in a data section, require even more code to be written. It is still very complicated to modify the data of the problem.

Not to mention that the number of variables and the number of equations cannot be altered solely by redefining parameters. A new variable or a new equation would require many additional parameters also.

However, we can understand the logic behind systems of linear equations: we know the exact way how a system with any number of variables and any number of equations, can be implemented. The model formulation is *redundant*: model elements are defined very similarly, much of the code can be copy-pasted – it seems like the model implementation does not exploit the underlying logic of systems of linear equations.

First, let us try to express how the solution of a systems of linear equations generally work.

1. There are variables. The number of variables is not known. These each are reals with no bounds.

2. There are equations. The number of equations is not known.

3. All equations have a single right-hand side value. This means there exists a parameter *for all equations* that define these RHS constants.

4. All the equations have a left-hand side where we add all the variables up, each with a single coefficient, which is a parameter. This means there exists a parameter *for all equations, AND all variables* that define the coefficients. Note that here *„AND"* means for all *pairs* of equations and variables. That implies, if there are n equations and k variables, there are a total of nk parameters.

5. After the `solve` statement, we simply write a `printf` statement individually *for all variables*, to print their values found by the solution algorithm.

We can see that the *for all* parts of the logic is where the redundancy is introduced into the model file, and also the data section. The reason is that in these cases, code must be reused many times. We need a uniform way to tell the solver that we actually want to do many similar things at once. For example, to write `var` and `printf` statements *for all* variables in the system, `s.t.`

statements, and `param` statements denoting RHS constants *for all* equations in the system. Also, to define `param` statements *for all equations and variables* that describe the coefficients, and to appropriately implement each of these constraints that involve summing up a particular coefficient multiplied by the particular variable, *for all* variables.

In GNU MathProg, this can be achieved by **indexing expressions**, which also requires the concept of **sets**.

Sets contain many different elements (possibly zero, in which case we have an empty set). There are many ways in the language sets can be defined. Note that there are also literals that denote sets. For example, `1..10` denote the set of integers from 1 to 10. However, in this example, we define our own sets with the `set` statement.

There are two sets needed in our model. One is for the variables, and one is for the equations.

```
set UnknownValues;
set Equations;
```

The set `UnknownValues` correspond to the variables of the system of linear equations. In our previous example, they were namely x, y, z, w, and v. The set `Equations` correspond to the equations in the system, they were `Eq1`, `Eq2`, `Eq3`, `Eq4` and `Eq5`.

We used the name `UnknownValues` to differentiate the variables of the system of linear equations to be solved (these are the „unknown values") from the variables defined in the GNU MathProg model by a `var` statement. These are in a one-to-one correspondence in this particular model: for each unknown value in the system of equations, there will be exactly one GNU MathProg model variable defined. Just remember that these are different concepts.

At this point, the contents of the sets are not defined. The reason is exactly the same as when parameters are only given values in the data section. The actual content of the sets are not part of the model logic, as the set of unknown values and equations both may vary. The procedure for giving values for sets is the same as parameters defined by the `param` statement. They can be given a value at definition, or only afterwards in a data section. In this case, we choose the latter.

Now, let us define the parameters in our model by the following code.

```
param Rhs {e in Equations};
param Coef {e in Equations, u in UnknownValues};
```

This is the first time we use indexing expressions. They are denoted by the curly braces. In the first example, `Equations` is the set providing the possible indices for the parameter `Rhs`. The line literally means that we define a parameter named `Rhs` *for all* `e`, for which `e` is an element of the `Equations` set. This means that whatever the number of equations is, there is an individual parameter for each of these equations.

Note that we could also legally provide a value for the parameter `Rhs`, like we did for the individual parameters before. We do not want that now, as constants at the RHS are not part of the model logic, they are represented in the data section. Note that we have much more freedom on indexing expressions, not mentioned for now, refer to the GNU MathProg language manual for the possibilities.

The parameter `Coef` is defined similarly, but there are two indices separated by a comma. This literally means that there is a parameter defined for all (e, u) ordered pairs, for which `e` is an equation and `u` is an unknown value. In other words, we define a parameter for each element in the Cartesian product of the two sets. There can be many indices in an indexing expression, the maximum is currently 20. They always mean the same: the index set is obtained by all possible combinations of the individual indices. We also call the number of indices as the number of **dimensions** of a set. The parameter `Rhs` has a one-dimensional index, while `Coef` has a two-dimensional index.

Now we can define the variables in the model.

```
var value {u in UnknownValues};
```

There is only one variable in this model, introducing the „unknown values" to be found in the system of linear equations. We define each model variable to be found, for each unknown value in the system of equations. The name of the model variable will be `value`.

Note that index names like *u* for `UnknownValues` can be arbitrary each time we refer to the same set, but it is strongly recommended that they are consistent throughout the model section for readability purposes. Here we use `e` for equations and `u` for unknown values.

Also note that the variables luckily have consistently no bounds, they all can take arbitrary real values.

Now that we have defined the required parameters and variables in the model we can express the constraints.

```
s.t. Cts {e in Equations}:
  sum {u in UnknownValues} Coef[e,u] * value[u] = Rhs[e];
```

Now let us see the parts of this implementation one-by-one. Whitespace characters, including newlines are for better readability. Note that this is a single constraint in GNU MathProg, and it is named `Cts`, but due to the indexing, it will generate a set of individual equations in the model.

First, there must be exactly one constraint written up for each equation in the system. This is established by the first indexing expression before the colon, `e in Equations`. This does not only causes the number of the equation constraints to be determined, but it also introduces `e` as a constant that we can refer to in the definition part.

The RHS of the constraint is the simpler, as it is the `Rhs` parameter itself. But here we can see how indexing works for parameters. When `Rhs` was defined, it was defined for all equations. That means, for example if there are three equations in the system, named `Eq1`, `Eq2` and `Eq3`, then exactly one parameter is defined in the model for each of these. They can be referred to as `Rhs['Eq1']`, `Rhs['Eq2']` and `Rhs['Eq3']` respectively. Remember that these are three different parameters. However, we do not write something like `Rhs['Eq1']` in the model section, because the exact equation name is not part of the model logic, it is rather arbitrarily given in a data section. Instead, we refer to `Rhs[e]`. Remember that `e` denotes the current element of the `Equations` set we are writing a constraint for. The result is that in each constraint, the RHS is the `Rhs` parameter for the *corresponding* equation.

The LHS of the constraint is more complicated, because it contains a summation of an undetermined number of terms. Note that the logic is that for each unknown value, we multiply the variable with the corresponding coefficient of the system, and these terms are added up. The `sum` operator does exactly this job for us. It involves an indexing expression, where we denote that we want to sum up terms *for all* unknown values `u`, and the term to be summed is followed. The `sum` operator in GNU MathProg is later in precedence than multiplication, so there is no need to use parentheses, `Coef[e,u] * value[u]` is the term that is summed up for all `u`. Each of these terms refers to the variable of the corresponding unknown value `u`, which is `value[u]`. The term also refers to `Coef[e,u]`, which is the coefficient parameter for the equation `e` and the unknown value `u`. Note that inside the scope of the `sum` operator, the `u` index is also available, as `e` inside the whole indexed constraint. However, `u` would not be available outside the sum, for example, in the RHS. It would not even make sense, because `u` is introduced only to define the terms of the sum.

Now that we have a single `s.t.` constraint statement describing the model of systems of linear equations, the `solve` statement may come, after which we can print the values of the variables. Again, we do not know the exact set of variables, but indexing can be used to do something *for all* variables.

```
for {u in UnknownValues}
{
  printf "%s = %g\n", u, value[u];
}
```

The `for` statement requires an indexing expression, and effectively repeats the statements written inside the block following it, throughout the index set given. Note that the statements allowed in a `for` block is limited, but it includes `printf` and another `for`, which are enough for very complex tasks to be implemented simply. The `for` statement cannot be used to define model elements like parameters, variables, constraints or the objective, but is useful for displaying information.

Now, for each `u` unknown value, we want to print out the given value in the solution of the system of equations. Therefore the `printf` statement refers to `u` itself, the name of the variable of the system of equations, and its value, which is `value[u]`.

At this point, we are ready with the model section. That means, any systems of linear equations can be translated into a data section, and if done with correct syntax, can be solved without altering the model file. Now, let us translate the previous problem instance into a well-formed data section.

```
set UnknownValues := x y z w v;
set Equations := Eq1 Eq2 Eq3 Eq4 Eq5;
```

The sets are given values here. That means, names of the unknown values and equations are given one-by-one. Note that in the data section, there is no need for apostrophes. If we wanted to hard-code a set definition into the model section, we would need apostrophes or quotation marks to denote string constraints.

One of the parameters that require values to be given is the RHS constants.

```
param Rhs :=
  Eq1  -5
  Eq2  -5
  Eq3  1.5
  Eq4  0
  Eq5  -0.5
  ;
```

There are multiple ways parameter data can be provided, refer to the GNU MathProg language manual for more details [1]. The simplest is shown here. The index and value pairs are simply enumerated, delimited by whitespace. One row for each pair is just a recommendation for readability. Note that here we must refer to the equation names we gave for the `Equations` set.

This similar format can also be applied if the index has more dimensions, as `Coef`.

```
param Coef :=
  Eq1  x  4
  Eq1  y  -2
  Eq1  z  3
  Eq1  w  0
  Eq1  v  0
  Eq2  x  0
  Eq2  y  0
  Eq2  z  -1
  Eq2  w  -1
  Eq2  v  0
  Eq3  x  0
  Eq3  y  1
  Eq3  z  1
  Eq3  w  0
  Eq3  v  0
  Eq4  x  4
  Eq4  y  -7
  Eq4  z  0
  Eq4  w  2
  Eq4  v  5
  Eq5  x  1
  Eq5  y  0
  Eq5  z  2
  Eq5  w  0
  Eq5  v  3
  ;
```

The scheme is the same. Pairs for indices and their corresponding values are enumerated. However, each such pair now actually consists of three elements, because the first part of the index is for the equation, and the second is for the unknown variable. Again, here we must refer to the names we given in the set definitions in the data block.

And now we are ready. The model section attained its most general form. From now on, solution of any system of linear equations can be done solely by writing a well-formed data section. This data section may come after the model section in the model file itself, or more practically, it can be put in separate file. Typically, each problem instance (in this case, each system of linear equations to be solved) can be implemented in a single data file, containing the data section.

The complete model file and one possible data section are the following. The solution is again exactly the same as before: $x = 2$, $y = 3.5$, $z = -2$, $w = 7$, $v = 0.5$.

```
set UnknownValues;
set Equations;
param Rhs {e in Equations};
param Coef {e in Equations, u in UnknownValues};

var value {u in UnknownValues};

s.t. ImplementingEquations {e in Equations}:
  sum {u in UnknownValues} Coef[e,u] * value[u] = Rhs[e];

solve;

for {u in UnknownValues}
{
  printf "%s = %g\n", u, value[u];
}

end;
```

```
data;

set UnknownValues := x y z w v;
set Equations := Eq1 Eq2 Eq3 Eq4 Eq5;

param Rhs :=
  Eq1  -5
  Eq2  -5
  Eq3  1.5
  Eq4  0
  Eq5  -0.5
  ;

param Coef :=
  Eq1  x  4
  Eq1  y  -2
  Eq1  z  3
  Eq1  w  0
  Eq1  v  0
  Eq2  x  0
  Eq2  y  0
  Eq2  z  -1
  Eq2  w  -1
  Eq2  v  0
  Eq3  x  0
  Eq3  y  1
  Eq3  z  1
  Eq3  w  0
  Eq3  v  0
  Eq4  x  4
  Eq4  y  -7
  Eq4  z  0
  Eq4  w  2
  Eq4  v  5
  Eq5  x  1
  Eq5  y  0
  Eq5  z  2
  Eq5  w  0
  Eq5  v  3
  ;

end;
```

4.6 Other options for data

Although we now have a single model file suitable for all possible systems of linear equations regardless of variable and equation count, there are other tricks in GNU MathProg that further help us by allowing shorter and/or more readable implementation of data sections. Two of these options are presented here: **default values**, and **matrix data format**. A common modeling error, the

out of domain error is also demonstrated here.

We have already mentioned that parameters (and also, sets) that we use in the model need a value to be provided. This value is either calculated on spot, or not given in the model at all, so that it can be provided separately in a data section. If a parameter has a calculated value hard-coded in the model, then providing data for it is forbidden, but if it is not hard-coded, there must be a value in the data sections.

However, we see that many of the coefficients in the system of linear equations can be zero. Actually it is something like a *default* behavior: an equality might only refer to a subset of the variables. If an equality does not refer to a variable, then it still has a zero coefficient due to the modeling logic. But we do not want to provide these zero parameters for all variables each time we write a new equation. Instead, we want to only provide those parameter values that are different than this default zero.

This can be achieved by indicating a **default value** for a given parameter in the model section. In this case, the model parameter does not have a hard-coded calculated value, but a given default value instead. If there is a value provided in a data section, that value is used. If there are no values provided, then the default value is used. Note that default values can also be given for sets. However, they cannot be given for variables, as variables are determined by the optimization procedure, so it does not make sense to provide default values for them.

To add a default value to a parameter, we must do the following.

```
param Coef {e in Equations, u in UnknownValues}, default 0;
```

Note that the default value in this example is zero, but in general it could be any expression that can be calculated on spot. For instance, it can depend on `e` and `u`, the indices of the parameter.

After the default value of zero is given for `Coef`, all entries in the data section that indicate a value of zero can be omitted, trimming the size of the data section significantly.

```
param Coef :=
  Eq1  x  4
  Eq1  y  -2
  Eq1  z  3
  Eq2  z  -1
  Eq2  w  -1
  Eq3  y  1
  Eq3  z  1
  Eq4  x  4
  Eq4  y  -7
  Eq4  w  2
  Eq4  v  5
  Eq5  x  1
  Eq5  z  2
  Eq5  v  3
  ;
```

Note that, while maintaining the code of this data section, care must be taken when adding a new index to a parameter data set, as it may already be there. Duplicate indices are forbidden in the data section, as common sense dictates.

Another trick that may help in designing shorter and more readable data sections is the **matrix data format**. It works in many situations where two-dimensional data must be provided – like in this case. Values for parameter `Coef` can be implemented in matrix format in the following way.

```
param Coef:
       x   y   z   w   v :=
  Eq1  4   -2  3   .   .
  Eq2  .   .   -1  -1  .
  Eq3  .   1   1   .   .
  Eq4  4   -7  .   2   5
  Eq5  1   .   2   .   3
  ;
```

Note that the colon and the `:=` are denoting the matrix data format, whitespace indentation is optional. It is recommended for readability that the data is indented similarly to the way shown.

The rows of the matrix correspond to the values of the first index, while the columns correspond to the second index. Each entry in the matrix gives a value for the parameter identified by the row and the column.

Note that in such a matrix, we do not have to mention all possible rows and columns. Those parameter indices for which we do not give a value will be defaulted. Also, using the `.` character instead of a value in the matrix makes that particular value to be defaulted.

If one wants to refer to the first indices by the columns and the second indices by the rows, it is also possible: it is called the **transposed matrix format**, and it is simply obtained by inserting `(tr)` before the colon.

There are more advanced data description formats in GNU MathProg, which are not mentioned here. For example, it is possible to give multi-dimensional data by describing several different matrices one after another.

An important issue about indexing is the **out of domain** error. For all parameters, variables that are indexed, there is a well described index set. If at any time, a parameter or variable is referred with an index that is not in its defined index set, then – not surprisingly – an error occurs.

Let us demonstrate it by changing the `u` and `e` indices in the constraint where the `Coef` parameter is indexed. The following message is obtained, and the model cannot be processed.

```
Generating Cts...
eqsystem-indexing-matrix-ERROR-outofdomain.mod:10: Coef[x,Eq1] out of domain
MathProg model processing error
```

This is an error because the correct indexing would be `Coef[Eq1,x]`, not `Coef[x,Eq1]`. The latter is not a valid index for parameter `Coef`, because neither `x` is in the set `Equations`, nor `Eq1` is in the set `UnknownValues`. The same error message may be reported if we do not use the indices properly in the data section.

This example suggests an important source of error: indexing must be done properly. In this case, the out of domain error is raised and the mistake could be corrected easily. However, the situation can be much worse, for example, if a wrong indexing would actually mean a valid index in the index set. In that case, the error remains unnoticed, and the wrong model is solved. This is especially dangerous if the two indices are from the same set, as they can be switched by mistake. For example, in a directed graph of vertices A, B and C, the arc AB is different from BA, but both describe an arc, and are valid if implemented in GNU MathProg.

In our example, the two indices are from two different sets, so it is unlikely that a wrong indexing is not reported. However, it may be the case if the two sets have common elements. For example, if both the variables are labeled from 1 to 5, and all the equations are labeled also from 1 to 5, there is nothing that forbids this. The correspondingly edited data section is the following.

```
data;

set UnknownValues := 1 2 3 4 5;
set Equations := 1 2 3 4 5;

param Rhs :=
  1  -5
  2  -5
  3  1.5
  4  0
  5  -0.5
  ;

param Coef:
     1   2   3   4   5 :=
  1  4   -2  3   .   .
  2  .   .   -1  -1  .
  3  .   1   1   .   .
  4  4   -7  .   2   5
  5  1   .   2   .   3
  ;

end;
```

Now, if this is the data section, and it is paired with the wrongly indexed model section we presented before, then there are no out of domain errors, as all wrong indices are actually valid. So a wrong model is solved now, which has a substantially different result than the original: 4.8, −20.6, −67.5, −10.3 and 17. Remember that, the result was originally 2, 3.5, −2, 7, 0.5. Conclusion: always pay special care for indexing, use meaningful and unique names even in the data sections, and always validate the result of model solution.

A valid question arises: what was the problem that the wrongly indexed model had solved? We leave it up to the reader, with a small hint: if the two indices are switched, then we effectively transpose the data matrix.

4.7 General columns

We have seen some capabilities of the GNU MathProg language with which we can make compact and more readable data sections. Now we will show how the already available techniques can be utilized to further decrease the redundancy in a data section.

There is still a bit of redundancy in case of the rows of the equations. The RHS of the equations are provided as a separate parameter where all the equation names must appear. The coefficient matrix for the system of equations is described by another parameter, now in matrix format, but the rows must start with the equation names again. It would be nice if these two kinds of data could appear in a single data matrix.

We can easily solve it in GNU MathProg. Now, a common `EqConstants` parameter is introduced which works as the following. Its indices are the **columns** of the problem in general. Each column can correspond to a variable (unknown value) in the system of equations, or the RHS of the equation. We can implement this by the following code.

```
set UnknownValues;
set Equations;

set Columns := UnknownValues union {'RHS'};

param EqConstants {e in Equations, u in Columns}, default 0;
```

Note that the `Columns` is a set just like `UnknownValues` and `Equations`. However, this set is not provided by a data section, but instead calculated on spot. The `Columns` is the set `UnknownValues`, with the string `RHS` added to it.

```
var value {u in UnknownValues};

s.t. ImplementingEquations {e in Equations}:
  sum {u in UnknownValues} EqConstants[e,u] * value[u]
  = EqConstants[e,'RHS'];
```

The rest of the model is basically the same, but instead of the RHS parameter, we refer to the `EqConstants` parameter with either the index of an unknown value `u` to access the coefficients, or the `RHS` string to access the RHS constant in the given equation.

The data section is the following. The only parameter is `EqConstants`. Note that the solution shall be exactly the same as before, the modifications only affected the format of the data section, but the data content remained the same.

```
data;

set UnknownValues := x y z w v;
set Equations := Eq1 Eq2 Eq3 Eq4 Eq5;

param EqConstants:
        x   y   z   w   v   RHS :=
  Eq1   4   -2  3   .   .   -5
  Eq2   .   .   -1  -1  .   -5
  Eq3   .   1   1   .   .   1.5
  Eq4   4   -7  .   2   5   0
  Eq5   1   .   2   .   3   -0.5
  ;

end;
```

Note that this implementation does not work in one extreme situation: where one of the unknown values are named `RHS`, in which case it is interchangeable with the string `RHS` denoting the right-hand side column instead of ordinary variables in the system. So, for this model, the usage of keyword `RHS` is forbidden for unknown values in the data section. But this restriction is a fair trade for the more compact data section we got instead.

We can make our model foolproof by establishing this rule with a `check` statement. It works like an assertion: a logical expression is evaluated, and if the result is false, the model generation is aborted.

In this example we state that `UnknownValues` shall not contain the string `RHS`. In other words, no variables in the system of equations shall be named `RHS`. The statement is best positioned soon after the definition of the `UnknownValues` set.

```
check 'RHS' !in UnknownValues;
```

Note that the `check` statement can also be indexed, which is an effective way of testing multiple logical statements at once.

The evaluation of a `check` statement is also reported in the output generated by `glpsol` as follows.

```
Checking (line 4)...
```

Checking is used to rule out wrong data by reporting it. Identifying bugs caused by unreported data errors can be substantially more difficult.

4.8 Minimizing error

After implementing a model for arbitrary systems of linear equations, let us consider a bit different problem: solving a system of equations „as well as we can".

Although we do not focus on mathematical proofs here, it is worth mentioning that a system of linear equations may have 0, 1 or infinitely many different solutions. This is related to the number of variables and equations in the system in the following way. Note these rules are **not general** as there are important exceptions that can be precisely characterized, but are typical if the coefficients and RHS values in the equations are random or unrelated.

- Infinitely many solutions is the typical case when there are fewer equations than variables.
- A unique solution is typically caused by having the same number of equations and variables. Note that this was the case in all of the previous examples in this section.
- If there are more equations than variables, the system is **overspecified** and the resulting problem is usually infeasible. This can be demonstrated as follows.

Problem 6.

Append a sixth equation $x + y + z + w + v = 0$ to Problem 5 mentioned in previous sections and solve it.

Now having the complete model, Problem 6 can be implemented in a single data section. A new element `Eq6` is added to the set `Equations`, and the corresponding RHS and coefficients are also introduced. Note that the original model and data section format is used, where parameters `Rhs` and `Coef` are not merged.

```
data;

set UnknownValues := x y z w v;
set Equations := Eq1 Eq2 Eq3 Eq4 Eq5 Eq6;

param Rhs :=
  Eq1  -5
  Eq2  -5
  Eq3  1.5
  Eq4  0
  Eq5  -0.5
  Eq6  0
  ;

param Coef:
    x  y  z  w  v :=
  Eq1  4  -2  3  .  .
  Eq2  .  .  -1  -1  .
  Eq3  .  1  1  .  .
  Eq4  4  -7  .  2  5
  Eq5  1  .  2  .  3
  Eq6  1  1  1  1  1
  ;

end;
```

We already know the original solution for Problem 5 with the system of just the first five equations, it is $x = 2$, $y = 3.5$, $z = -2$, $w = 7$, $v = 0.5$. This literally means that the first five equations imply that the values of the five variables are as mentioned. That means $x+y+z+w+v = 11$ is also implied which contradicts the sixth equation $x+y+z+w+v = 0$ in Problem 6. In short, if the first five equations hold, then the sixth cannot. Not surprisingly, the overspecified model turns out to be infeasible.

```
GLPSOL: GLPK LP/MIP Solver, v4.65
Parameter(s) specified in the command line:
 -m eqsystem-original.mod -d eqsystem-overspecified.dat --log
  ↪  eqsystem-overspecified-original.log
Reading model section from eqsystem-original.mod...
21 lines were read
Reading data section from eqsystem-overspecified.dat...
30 lines were read
Generating Cts...
Model has been successfully generated
GLPK Simplex Optimizer, v4.65
6 rows, 5 columns, 19 non-zeros
Preprocessing...
6 rows, 5 columns, 19 non-zeros
Scaling...
 A: min|aij| =  1.000e+00  max|aij| =  7.000e+00  ratio =  7.000e+00
Problem data seem to be well scaled
Constructing initial basis...
Size of triangular part is 4
      0: obj =   0.000000000e+00 inf =   8.500e+00 (2)
      1: obj =   0.000000000e+00 inf =   3.667e+00 (1)
LP HAS NO PRIMAL FEASIBLE SOLUTION
glp_simplex: unable to recover undefined or non-optimal solution
Time used:   0.0 secs
Memory used: 0.1 Mb (118495 bytes)
```

However there is still an interesting question if we are faced an overspecified system of equations. Instead of finding a perfect solution, find a solution for which the *error at the equations is kept minimal*. More precisely, the maximum of errors.

Problem 7.

Given a system of linear equations, find a solution for it for which the maximum of the errors among all equations is minimal. The error for a given equation is defined by the absolute value of the difference between the LHS and the RHS.

(The LHS of the equation is the sum of the variables multiplied by coefficients, and the RHS is a constant. Use the same data description method as before.)

Now, this is not a feasibility problem anymore. Being precise, now all possible values of the variables give a feasible solution: it is just the value of the objective (maximal error) what can be smaller or larger for some solutions. This is therefore an optimization problem.

Solving it efficiently requires a modeling trick: **minimizing the maximum objective**. It can be regarded as a „design pattern" in mathematical programming.

The first part of the idea is that we introduce the objective, the maximal error itself into the model as an individual variable as follows.

```
var maxError;

minimize maxOfAllErrors: maxError;
```

At this point, the objective is just an independent variable that can be freely set. Our problems is to find a way that enforce this `maxError` variable to actually represent the maximum error of the equations.

Suppose that L and R are the left-hand side and right-hand side values of an equation for some values of the variables. Ideally, $L = R$ and the error is zero. Otherwise, the error is $|L-R|$, which is not a linear function of the variables. Even if it was, it would be still problematic to find a maximum of these, because the „maximum" function is itself not linear.

Here comes our second idea. The errors themselves are not represented in the model, rather, an *upper bound for the error* is used. Suppose that E is a valid upper bound for the error between L and R, meaning $|L - R| \leq E$. Now, this can be expressed as linear constraints as follows.

$$\begin{aligned} L - R &\leq E \\ L - R &\geq -E \end{aligned} \tag{11}$$

Note that E must be nonnegative, as the errors themselves are nonnegative. Alternatively, we could say $R - E \leq L \leq R + E$. Anyways, these are two linear inequalities.

Now, what can possibly be a good candidate to be an upper bound for the error in any equations? Well, the maximum of the errors, of course, which is denoted by `maxError`. Therefore we define two constraints so that the two sides of the equations may differ *at most* by the amount of `maxError`.

```
s.t. Cts_Error_Up {e in Equations}:
  sum {u in UnknownValues} Coef[e,u] * value[u]
  <= Rhs[e] + maxError;

s.t. Cts_Error_Down {e in Equations}:
  sum {u in UnknownValues} Coef[e,u] * value[u]
  >= Rhs[e] - maxError;
```

This can be achieved by duplicating the original constraint, and involve `maxError`. The constraints must have unique names, here `Cts_Error_Up` and `Cts_Error_Down` were chosen.

Surprisingly, our model is ready. Now let us try to understand what happens when it is solved. By the constraints, `maxError` is forced to work as a valid upper bound for the errors for all equations at the same time. Meanwhile, `maxError` is an objective that is to be minimized. So these together will effectively find the least possible `maxError` value, for which all the equations can be satisfied with a maximum error of `maxError`.

Here is the full model section, with some `printf` statements added to provide us a meaningful output.

```
set UnknownValues;
set Equations;

param Rhs {e in Equations};

param Coef {e in Equations, u in UnknownValues}, default 0;

var value {u in UnknownValues};

var maxError;

s.t. Cts_Error_Up {e in Equations}:
  sum {u in UnknownValues} Coef[e,u] * value[u]
  <= Rhs[e] + maxError;

s.t. Cts_Error_Down {e in Equations}:
  sum {u in UnknownValues} Coef[e,u] * value[u]
  >= Rhs[e] - maxError;

minimize maxOfAllErrors: maxError;

solve;

printf "Optimal error: %g\n", maxError;
printf "Variables:\n";
for {u in UnknownValues}
{
  printf "%s = %g\n", u, value[u];
}
printf "Equations:\n";
for {e in Equations}
{
  printf "%5s: RHS=%10f, actual=%10f, error=%10f\n",
    e, Rhs[e],
    sum {u in UnknownValues} Coef[e,u] * value[u],
    abs(sum {u in UnknownValues} Coef[e,u] * value[u] - Rhs[e]);
}

end;
```

If we solve it we get the following results.

```
Optimal error: 0.478261
Variables:
x = -3.42029
```

```
y = -1.21884
z = 2.24058
w = 3.23768
v = -0.36087
Equations:
  Eq1: RHS= -5.000000, LHS= -4.521739, error=  0.478261
  Eq2: RHS= -5.000000, LHS= -5.478261, error=  0.478261
  Eq3: RHS=  1.500000, LHS=  1.021739, error=  0.478261
  Eq4: RHS=  0.000000, LHS= -0.478261, error=  0.478261
  Eq5: RHS= -0.500000, LHS= -0.021739, error=  0.478261
  Eq6: RHS=  0.000000, LHS=  0.478261, error=  0.478261
```

If we manually investigate the results, we can verify that all the variables are multiples of $\frac{1}{690}$, they are not trivial values.

What can be interesting is that the error is the same for all six equations. Maybe it is a general rule if the number of equations is one more than the number of variables? This leads to another mathematical problem.

4.9 Equation systems – Summary

We have learnt through the basic skills in GNU MathProg with which we can implement linear mathematical models: using parameters, separating the model and the data section, and most importantly, indexing expressions. We also solved a simple but nontrivial optimization problem about minimizing the errors in equations.

Note that the coding effort was minimal in GNU MathProg, or mathematical programming in general, compared to solution algorithms we had to otherwise implement ourselves.

From now on, it is recommended for each subsequent optimization problem to implement a single general model file containing the model section, but no problem data. Each problem instance the model needs to be solved for, can be implemented in a single separate data file containing the whole data section. Therefore, there should be as many data files as problem instances.

Chapter 5

Production problem

In this chapter, a fundamental LP problem is explored, in terms of model implementation. This is the **production problem**: determining which products we shall fabricate to obtain the most profit, if our resources are limited. It is also called **product mix problem**, or production planning problem. The production problem is one of the oldest problems in Operations Research, plenty of alternative tutorials use it as introduction to mathematical programming (see for example [11, 12]).

The diet problem is also presented, where the least expensive food mix shall be found to support all nutrient needs. The two problems can have a common generalization which is also shown. A lookout into integer programming is also presented in the final section, where „packages" of raw materials and products can be bought and sold at once.

The chapter focuses on how a single model can be extended to incorporate various circumstances one-by-one. We start from a common exercise that can even be done by hand, and finally arrive at a very general and complex optimization model.

5.1 Basic production problem

The simplest case of the production problem can be described as follows, in general.

Problem 8.

Given a set of products, and a set of raw materials needed for them. Production is linear and can be done in any amount, with consumption rates of raw materials given: we know how much of each raw material is consumed to produce 1 unit of each product. We have a fixed amount of each raw material available, and a fixed unit revenue for each product.

Determine the optimal amounts of all products to be made, so that we do not consume more of each raw material than available, and the total revenue from all products is maximized.

This above definition can be a bit difficult to understand. This actually correspond only to the *model* logic, and will be implemented in a model file. However, let us see a particular example for the production problem, with the data supported.

Problem 9.

We have a manufacturing plant, which is capable of producing three different products, named `P1`, `P2` and `P3`. There are four raw materials that are required for production, named `A`, `B`, `C` and `D`. We have exact data for the following.

- *Amount of each raw material required for producing 1 unit of each product.*
- *Available amount of each raw material that can be used for production.*

- *Revenue for 1 unit of each product.*

These can be viewed in a single table, as shown below.

	P1	*P2*	*P3*	*Available*
A (electricity)	200 kWh	50 kWh	0 kWh	23000 kWh
B (working time)	25 h	180 h	75 h	31000 h
C (materials)	3200 kg	1000 kg	4500 kg	450000 kg
D (production quota)	1	1	1	200
Revenue (per unit)	252 $	89 $	139 $	

Note that any amounts of each product can be produced, the raw material requirements are exactly proportional. There is no other limitation than total availability of each raw material. Amounts can also be fractional.

Determine the optimal amount of `P1`, `P2` and `P3` to be produced, so that raw material availability is respected, and the total revenue after products is maximized.

(Note that problem data are entirely fictional.)

Note that compared to „pure mathematical" models as the system of linear equations we have seen before, there are units in the problem data. However, GNU MathProg does not have a built-in feature to remember the quantities, currencies of model elements (variables, parameters, constraints). We have to work with scalars only. The general approach of handling units is that data corresponding to the same quantity are converted to the same unit, and treated consistently in the whole model. This must always be kept in mind during model formulation.

However, understanding the units in the problem is recommended, because it helps us evading mistakes, by remembering that only scalars of the same unit shall be added together at any time. For example, amounts of kWh and kg cannot be added. Then, either adding them is wrong at all, or we are missing one or more factors that will bring these quantities to the same dimension and same unit.

Now let us start formulating the model. The first step is the selection of the **decision variables**. The goal of optimization is to determine the values of these decision variables. Each solution obtained describes a decision of how the plan will be operated. Of course, generally not all solutions are be feasible, and the revenues are also generally different. Therefore we look for the feasible solution with the highest revenue.

The decision variables can be directly read from the problem text. The *amounts of each product to be determined* are the decision variables, these must definitely be determined by the optimization. The question arises, whether there shall be more variables or not? If we only know the amounts produced, we can calculate everything in the plant which is relevant now: the exact amounts consumed of each raw material, and the total revenue. So, at this point, we do not need any more variables in our optimization model.

The **objective function** can be easily determined. The amounts of production each must be multiplied by the unit revenue for that product, and the total revenue is the sum of such products.

What remains are the **constraints**, and variable **bounds**. In general, what we only need is that a production can be zero, or a positive number. But, it definitely cannot be negative. So each variable shall be nonnegative, this is a lower bound. There is no upper bound, as any production amount is considered feasible as long as there is sufficient raw material for it. And here we arrived to the only constraint, which is about *raw material availability.* Based on the amounts produced (these are denoted by the variables), we can easily calculate how much of each raw material is used per product, and in total. These must not be greater than the availability of each particular raw material.

We defined the variables, the objective, constraints and bounds, so we are ready to implement our model in GNU MathProg. First, we do not apply indexing, and work with the most straightforward way.

The variables denote production amounts. For convention, these are all in \$ currency.

```
var P1, >=0;
var P2, >=0;
var P3, >=0;
```

Constraints are formulated next. For each production amount, it must be multiplied by the coefficient that describes raw material consumption per product unit. These shall be added for all three products to obtain the total consumption for a given raw material.

Note how the tabular data of the problem correspond to the implementation of constraints.

```
s.t. Raw_material_A:  200 * P1 +   50 * P2 +    0 * P3 <= 23000;
s.t. Raw_material_B:   25 * P1 +  180 * P2 +   50 * P3 <= 31000;
s.t. Raw_material_C: 3200 * P1 + 1000 * P2 + 4500 * P3 <= 450000;
s.t. Raw_material_D:    1 * P1 +    1 * P2 +    1 * P3 <= 200;
```

Finally, the objective can be defined based on production amounts as well. Note that each constraint is within its own unit for the raw material, and the objective is in \$ unit. From now on, we use units consistently and do not refer to them.

```
maximize Raw_material: 252 * P1 +   89 * P2 +  139 * P3;
```

A `solve` statement can be inserted in the model, after which some additional post-processing work can be done to print out the solution. The full code is the following. We print the total revenue (the objective), the production of each products (the variables), and the usage of each raw material. In the usage part, we print both the total amount consumed for production, and the amount remained available.

```
var P1, >=0;
var P2, >=0;
var P3, >=0;

s.t. Raw_material_A:  200 * P1 +   50 * P2 +    0 * P3 <= 23000;
s.t. Raw_material_B:   25 * P1 +  180 * P2 +   50 * P3 <= 31000;
s.t. Raw_material_C: 3200 * P1 + 1000 * P2 + 4500 * P3 <= 450000;
s.t. Raw_material_D:    1 * P1 +    1 * P2 +    1 * P3 <= 200;

maximize Raw_material: 252 * P1 +   89 * P2 +  139 * P3;

solve;

printf "Total Revenue: %g\n", ( 252 * P1 +   89 * P2 +  139 * P3);

printf "Production of P1: %g\n", P1;
printf "Production of P2: %g\n", P2;
printf "Production of P3: %g\n", P3;

printf "Usage of A: %g, remaining: %g\n",
              ( 200 * P1 +   50 * P2 +    0 * P3),
    23000   - ( 200 * P1 +   50 * P2 +    0 * P3);
printf "Usage of B: %g, remaining: %g\n",
              (  25 * P1 +  180 * P2 +   75 * P3),
    31000   - (  25 * P1 +  180 * P2 +   75 * P3);
printf "Usage of C: %g, remaining: %g\n",
              (3200 * P1 + 1000 * P2 + 4500 * P3),
    450000 - (3200 * P1 + 1000 * P2 + 4500 * P3);
printf "Usage of D: %g, remaining: %g\n",
              (   1 * P1 +    1 * P2 +    1 * P3),
    200     - (   1 * P1 +    1 * P2 +    1 * P3);

end;
```

In this model file, there is no data section at all, data is hard-coded into the model. Therefore we can solve it with `glpsol` without providing any additional data files, and by doing so the following result is obtained. Only our own `printf` results are shown here.

```
Total Revenue: 33389
Production of P1: 91.3386
Production of P2: 94.6457
Production of P3: 14.0157
Usage of A: 23000, remaining: -3.63798e-12
Usage of B: 20370.9, remaining: 10629.1
Usage of C: 450000, remaining: -5.82077e-11
Usage of D: 200, remaining: 0
```

We interpret the solutions as follows. It turns out that 33389 is the maximal revenue that can be obtained. The method of obtaining this revenue is that 91.34, 94.65 and 14.02 units of `P1`, `P2` and `P3` must be produced. Note that, in this case it is allowed to produce fractional amounts of a product – this can be the case in practice for example if the products and raw materials are chemicals, fluids, heat, electricity or other quantity that can be divided.

Production consumes all of `A`, `C` and `D`, but there is a surplus of `B` which is not used up. Some remaining amounts are reported to be extremely small positive numbers. These are actually small numerical errors from the actual value of zero in the optimal solution, because `glpsol` uses floating-point arithmetic and it is not perfect. If this output is inconvenient, we may use the format specifier `%f` instead of `%g`, or alternatively, round down the numbers to be printed explicitly in the model with the built-in `floor()` function.

Another option is to add `--xcheck` as command line argument to `glpsol`. This forces the final solution to be recalculated with exact arithmetic, eliminating rounding errors.

```
glpsol -m model.mod -d data.dat --xcheck
```

One interesting thing about the solution is that three of the remaining amounts are zero. If we varied the problem data and solve the model again and again, it would turn out that from the seven values printed (three production amounts and four remaining amounts), there are almost always three zeroes. In general, the number of zeroes is the number of products, and the number of nonzeroes is the number of raw materials. (Exceptions arise in some special cases.) This is a beautiful property of production problems that is better understood if we know how the solution algorithms (particularly the simplex method) work, generally for LP problems. However, we do not focus on the algorithms here, only the model implementations. Nevertheless, understanding of how a good solution looks like is a very valuable skill.

We now have a working implementation for the particular production problem described. However, we know that this solution is not very general. If another production problem is in question, we must understand and tamper with the code describing the model logic. Also note that the exact expressions describing the total consumption of each raw material is appearing three times: once in the constraints, and twice in the post-processing work. This level of redundancy is usually bad code design, regardless of programming language.

Our next task is a more general, indexed model, which requires only a properly formatted data section to solve any production problems.

In the production problem, there are two sets that are in question, these are the set of products and the set of raw materials.

```
set Products;
set Raw_Materials;
```

We can also identify three important parameters. One for the ratios of production, this is defined for each pairs of raw materials and products, we will call it `Consumption_Rate`. One parameter is for availability. This is defined for each raw material, we name it `Storage`. The name „storage" captures the logic of how raw materials work in the modeling point of view: they are present in a given amount beforehand, like physically stored material, then no more than this amount can be used up for production. Another parameter is the `Revenue`, which is defined for each product.

```
param Storage {r in Raw_Materials}, >=0;
param Consumption_Rate {r in Raw_Materials, p in Products}, >=0, default 0;
param Revenue {p in Products}, >=0, default 0;
```

Notice how indexing is used so that each `param` statement refer to not only a single scalar, but a collection of values instead. For the `Consumption_Rate` and `Revenue`, we also provide a default value of zero. That means if we do not provide data, then we assume no raw material need or revenue for that particular case.

Also, in GNU MathProg, we are able to define bounds and other value restrictions for parameters. In this case, all three parameters are forced to be nonnegative, by the `>=0` restriction. This is generally a good practice if we do not except specific values for a given parameter. If a restriction is violated by a value given for the parameter (for example, in the data section, or calculated on spot), then model processing terminates with an error describing the exact situation. It is much easier to notice and correct errors in this case, than allowing a wrong parameter value in the model, which can be solvable to an invalid solution. It is generally difficult to debug a model once it can be processed, so it is recommended to explicitly check data as much as possible.

The variables can be defined now. They denote production amounts, and each are nonnegative.

```
var production {p in Products}, >=0;
```

Finally, all the constraints can be described by one general `s.t.` statement. The logic is the following. There is a single inequality for each raw material: its total consumption cannot exceed its availability. The availability is simply described as a parameter, but the total consumption is obtained by a summation. We must sum, for each product, its amount multiplied by the consumption rate of that particular raw material.

```
s.t. Material_Balance {r in Raw_Materials}:
    sum {p in Products} Consumption_Rate[r,p] * production[p]
    <= Storage[r];
```

The objective is obtained as a sum for all products, the amounts must be multiplied by the unit revenues.

```
maximize Total_Revenue:
    sum {p in Products} Revenue[p] * production[p];
```

Finally, we can implement a general post-processing work to print the total revenue, production amounts and raw material usages. The full model file is the following.

```
set Products;
set Raw_Materials;

param Storage {r in Raw_Materials}, >=0;
param Consumption_Rate {r in Raw_Materials, p in Products}, >=0, default 0;
param Revenue {p in Products}, >=0, default 0;

var production {p in Products}, >=0;

s.t. Material_Balance {r in Raw_Materials}:
    sum {p in Products} Consumption_Rate[r,p] * production[p]
    <= Storage[r];

maximize Total_Revenue: sum {p in Products} Revenue[p] * production[p];

solve;

printf "Total Revenue: %g\n", sum {p in Products} Revenue[p] * production[p];

for {p in Products}
{
  printf "Production of %s: %g\n", p, production[p];
}

for {r in Raw_Materials}
{
  printf "Usage of %s: %g, remaining: %g\n",
      r, sum {p in Products} Consumption_Rate[r,p] * production[p],
      Storage[r] - sum {p in Products} Consumption_Rate[r,p] * production[p];
}

end;
```

If the corresponding data file is implemented as follows, we should get the same result as for the straightforward implementation.

```
data;

set Products := P1 P2 P3;
set Raw_Materials := A B C D;

param Storage :=
  A  23000
  B  31000
  C  450000
  D  200
  ;

param Consumption_Rate:
       P1    P2    P3 :=
  A   200    50     0
  B    25   180    75
  C  3200  1000  4500
  D     1     1     1
  ;

param Revenue :=
  P1  252
  P2  89
  P3  139
  ;

end;
```

Although the model is very general and compact, it still contains some redundancy. The total consumed amount of each raw material is still represented three times in the code. At least, we do not have to rewrite that code ever again if another problem data is given, we only have to modify the data section. However, we still want to eliminate this redundancy.

Remember that we can introduce parameters in the model section and calculate values on spot. If we are after the `solve` statement, then even variable values can be referred to, as their values are already determined by the solver. We introduce `Material_Consumed` and `Material_Remained` to denote the total amount consumed, and remained, for each material.

```
param Material_Consumed {r in Raw_Materials} :=
    sum {p in Products} Consumption_Rate[r,p] * production[p];
param Material_Remained {r in Raw_Materials} :=
    Storage[r] - Material_Consumed[r];

for {p in Products}
{
  printf "Production of %s: %g\n", p, production[p];
}

for {r in Raw_Materials}
{
  printf "Usage of %s: %g, remaining: %g\n",
      r, Material_Consumed[r], Material_Remained[r];
}
```

The solution shall be exactly the same as before again, for the same data file. But now, some of the redundancy is eliminated from the model section. Unfortunately, the parameter for the total amounts consumed cannot be used in the constraints where it appears first. More on that later.

5.2 Introducing limits

Now that we have a working implementation for arbitrary production problems, let us change the problem description itself.

Problem 10.
Solve the production problem, provided that for each raw material and each product, a minimal and maximal total usage is also given that must be respected by the solution.
The usage of a raw material is the total amount consumed, and the usage of a product is the total amount produced.

These are additional restrictions on the production mix in question. Let us see how it affects the model formulation. The problem remained almost the same, just additional solutions must be excluded from the feasible set. Namely, those for which any newly introduced restriction is violated.

The change only added new restrictions, so the variables, the objective and even the already mentioned constraints and bounds may remain the same. The new limits shall be implemented by new constraints and/or bounds. We also need to provide new options in the data sections where these limits can be given as parameter values.

There are four limits altogether.

- *Upper limit on production amount.* The production amount for any product `p` appears as the `production[p]` variable in the model. A constant upper limit for this value can easily be implemented as a linear constraint, but an even easier way is to implement it as an upper bound of variable `production`.
- *Lower limit on production amount.* The same as for the upper limit. Just note that there is already a nonnegativity bound defined for the variables, and defining two lower or two upper bounds on the same variable is forbidden in GNU MathProg.
- *Upper limit on raw material consumption.* The total consumption of each raw material is already represented in the model as an expression. Actually, the only constraint in the production problem so far defines an upper limit for this expression as `Storage[r]` for any raw material `r`. Therefore there is no need to further define upper bounds. If one wants to give an extra upper limitation for the total consumption of a raw material, then it can be done without modifying the model section, just by decreasing the appropriate `Storage` parameter value in the data section.
- *Lower limit on raw material consumption.* As we noticed before, the expression already appears in a constraint. We need to implement another constraint with exactly the same expression, but expressing a minimum limit instead of the maximum.

First, we have to add the extra parameters to describe the limits.

```
param Storage {r in Raw_Materials}, >=0, default 1e100;
param Consumption_Rate {r in Raw_Materials, p in Products}, >=0, default 0;
param Revenue {p in Products}, >=0, default 0;
param Min_Usage {r in Raw_Materials}, >=0, <=Storage[r], default 0;
param Min_Production {p in Products}, >=0, default 0;
param Max_Production {p in Products}, >=Min_Production[p], default 1e100;
```

The new `Min_Usage` is for the minimal total consumption for each raw material, while parameters `Min_Production` and `Max_Production` are for the lower and upper limit of production of each product. The fourth limit parameter is the original `Storage`, for upper limit of raw material consumption. All these limit parameters now have sensible bounds and default values. Lower limits are defaulted to 0, while upper limits are defaulted to 10^{100}, which is a number of a large magnitude that we except not to appear in problem data. Also, the lower limits must be not larger than the upper limits. Meanwhile, all limits must be nonnegative. Remember that these restrictions on parameter values are valuable to prevent mistakes in the data sections but we do not expect them to contribute to generating and solving the model if once satisfied. This is in contrast to variable bounds, which are part of the model formulation and may affect solutions.

The variables and the objective remains the same. However, we must add three additional constraints to tighten the search space of the model. Each of the four constraints actually correspond to the four limits mentioned, while the `Material_Balance` being for the upper limit of total consumption of raw materials, this one was originally there and was left unchanged.

```
var production {p in Products}, >=0;

s.t. Material_Balance {r in Raw_Materials}:
    sum {p in Products} Consumption_Rate[r,p] * production[p]
    <= Storage[r];
s.t. Cons_Min_Usage {r in Raw_Materials}:
    sum {p in Products} Consumption_Rate[r,p] * production[p]
    >= Min_Usage[r];
s.t. Cons_Min_Production {p in Products}:
    production[p] >= Min_Production[p];
s.t. Cons_Max_Production {p in Products}:
    production[p] <= Max_Production[p];

maximize Total_Revenue: sum {p in Products} Revenue[p] * production[p];
```

The post-processing work to print out solution data may remain the same as for the original model. Now, our model section is ready.

To demonstrate how it works, consider the following production problem with limits.

Problem 11.
Solve the production problem described in Problem 9, but with the following restrictions added.

- *Use at least* 20000 h *of working time (raw material `B`).*
- *Fill the production quota: produce at least* 200 *units (raw material `D`), which is actually also the maximum for that raw material.*
- *Produce at most* 10 *units of `P3`.*

As the problem is just an „extension" of the original one, its implementation can be done by just extending the data section with the aforementioned limits. Note that each parameter is indexed with the set of all raw materials and all products. Those that do not appear in the data section will be simply be defaulted to 0 for lower and 10^{100} for upper limits, effectively causing the limits to be **redundant**. In that case, they do not modify the search space of the model, because those limits are true anyways for any otherwise feasible solution.

```
param Min_Usage :=
  B  21000
  D  200
  ;

param Min_Production :=
  P2  100
  ;

param Max_Production :=
  P3  10
  ;
```

The solution of the problem is now slightly different with the newly defined bounds.

```
Total Revenue: 32970
Production of P1: 90
Production of P2: 100
Production of P3: 10
Usage of A: 23000, remaining: -7.27596e-12
Usage of B: 21000, remaining: 10000
Usage of C: 433000, remaining: 17000
Usage of D: 200, remaining: 0
```

This means that 90 units of P1, 100 units of P2 and 10 units of P3 are produced. We may check that all the limitations are met. It is interesting to note about this solution that all the variables are integers, although they are not forced to be. This means that if the problem would change to only consider integer solutions (for example if the product is an object, of which an integer number shall be produced), then this solution would also be valid. Moreover, it would also be the optimal solution, too, because restricting variables to only attain integer values just makes the search space of the model even tighter. So if a solution is optimal even in the original model, meaning that there are no better solutions, then there should not be better solutions in the more restrictive integer counterpart either.

Now, our implementation for limits are done. But we can still improve the model implementation by making it a bit more readable and less redundant. First, there is one nice feature in GNU MathProg: if there is a linear expression that can be bounded by both an upper and a lower value and both of these limits are constants, then these can be done in a single constraint instead of two containing the same expression twice. With this feature, we can reduce the number of `s.t.` statements from four to two in our model section as follows. All other parts of the model, the data, and the solution remain the same.

```
s.t. Material_Balance {r in Raw_Materials}: Min_Usage[r] <=
    sum {p in Products} Consumption_Rate[r,p] * production[p]
    <= Storage[r];
```

```
s.t. Production_Limits {p in Products}:
    Min_Production[p] <= production[p] <= Max_Production[p];
```

There is another thing we can improve, which is actually a modeling technique rather than a language feature: we can introduce **auxiliary variables for linear expressions**. The variables, constraints and the objective will look like the following.

```
var production {p in Products}, >=Min_Production[p], <=Max_Production[p];
var usage {r in Raw_Materials}, >=Min_Usage[r], <=Storage[r];
var total_revenue;

s.t. Usage_Calc {r in Raw_Materials}:
    sum {p in Products} Consumption_Rate[r,p] * production[p] = usage[r];
s.t. Total_Revenue_Calc: total_revenue =
    sum {p in Products} Revenue[p] * production[p];

maximize Total_Revenue: total_revenue;
```

Observe the newly introduced `usage`. We intend to denote the total consumption of a raw material by this variable. Therefore, we add the `Usage_Calc` constraint that ensures this. We now have that variable throughout the model to denote this value. We also do this for the total revenue, which is denoted by variable `total_revenue`, calculated in the `Total_Revenue_Calc` constraint, and then we use it in the objective function. Actually the objective function is itself the `total_revenue` variable.

Now observe that all the expressions we have to limit are actually variables, and all the limits are constants. This means that the constraints for the limits can be converted to bounds of these variables, namely `production` and `usage`. This way, we effectively got rid of all the previously defined constraints and converted them into bounds, but we needed two more constraints to calculate the values of `usage` and `total_revenue`.

We might think that the key importance of introducing the new variable lies on the possibility of using bounds instead of `s.t.` statements which makes our implementation shorter. This is only an example for being useful. The key point is that if some expression is used more than once in our model, we can simply introduce a new variable for it, define a new constraint so that the variable equals that expression, and then use the variable instead of that expression everywhere.

Think about a bit: does this operation of adding auxiliary variables change the search space? Well, formally, yes. The search space is of different dimension. There are more variables, so in a solution to the new problem, we have to decide more values. However, also note that *feasible solutions of the new model will be in a one-to-one correspondence with the original ones*. For a solution feasible to the original problem, we can introduce the variable with the corresponding value of the expression and get a feasible solution for the extended problem, and vice versa, each feasible solution with the extended problem must have its auxiliary variable equal to the expression it is defined for, and hence it can be substituted in the model and we get back to a feasible solution to the original model.

In short, the search space formally changes, but the (feasible) solutions for the problem logically remain the same.

An important question arises: how does the introduction of auxiliary variables change the course of the algorithms, and solver performance? The general answer is that we do not know. There are more variables, so computational performance might be slightly worse, but this is often negligible, because the main difficulty of solving a model in practice comes from the complexity of the search space, which is logically unchanged. Of course, if there are magnitudes more auxiliary variables than ordinary variables and constraints themselves, it might cause technical problems. Also note that, in theory, the solver has the right to substitute into auxiliary variables, effectively reverting back to the original problem formulation, but we generally cannot be sure that it does so. The solver does not see which of our variables are intended to be „auxiliary". If there is an equation constraint in the model, the solver might use that equation to express one of the variables appearing in it, and do a substitution into that variable, despite we had not even thought of it as auxiliary at all.

In short, the course of the solution algorithm may be different, the computational performance of the solver might change, but this is usually negligible.

Note also that we have already done some introduction of new values in the post-processing work, so that we did not have to write the total consumption expression twice. That was with the introduction of a new parameter, not a variable, and only worked after the `solve` statement, because it involved variable values that are only available when the model is already solved. However, now with an auxiliary variable introduced for the total consumption, we can use it both before and after the `solve` statement. Therefore we write this expression down in code only once, and that redundancy finally totally diminished.

The ultimate model code for the limits is the following. Note that the data section, and the solution remained the same as before. Also note that we did not have to introduce new parameters after the `solve` statement as the auxiliary variables do the work.

```
set Products;
set Raw_Materials;

param Storage {r in Raw_Materials}, >=0, default 1e100;
param Consumption_Rate {r in Raw_Materials, p in Products}, >=0, default 0;
param Revenue {p in Products}, >=0, default 0;
param Min_Usage {r in Raw_Materials}, >=0, <=Storage[r], default 0;
param Min_Production {p in Products}, >=0, default 0;
param Max_Production {p in Products}, >=Min_Production[p], default 1e100;

var production {p in Products}, >=Min_Production[p], <=Max_Production[p];
var usage {r in Raw_Materials}, >=Min_Usage[r], <=Storage[r];
var total_revenue;

s.t. Usage_Calc {r in Raw_Materials}:
    sum {p in Products} Consumption_Rate[r,p] * production[p] = usage[r];
s.t. Total_Revenue_Calc: total_revenue =
    sum {p in Products} Revenue[p] * production[p];

maximize Total_Revenue: total_revenue;

solve;

printf "Total Revenue: %g\n", total_revenue;

for {p in Products}
{
  printf "Production of %s: %g\n", p, production[p];
}

for {r in Raw_Materials}
{
  printf "Usage of %s: %g, remaining: %g\n",
      r, usage[r], Storage[r] - usage[r];
}

end;
```

5.3 Maximizing minimum production

In the previous parts, we have seen a complete implementation for the production problem where total raw material consumption and production can be limited by constants for all raw materials and products. Now modify the goal of the optimization, to maximize the minimum production. The exact definition is the following.

Problem 12.
The minimum production in a production problem refers to the product from which the least amount is produced. Maximize the minimum production while problem data are the same.

Regardless of what model we take as a starting point, the simple one or the one with the limitations added, this problem only wants us to modify the objective function. That means, the feasible solutions remain exactly the same as before, and the search space does not change.

This also makes the revenue parameter in the problem irrelevant, because we are no longer interested in how much the products worth in total, only the amounts produced (moreover, only the minimum of those amounts).

We have already done a similar problem in case of the system of linear equations (see Problem 7 from Section 4.8). Then the maximum of the errors of all equations was the objective to be minimized. Now, the minimum of the productions is to be maximized. The modeling technique required is indeed similar.

The idea is that we introduce a variable for the minimum of the production amounts. Precisely, the variable will denote a *lower bound for all the production amounts.* That works as a lower bound for the minimum of those amounts as well. If this variable is maximized, then by the optimization, it will eventually go as large as possible, until it reaches the minimum of the production amounts. With this modeling trick, we can ensure that the final optimal solution found will be the maximal possible value of the minimum of the production amounts. Of course, this is a more general method that works on any number of linear expressions, a minimum of which is to be maximized, or a maximum of which is to be minimized.

In implementation, parameters and post-processing work remains the same, as well as the already-existing variables, bounds and constraints.

```
var production {p in Products}, >=Min_Production[p], <=Max_Production[p];
var usage {r in Raw_Materials}, >=Min_Usage[r], <=Storage[r];
var total_revenue;

s.t. Usage_Calc {r in Raw_Materials}:
    sum {p in Products} Consumption_Rate[r,p] * production[p] = usage[r];
s.t. Total_Revenue_Calc: total_revenue =
    sum {p in Products} Revenue[p] * production[p];
```

The change is that instead of the original objective of the total revenue, we set the minimum production as objective. This requires a newly introduced `min_production` variable to denote a lower bound for all the production amounts, and a constraint which ensures that the variable is itself a lower bound. Then, the variable is to be maximized, and that is the objective function.

```
var min_production;

s.t. Minimum_Production_Calc {p in Products}:
    min_production <= production[p];

maximize Minimum_Production: min_production;
```

Post-processing code can be the same as before, so we print the same data as we did for the total revenue objective. We only added a `printf` statement to emphasize the minimum production amount, which is our current objective.

```
printf "Minimum Production: %g\n", min_production;
```

Note that in our code, `min_production` is the name of the variable, and `Min_production` is the name of the objective function. In GNU MathProg, we could use both as values. However, be cautious when referring to the objective by its name, because constant terms in the objective are omitted. The reason behind this is that the selection of the optimal solution is not affected by constant terms, just the value of the objective. Therefore, it is recommended to refer to the objective function value by the variable `min_production` instead.

We can now run our model to solve two problems. In the first one, all the limits are ignored, except the `Storage`. Note that, in the data section, we can start rows by a hash mark (`#`). This makes the row a comment, effectively excluding it from processing. The data section looks like the following.

```
data;

set Products := P1 P2 P3;
set Raw_Materials := A B C D;

param Storage :=
  A  23000
  B  31000
  C  450000
  D  200
  ;

param Consumption_Rate:
       P1    P2    P3 :=
  A   200    50     0
  B    25   180    75
  C  3200  1000  4500
  D     1     1     1
  ;

param Revenue :=
  P1  252
  P2  89
  P3  139
  ;

param Min_Usage :=
#  B  21000
#  D  200
  ;

param Min_Production :=
#  P2  100
  ;

param Max_Production :=
#  P3  10
  ;

end;
```

Note that revenue data are not needed at all in this model, but it is not a mistake to give them values as well, as long as the parameter is represented in the model section.

The solution to the problem is a production amount of 51.72.

```
Total Revenue: 24827.6
Minimum Production: 51.7241
Production of P1: 51.7241
Production of P2: 51.7241
Production of P3: 51.7241
Usage of A: 12931, remaining: 10069
Usage of B: 14482.8, remaining: 16517.2
Usage of C: 450000, remaining: 0
Usage of D: 155.172, remaining: 44.8276
```

It turns out that production is balanced to the edge to achieve this solution. All the resources are evenly distributed, and all three products are produced in an equal amount. If we think about it, that is not surprising. Why would we produce any amount of products above the minimum, if it is not an advantage regarding the objective function, but a disadvantage regarding the raw materials consumed?

It looks like raw material `C` is the **bottleneck** in this problem. We call a factor a bottleneck if that factor has a visible impact on the final solution if changed, while other factors remained the same. For example, the most scarce resource, as in this case. If there were slightly less or more of raw materials `A`, `B` and `D`, the solution were the same, because all of `C` are completely used up, and distributed evenly among the products. This kind of knowledge can be fundamental in real-world optimization problems, because it tells decision makers that some factors are unnecessary to improve. On the other hand, slightly more or less `C` would probably result in slightly more or less production (probably, because there can be other limitations in the model that we do not see). For this reason raw material `C` is the bottleneck in this particular production problem.

In the second example, the Problem 11 is used without the constraint on `P3`. That means, all the limitations are included now in the model, except that `P3` is not maximized in 10 units. Note that if this constraint were enabled, then there is no question that the optimal solution would be 10 units, as the former solution we already know just produces 10 units of `P3`, and there cannot be more.

In short, we only get a meaningful new problem if we omit this constraint about P3 maximized at 10 units.

```
Total Revenue: 27142.1
Minimum Production: 43.8596
Production of P1: 43.8596
Production of P2: 112.281
Production of P3: 43.8596
Usage of A: 14386, remaining: 8614.04
Usage of B: 24596.5, remaining: 6403.51
Usage of C: 450000, remaining: 0
Usage of D: 200, remaining: 0
```

The solution is slightly different in this case. Now only P1 and P3 is produced in even amounts, both in the minimum production amount of 43.86. Also, not only C, but D is also used up totally. Note that there is a need of P2 much more because all 200 units of D must be used up in this problem. And this answers a former question: why is it an advantage to produce products above the minimum limit? Because it might help satisfying the minimum production and/or minimum consumption constraints.

Note that the objective in this case is slightly worse than that of the first example problem, where the objective was 51.72. This is a natural consequence of differences in the data. In the second example, the data were the same, except that there were additional constraints on consumption and production. If a problem is more constrained, we can only get an optimal solution with the same, or a worse objective value.

Generally, if only the objective is changed in a model, then the set of feasible solutions remains the same, because the objective only guides in selecting the most suitable solution, but not in which solutions are feasible. In this case, the new objective function involved a new variable min_production, which served as a lower bound and was maximized.

Note that, in theory, the variable min_production does not need any lower bound, and can be allowed to even be zero or negative. The solution would still be feasible. However, such solutions are not reported because they are not optimal. Therefore only those feasible solutions are interesting for which min_production is not only a valid lower bound on production amounts, but is actually *strict*, that is, equals the minimum production amount.

And among these solutions we are interested in, min_production works just like an *auxiliary variable*. Each optimal solution where min_production is a strict bound corresponds to a feasible solution of the original problem where revenues were maximized, and vice versa.

5.4 Raw material costs and profit

We have seen an example where only the objective was changed. Now let us see another example, which is more like a natural extension to the problem. From now on, raw materials are not considered „free" anymore: they must be produced, purchased, stored, etc. in general, they have costs. As there is a revenue for each product per unit produced, there is now a cost for each raw material per unit consumed.

Problem 13.
Solve the production problem, but now instead of optimizing for revenue, optimize for profit.

The profit is the difference between total revenue from products, and total costs of raw materials consumed for production. The cost for each raw material is proportional to the material consumed, and is independent of which product it is used for. A single unit cost for each raw material is given.

This is the general description for the problem. Again, we use an example for demonstration, which includes costs for raw materials as well. Other data of the example problem remains the same as before.

Problem 14.
Solve the production problem with the objective of maximized profit, where problem data are the following, with unit costs of raw materials added.

	P1	*P2*	*P3*	*Available*	*Cost*
A	200 kWh	50 kWh	0 kWh	23000 kWh	1 $
B	25 h	180 h	75 h	31000 h	0.07 $
C	3200 kg	1000 kg	4500 kg	450000 kg	0.0013 $
D	1	1	1	200	8 $
Revenue	252 $	89 $	139 $		

Also, there are three limitations, as before.

- *Use at least* 20000 h *of working time (raw material* `B`*).*
- *Fill the production quota: produce at least* 200 *units (raw material* `D`*), which is actually also the maximum for that raw material.*
- *Produce at most* 10 *units of* `P3`*.*

Again, we want to write a general model, and a separate, corresponding data section for the particular example problem. The starting point is the model and data sections we obtained by solving Problem 11, where the limits were introduced, as both the model and data sections are almost ready, we only need to implement some modifications.

First, we definitely need a new parameter that describes the costs. Let us call this parameter `Material_Cost`. We can simply add it to the model section. No other data is needed.

Before going on, let us observe what happens when material costs are zero. This means that our problem is the same as the original, where raw materials were free. This means that the current problem with raw material costs is a **generalization** of the original problem. Or, in other words, the original problem is a **special case** of the current problem with raw material costs all being zero.

For this reason, it is sensible to give a zero default value to material costs. This makes data files implemented for the original problem compatible with our new model as well.

```
param Material_Cost {r in Raw_Materials}, >=0, default 0;
```

We also know the corresponding data, which can be implemented in the data section. Each raw material has a nonzero cost.

```
param Material_Cost :=
  A  1
  B  0.07
  C  0.013
  D  8
  ;
```

Beyond the data, the only difference in the model is the objective value. We could just change the line and our model would be complete. However, for more readability, we introduce an auxiliary variable for the profit and use it instead. The modified part of the model section is the following. Only three lines of code changed: the `profit` variable was introduced, the objective changed, and the `Profit_Calc` constraint was added to ensure that the `profit` variable obtain the corresponding value. We also print it after the `solve` statement.

```
var production {p in Products}, >=Min_Production[p], <=Max_Production[p];
var usage {r in Raw_Materials}, >=Min_Usage[r], <=Storage[r];
var total_revenue;
var profit;

s.t. Usage_Calc {r in Raw_Materials}:
    sum {p in Products} Consumption_Rate[r,p] * production[p] = usage[r];
s.t. Total_Revenue_Calc: total_revenue =
    sum {p in Products} Revenue[p] * production[p];
s.t. Profit_Calc: profit = total_revenue -
    sum {r in Raw_Materials} Material_Cost[r] * usage[r];

maximize Profit: profit;

solve;

printf "Total Revenue: %g\n", total_revenue;
printf "Profit: %g\n", profit;
```

Now both the model and the data sections are ready. If we optimize for profit now, we get the following result.

```
Total Revenue: 22453.9
Profit: 1577.45
Production of P1: 25.4839
Production of P2: 164.516
Production of P3: 10
Usage of A: 13322.6, remaining: 9677.42
Usage of B: 31000, remaining: 0
Usage of C: 291065, remaining: 158935
Usage of D: 200, remaining: 0
```

Now let us investigate this solution a bit. Now, it is true that the search space of the problem did not change in logic: exactly the same solutions are feasible in both the original problem where revenue is maximized, and the current problem where profit is maximized. This explains the total revenues. In the original problem, the optimal (maximal) total revenue was 32970, it was obtained by producing 90 units of `P1`, 100 units of `P2` and 10 units of `P3`. However, if we optimize for profit instead, we get a total revenue of 22453.87, which is worse. The profit in this case is much smaller, 1577.45, so most of the revenue is diminished by raw material costs. The production is also slightly different, now 25.48 units of `P1`, 164.52 units of `P2` and 10 units of `P3` are produced.

Although the revenue is significantly higher for the original solution than for the current, we can now be sure that the profit is no more than 1577.45 in that case either.

5.5 Diet problem

We have seen several versions for the production problem. Now a seemingly unrelated problem is considered, which is called the **diet problem** [13, 14], or the nutrition problem.

Problem 15.
Given a set of food types, and a set of nutrients. Each food consists of a given, fixed ratio of the nutrients.
We aim to arrange a diet, which is any combination of the set of food types, in any amounts. However, for each nutrient, there is a minimum requirement that the diet must satisfy in order to be healthy. Also, each food has its own proportional cost.
Find the healthy diet with the lowest total cost of food involved.

After the general problem definition, let us see a particular example.

Problem 16.
Solve the diet problem with the following data. There are five food types, named `F1` *to* `F5`*, and there are four nutrients under focus, named* `N1` *to* `N4`*. The contents of unit amount of each food, the unit cost of each food, and the minimum requirement of each nutrient are shown in the following table.*

	N1	*N2*	*N3*	*N4*	*Cost*
F1	30	5.2	0.2	0.0001	450
F2	20	0	0.7	0.0001	220
F3	25	2	0.1	0.0001	675
F4	13	3.6	0	0.0002	120
F5	19	0.1	0	0.0009	500
Required	2000	180	30	0.04	

The diet problem is regarded as one of the first problems that raised the field of Operations Research.

Note that dimensions are slightly different at the diet problem and the production problem. Here we omitted fictional physical dimensions, but the scales of each nutrient suggests which are the data that are of one dimension. In this data table, each column corresponds for a single nutrient, and therefore it has its own unit of measure. The last column shows unit costs.

Usually, when we want to implement a model, the first thing we must decide is how our freedom of choice would be implemented as decision variables, then how the search space can be described by constraints, and how the objective can be calculated. These steps are essential if we first want to decide about a real-world problem whether a mathematical programming approach is suitable for it at all.

However, we now instead start by defining all the data that are available in the problem, and then we will go on with the procedure above. This makes defining the variables, constraints and objectives easier afterwards. So the first part of modeling in GNU MathProg is now defining the sets and parameters. These are either calculated on spot, or provided afterwards in a data section at the end of the model file or separate data file(s).

Two sets appear in the problem. One is for food types, and one is for nutrients. These sets will be used in indexing expressions.

```
set FoodTypes;
set Nutrients;
```

There are three parameters available. One is for denoting food costs, obviously defined for each food type. One parameter is for denoting minimum required amount of nutrients, defined for each nutrient. Finally, one parameter is for the contents of food, which can be given by a unit amount for each pair of food type and nutrient. In the particular problem, for example, each unit of `F2` contains 20 units of `N1`, 0.7 units of `N3` and 0.0001 units of `N4`. The default values are all set to zero, and all these parameters are nonnegative by nature.

```
param Food_Cost {f in FoodTypes}, >=0, default 0;
param Content {f in FoodTypes, n in Nutrients}, >=0, default 0;
param Requirement {n in Nutrients}, >=0, default 0;
```

Now that all the data are defined as sets and variables, we must identify our freedom of choice. The amounts of each food used is clearly under decision. On the other hand, if we know these amounts, then we can easily calculate the total cost, and nutrient contents as well, so we can decide whether the diet is healthy, and if so, how much it costs. This means that we should define a variable denoting food consumption, for each food type separately, and no more variables are needed in the model.

We introduce a variable named `eaten` to denote the amount of each food included in the diet. This variable is obviously nonnegative, and is indexed over the set of food types. We also introduce the auxiliary variable for the total costs so that it can be printed out easier. We could define bounds for the auxiliary variable, but it is unnecessary as its value will be exactly calculated by a constraint.

```
var eaten {f in FoodTypes}, >=0;
var total_costs;
```

There is one significant factor in our model, which restricts which diets are acceptable, and this is total nutritional content. This is a constraint for each nutrient. The total amount contained in the selected diet must be added up, and this must not be less than the minimal requirement for that particular nutrient.

Also, there is another constraint for calculating the auxiliary variable denoting the total food costs.

```
s.t. Nutrient_Requirements {n in Nutrients}:
    sum {f in FoodTypes} Content[f,n] * eaten[f] >= Requirement[n];
s.t. Total_Costs_Calc: total_costs =
    sum {f in FoodTypes} Food_Cost[f] * eaten[f];
```

With the auxiliary `total_costs` variable, defining the objective is straightforward.

```
minimize Total_Costs: total_costs;
```

After the solve statement, again we can write our own printing code that will show the solution found by the solver. This time, the amount for each food in the diet are shown, as well as the total consumption per nutrient, together with the lower limit. Our model section is now ready.

```
set FoodTypes;
set Nutrients;

param Food_Cost {f in FoodTypes}, >=0, default 0;
param Content {f in FoodTypes, n in Nutrients}, >=0, default 0;
param Requirement {n in Nutrients}, >=0, default 0;

var eaten {f in FoodTypes}, >=0;
var total_costs;

s.t. Nutrient_Requirements {n in Nutrients}:
    sum {f in FoodTypes} Content[f,n] * eaten[f] >= Requirement[n];
s.t. Total_Costs_Calc: total_costs =
    sum {f in FoodTypes} Food_Cost[f] * eaten[f];

minimize Total_Costs: total_costs;

solve;

printf "Total Costs: %g\n", total_costs;

param Nutrient_Intake {n in Nutrients} :=
    sum {f in FoodTypes} Content[f,n] * eaten[f];

for {f in FoodTypes}
{
  printf "Eaten of %s: %g\n", f, eaten[f];
}

for {n in Nutrients}
{
  printf "Requirement %g of nutrient %s done with %g\n",
      Requirement[n], n, Nutrient_Intake[n];
}

end;
```

Now we also implement the particular problem mentioned. It is described in a single data section as follows.

```
data;

set FoodTypes := F1 F2 F3 F4 F5;
set Nutrients := N1 N2 N3 N4;

param Food_Cost :=
  F1  450
  F2  220
  F3  675
  F4  120
  F5  500
  ;

param Content:
      N1   N2   N3       N4 :=
  F1   30  5.2  0.2  0.0001
  F2   20    .  0.7  0.0001
  F3   25    2  0.1  0.0001
  F4   13  3.6    .  0.0002
  F5   19  0.1    .  0.0009
  ;

param Requirement :=
  N1  2000
  N2  180
  N3  30
  N4  0.04
  ;

end;
```

Now we can solve the problem and the result is the following.

```
Total Costs: 29707.2
Eaten of F1: 0
Eaten of F2: 42.8571
Eaten of F3: 0
Eaten of F4: 49.2014
Eaten of F5: 28.7489
Requirement 2000 of nutrient N1 done with 2042.99
Requirement 180 of nutrient N2 done with 180
Requirement 30 of nutrient N3 done with 30
Requirement 0.04 of nutrient N4 done with 0.04
```

The optimal objective is 29707.2. This result literally means that any diet containing at least 2000 units of `N1`, 180 units of `N2`, 30 units of `N3` and 0.04 units of `N4`, and consisting of `F1` to `F5`, would costs at least 29707.2, and there exists a solution to obtain exactly this number.

The optimal solution only uses `F2`, `F4` and `F5`. This means that if `F1` and `F3` were omitted from the problem data at all, then the optimal solution would be exactly the same. This is because introducing a new food type only increases the freedom in the model, leaving any previously feasible solutions as feasible afterwards, and possibly introducing new feasible solutions which utilize the newly introduced food type. We can also observe that `N1` is the only nutrient for which the consumption in the optimal diet is more than the minimal amount required, with 2042.99 units rather than 2000. Of the other three nutrients, there is exactly enough. This suggests that the solution shows a diet which is balanced to the edge to meet the minimum requirements while optimizing costs.

5.6 Arbitrary recipes

The implementation of the diet problem and the production problem are surprisingly similar. This is true for the number of sets, parameters, variables, constraints, the contents of the constraints, and the objective. In this part, we will show how the diet problem can be regarded as a production problem. Finally, we will show a production problem with arbitrary recipes that generalizes both problems at the same time.

There are two ways a diet problem can be represented as a production problem.

The first way is when *products represent the food types, and raw materials represent the nutrients.* This can be logical in reality as well. The food can be treated as if it were „produced" from its nutrients, in given ratios. In the diet, we eat the products and decompose it to its nutrients, so the process is just reversed in time. The products have costs, as foods, and the amounts exactly define the solution. The only difference is that instead of „storage" amounts for raw materials, which serve as an upper bound for usage, we have a lower bound because each nutrient must be used in a minimal total amount in order to obtain a good diet. But this feature is already implemented in the limits extension, in Problem 10. Another issue is that in this case, food „production" shall be minimized instead of maximized, but that can be easily achieved if food costs are represented by negative revenues in the model. So technically we would have no difficulty in rewriting our diet problem into a production problem with limits and negative unit revenues. This works if we consider the food type as the products and the nutrients as the raw materials.

However, there is another representation which suggests a more general model behind both the production problem and the diet problem. The second way is when *products represent the nutrients, and raw materials represent the food types.* This matches the process in time: the foods are the „inputs" which are available first, and then the products are the nutrients which we want to obtain by the whole process. There are more differences in this representation for which it may seem unnatural. There are no upper or lower limits for foods, but there is a lower limit for nutrients, which means there would be a minimum production amount for each product in the production problem. There are no costs for the nutrients, but for the foods, which means that in the production problem there would be only costs for the raw materials, but zero revenues for all products.

But the most important difference, which actually prevents us in this second representation to utilize the production problem model directly: the logic of production is reversed. That means, in the production problem there are *many raw materials producing a single product*, in given ratios. However, in the case of the diet problem there would be *a single raw material (food) producing many products*, in given ratios.

So at this point we could say that the second representation is erroneous, and we therefore use only the first one. Technically nothing prevents us from doing that. However, the second representation suggests a valuable generalization for the production problem itself: what if we relax the rule that there is only a single product in each production step? This leads to **the production problem with arbitrary recipes**.

A **recipe** describes a process that consumes several inputs at once, and produces several outputs at once, in given amounts. Each recipe can be **utilized** in an arbitrary **volume**, and both inputs and outputs are proportionally sized according to this volume.

We can see that this concept of recipes may describe the production problem as well as the diet problem.

- In the production problem, there is one recipe for each product. That recipe only produces that particular product as output, but can consume any given combination of raw materials as inputs.
- In the diet problem, there is one recipe for each food type. That recipe consumes only that particular food type as input, but can produce any given combination of products as outputs.
- Moreover, there may be other problems where there are many inputs and many outputs at the same time in a single recipe. These are covered neither by the production nor the diet problem.

Now we can define the production problem with arbitrary recipes as follows.

Problem 17.

Given a set of ***raw materials*** *and a set of* ***products****. There is also a set of* ***recipes*** *defined. Each recipe describes the ratio in which it consumes raw materials, and produces products, these are arbitrary nonnegative numbers. Each recipe may be utilized in an arbitrary amount, which is named its volume.*

There is a ***unit cost*** *defined for each raw material, and a* ***unit revenue*** *for each product.*

There can be minimal and maximal ***total consumption*** *amounts defined for each raw material, and minimal and maximal* ***total production*** *amounts defined for each product.*

For practical purposes, the amounts of consumed raw materials are limited: their total costs cannot exceed a given value, which is the initial ***funds*** *available for raw material purchase.*

Find the optimal production, where recipes are utilized in arbitrary volumes so that all the limits on consumption and production are satisfied, and the ***total profit*** *is maximal. The total profit is the difference of the total revenue from products and the total costs of raw materials consumed.*

Now let us try to implement this problem first without a particular example problem. We will find it is very similar in implementation to the original production problem. The first thing to do is to „read" all the available data for further use. For this reason, there are three sets in the model: the set of raw materials, the set of products, and the set of recipes.

```
set Raws;
set Products;
set Recipes;
```

Note that the problem definition does not exclude the case where the same material is both a *raw material and a product at the same time* in the recipes. It is actually natural in real-world situations: some material is produced by a recipe, and consumed by another. However, questions would arise if we wanted to take this case into account. For example, how to consider timing? If a material is consumed as a raw material in a second recipe, then it must first be produced by another in the first recipe. So production according to the first recipe must happen *before* the execution of the second. Or, it is possible that the production describes an equilibrium where amounts must be maintained, therefore timing is not important at all. There are many possibilities, problem definitions and implementation would be complex, so we do not go into the details. For the sake of simplicity, we assume that *raw materials and products are distinct.*

To ensure that raw materials are indeed distinct from products a `check` statement is introduced. Literally, we express that the intersection of the set of raw materials and the set of products shall contain exactly zero elements. If this does not hold, then model construction fails immediately, as it should.

```
check card(Raws inter Products) == 0;
```

For each recipe, there are ratios, for both the raw materials and the products. Because these are two different sets, it would require two different parameters: one for ratios of raw materials in each recipe, and one for ratios of products in each recipe. This holds for other parameters as well. Instead, we introduce the concept of **materials** in our model. We simply call raw materials and products together as materials. In GNU MathProg, it is legal to introduce a set which is calculated on spot based on other sets.

```
set Materials := Products union Raws;
```

Note that since we assumed that there is no material that is both a product and a raw material, we can further assume that each raw material is represented in the union once, as well as each product, and these are all distinct. Now, with the new set, we can define the necessary parameters in a compact way.

```
param Min_Usage {m in Materials}, >=0, default 0;
param Max_Usage {m in Materials}, >=Min_Usage[m], default 1e100;
param Value {m in Materials}, >=0, default 0;
param Recipe_Ratio {c in Recipes, m in Materials}, >=0, default 0;
param Initial_Funds, >=0, default 1e100;
```

Parameters `Min_Usage` and `Max_Usage` denote the lower and upper bound for each material in the model. These two parameters are indexed over the `Materials` set. For raw materials, the `Min_Usage` and `Max_Usage` mean limits for total consumption. For products, these parameters mean limits for total production. Because each raw material and each product is represented in the `Materials` set exactly once, this definition unambiguously describes all the limits for both raw materials and products. Note that the default limits are 0 for lower and a very large number, 10^{100} for upper limits. So technically there is no limit by default.

The `Value` parameter works similarly, it represents raw material costs in case of raw materials, and revenues in case of products. Both are nonnegative and defaulted to zero.

The `Recipe_Ratio` parameter is for describing recipes. The only needed data for the recipes are the exact amount of inputs consumed and outputs produced. With the common `Materials` set, this can be done with a single parameter. `Recipe_Ratio` is defined for all recipes and all materials. If the material is a raw material, it describes consumption amount, if it is a product, it describes production amount. We call this parameter „ratio" because it corresponds to the amounts consumed and produced when utilizing the recipe with a volume of 1. In general, because both inputs and outputs are proportional, the ratio must be multiplied by the volume to obtain the amounts consumed or produced.

Finally, there is a single numeric parameter `Initial_Funds`. This serves as an upper limit for raw material consumption. In practice, it is generally not possible to invest into any amounts of raw materials, there is usually a cap on this. Note that without such a restriction and any upper limits for consumption or production, it may be possible to gain unlimited profit by consuming an unlimited amount of raw materials to produce at least one product in unlimited amounts. By default, `Initial_Funds` is set to the extreme value and 10^{100} again so that it does not change the model.

Now all the parameters and sets are defined, let us see what the freedom in our model is. What we have to decide is the volume for each recipe utilized. This is slightly different than before, because decisions do not correspond to one particular product or raw material, but for a given recipe. If recipe amounts are defined, then the solution is exactly determined and all other information can be calculated, including raw material usages, production amounts, costs, revenues, the profit, and corresponding limitations.

```
var volume {c in Recipes}, >=0;
var usage {m in Materials}, >=Min_Usage[m], <=Max_Usage[m];
var total_costs, <=Initial_Funds;
var total_revenue;
var profit;
```

In our implementation `volume` is the variable which denotes for each recipe, which volume it shall be utilized in. This is a nonnegative value but can be zero and even fractional, as usual. As we mentioned, this only variable would be sufficient for model formulation, but we want to write a compact and readable model instead, so we introduce a few auxiliary variables as well.

The variable `usage` is the total „usage" of each material. It is indexed over the `Materials` set, and works similarly to the parameters, it has a slightly different meaning for raw materials and products, but for sake of simplicity it can be denoted by the same single variable. For raw materials `r`, `usage[r]` is the total consumption, while for products `p`, `usage[p]` is the total production amount. Note that we can give `Min_Usage[m]` and `Max_Usage[m]` as bounds for this variable, which implements the limitations in our problem.

There is also a variable named `total_costs` which denotes the total costs of consumed raw materials, a variable `total_revenue` for the total revenue from products, and finally, a variable `profit` for the difference, which is actually our objective function. We set `Initial_Funds` as an upper bound for `total_costs`, which implements the maximum usage limitation.

Constraints are implemented next.

```
s.t. Material_Balance {m in Materials}: usage[m] =
    sum {c in Recipes} Recipe_Ratio[c,m] * volume[c];
s.t. Total_Costs_Calc: total_costs = sum {r in Raws} Value[r] * usage[r];
s.t. Total_Revenue_Calc: total_revenue =
    sum {p in Products} Value[p] * usage[p];
s.t. Profit_Calc: profit = total_revenue - total_costs;
```

Although this model is intended to be a generalization for both the production problem and the diet problem, with supporting most of the features mentioned so far, there is only one key constraint: the **material balance** established by recipe utilization. The constraint says that the usage of each *material*, regardless of being a *raw material* or a *product*, is calculated by adding for each recipe its volume, multiplied by the ratio the material is represented in the recipe. This is exactly the same for both the original production problem and the diet problem.

We also define three additional constraints to calculate the values of the auxiliary variables `total_costs`, `total_revenue` and `profit`. Note that even though parameters and variables (here `Value` and `usage`) are indexed over the `Materials` set, it is valid in GNU MathProg to index those parameters and variables over a smaller set. Using the original `Raws` and `Products` sets, we can sum up only for raw materials and only for products. Be careful, we can only index over the original domain, or its subset, otherwise we get an out of domain error (and the model is also guaranteed to be logically wrong).

The objective is straightforward, the profit itself.

```
maximize Profit: profit;
```

After solving the problem, we can print out the auxiliary variables, as well as the utilization volumes for each recipe and total consumption and production amounts for each material. The full model section is ready as follows.

```
set Raws;
set Products;
set Recipes;

check card(Raws inter Products) == 0;
set Materials := Products union Raws;

param Min_Usage {m in Materials}, >=0, default 0;
param Max_Usage {m in Materials}, >=Min_Usage[m], default 1e100;

param Value {m in Materials}, >=0, default 0;
param Recipe_Ratio {c in Recipes, m in Materials}, >=0, default 0;
param Initial_Funds, >=0, default 1e100;

var volume {c in Recipes}, >=0;
var usage {m in Materials}, >=Min_Usage[m], <=Max_Usage[m];
var total_costs, <=Initial_Funds;
var total_revenue;
var profit;

s.t. Material_Balance {m in Materials}: usage[m] =
    sum {c in Recipes} Recipe_Ratio[c,m] * volume[c];
s.t. Total_Costs_Calc: total_costs = sum {r in Raws} Value[r] * usage[r];
s.t. Total_Revenue_Calc: total_revenue =
    sum {p in Products} Value[p] * usage[p];
s.t. Profit_Calc: profit = total_revenue - total_costs;

maximize Profit: profit;

solve;

printf "Total Costs: %g\n", total_costs;
printf "Total Revenue: %g\n", total_revenue;
printf "Profit: %g\n", profit;

for {c in Recipes}
{
  printf "Volume of recipe %s: %g\n", c, volume[c];
}
for {r in Raws}
{
  printf "Consumption of raw %s: %g\n", r, usage[r];
}
for {p in Products}
{
  printf "Production of product %s: %g\n", p, usage[p];
}

end;
```

Although the model for arbitrary recipes is similar in nature to the former models, we still implemented it at once. The question arises: how can a large, complex model be implemented in GNU MathProg from scratch? Or, generally, in any mathematical programming language?

There is no general guide for modeling, but there are good rules of thumb to follow. The recommendation is the following, specifically for GNU MathProg.

1. First, we must decide whether the problem under consideration can be effectively solved by LP (or MILP) models. There are many problems that are simply cannot, or only with very complicated workarounds, or there is a much more suitable algorithm or other method for solving it. This is the hardest part, we basically have to think what the decision variables will be, and how the appropriate search space can be defined by the addition of constraints and

other variables. If we are sure we can implement an LP (or MILP) model for the problem, then we can go on with the implementation of the model file.

2. Collect all data that are available and needed. Define sets and parameters which will be provided by the data sections. Data files can be implemented at this point if there are example problem instances available. If some data is missing or must be calculated afterwards, we will always have the opportunity to introduce other sets and parameters, and calculate other data in the model file.

3. Define the decision variables. Keep in mind that the values of all the variables should exactly determine what is happening in the real-world. In particular, we must be able to calculate the objective and decide for each restriction whether it is violated or not, based on the variables.

4. Implement all possible rules as constraints or bounds. Keep in mind that there are two mistakes we can make: a problem may be **under-constrained** or **over-constrained**, or both. In under-constrained problems, there are solutions left in the search space that are infeasible in the problem but feasible in the model. These additional solutions may be found by the solver and reported as fake optimal solutions. Then, additional constraints must be defined to exclude those solutions, or already existing constraints redefined to be more restrictive. In over-constrained problems, interesting solutions are excluded from the search space, and therefore not found by the solver. Then, some constraints or bounds are too restrictive, we have to reformulate or remove them. Remember that we can always introduce new auxiliary variables in the model.

5. Define the objective.

6. After the `solve` statement, report the relevant details of the solution found, in the desired format.

Complex models may have several dozens of constraints, so how can we be sure we have not forgotten any rules? One idea for that is to focus on parameters or variables. In many cases, parameters are used only once in the model. Even if not, we can enlist all the roles the parameter or variable must appear in the model: as a bound, a constraint, or objective term, etc. Then it is easy to spot one which is forgotten.

Now that we have our model for arbitrary recipes ready, we will demonstrate how this works for all the problems mentioned so far in this chapter (with the exception of the maximum of minimum production amounts case).

First, Problem 14, which introduced raw material costs, is solved. Since this is a pure production problem in the original way, a recipe is introduced to produce each of the products. All the limits are implemented by the `Min_Usage` parameter, while raw material costs and product revenues are implemented by the `Value` parameter. The data section is the following.

```
data;

set Raws := A B C D;
set Products := P1 P2 P3;
set Recipes := MakeP1 MakeP2 MakeP3;

param Min_Usage :=
  B  21000
  D  200
  P2  100
  ;

```

```

param Max_Usage :=
  A  23000
  B  31000
  C  450000
  D  200
  P3  10
  ;

param Value :=
  A  1
  B  0.07
  C  0.013
  D  8
  P1  252
  P2  89
  P3  139
  ;

param Recipe_Ratio:
            A    B     C  D  P1  P2  P3 :=
  MakeP1  200   25  3200  1   1   0   0
  MakeP2   50  180  1000  1   0   1   0
  MakeP3    0   75  4500  1   0   0   1
  ;

end;
```

If we solve it, then we get exactly the same result as for the original model with raw costs. The optimal profit is 1577.45, with production of 25.48 units of P1, 164.52 units of P2 and 10 units of P3. The output is the following.

```
Total Costs: 20876.4
Total Revenue: 22453.9
Profit: 1577.45
Volume of recipe MakeP1: 25.4839
Volume of recipe MakeP2: 164.516
Volume of recipe MakeP3: 10
Consumption of raw A: 13322.6
Consumption of raw B: 31000
Consumption of raw C: 291065
Consumption of raw D: 200
Production of product P1: 25.4839
Production of product P2: 164.516
Production of product P3: 10
```

The second application is the diet problem. We solve exactly the same problem instance as in Problem 16. In this example, the food types are the raw materials and the nutrients are the products we want to obtain. Contrary to the original production problem, where there were several inputs and one output per recipe, here there is only one input, a food type per recipe, which produces several nutrients with given ratios.

```
data;

set Raws := F1 F2 F3 F4 F5;
set Products := N1 N2 N3 N4;
set Recipes := EatF1 EatF2 EatF3 EatF4 EatF5;

param Min_Usage :=
  N1  2000
  N2  180
  N3  30
  N4  0.04
  ;

param Value :=
  F1  450
  F2  220
  F3  675
  F4  120
  F5  500
  ;

param Recipe_Ratio:
         F1  F2  F3  F4  F5  N1   N2   N3      N4 :=
  EatF1   1   0   0   0   0  30  5.2  0.2  0.0001
  EatF2   0   1   0   0   0  20    .  0.7  0.0001
  EatF3   0   0   1   0   0  25    2  0.1  0.0001
  EatF4   0   0   0   1   0  13  3.6    .  0.0002
  EatF5   0   0   0   0   1  19  0.1    .  0.0009
  ;

end;
```

The solution is again exactly the same as for the original diet problem, which is an optimal cost of 29707.2, with food consumption amounts of 24.86 of F2, 49.20 of F4, and 28.75 of F5. Note that in this case, the objective reported by the solver is −29707.2, because the profit is determined solely by the food costs.

```
Total Costs: 29707.2
Total Revenue: 0
Profit: -29707.2
Volume of recipe EatF1: 0
Volume of recipe EatF2: 42.8571
Volume of recipe EatF3: 0
Volume of recipe EatF4: 49.2014
Volume of recipe EatF5: 28.7489
Consumption of raw F1: 0
Consumption of raw F2: 42.8571
Consumption of raw F3: 0
Consumption of raw F4: 49.2014
Consumption of raw F5: 28.7489
```

```
Production of product N1: 2042.99
Production of product N2: 180
Production of product N3: 30
Production of product N4: 0.04
```

Recall that we mentioned two ways a diet problem can be represented as a production problem. Now the second of them was implemented, which is the case when food types are the raw materials, and nutrients are the products. But how could we implement the representation of the first case, when food types are the products and nutrients are the raw materials?

Surprisingly, this arbitrary recipe model is able to do that as well, just the role of the products and raw materials must be exchanged. If we understand that, we can see that there is a very high degree of symmetry in the production problem of arbitrary recipes. There are minimum and maximum usages, for both the raw materials and the products. The cost of a raw material is the counterpart of the revenue of a product. If we look at the recipes, we can see that there is no „source-target" relation between raw materials and products. These two roles are interchangeable. The only slight difference between raw materials and products that breaks the symmetry is the `Initial_Funds` feature which gives an upper limit for the total of raw material costs. The symmetry would be perfect if `Initial_Funds` was set to infinity, or if another feature like maximal revenue was also introduced.

Finally, let us see a new example problem to further demonstrate the capabilities of the arbitrary recipe model. The starting point for the problem instance is Problem 14 where raw material costs were introduced, but we do a few modifications.

Problem 18.

Solve Problem 14, the original production problem, where data are exactly the same, but there are two other options for production.

- *`P1` and `P2` can be produced jointly, with slightly different consumption amounts than separately. Producing one unit of both products requires 240 units of raw material `A` (instead of 250 when done separately), 200 units of raw material `B` (instead of 205 when done separately), 4400 units of raw material `C` (slightly more than 4200 when done separately), and 2 units of raw material `D` (exactly as if done separately).*
- *Similarly, `P2` and `P3` can also be produced jointly. The costs are 51 units of raw material `A` (slightly more than 50 when done separately), 250 units of raw material `B` (instead of 255 when done separately), and 5400 units of raw material `C` (instead of 5500 when done separately), and 2 units of raw material `D` (exactly as if done separately).*

This problem can be solved by the manipulation of the data section of the original problem. We have to add two additional recipes to the `Recipes` set, then two rows to the `Recipe_Ratios` parameter to describe these two new recipes. The data section and the results are the following.

```
data;

set Raws := A B C D;
set Products := P1 P2 P3;
set Recipes := MakeP1 MakeP2 MakeP3 Comp1 Comp2;

param Min_Usage :=
  B  21000
  D  200

  P2  100
  ;

param Max_Usage :=
  A  23000
  B  31000
  C  450000
  D  200
  P3  10
  ;

param Value :=
  A  1
  B  0.07
  C  0.013
  D  8
  P1  252
  P2  89
  P3  139
  ;

param Recipe_Ratio:
             A    B     C  D  P1  P2  P3 :=
    MakeP1  200   25  3200  1   1   0   0
    MakeP2   50  180  1000  1   0   1   0
    MakeP3    0   75  4500  1   0   0   1
    Comp1   240  200  4400  2   1   1   0
    Comp2    51  250  5400  2   0   1   1
    ;

end;
```

```
Total Costs: 30495
Total Revenue: 32460.6
Profit: 1965.62
Volume of recipe MakeP1: 0
Volume of recipe MakeP2: 6.25
Volume of recipe MakeP3: 0
Volume of recipe Comp1: 86.875
Volume of recipe Comp2: 10
Consumption of raw A: 21672.5
Consumption of raw B: 21000
Consumption of raw C: 442500
Consumption of raw D: 200
Production of product P1: 86.875
Production of product P2: 103.125
Production of product P3: 10
```

Now it turns out that the optimal solution is 1965.62, which is a bit better than the original 1577.45. This is due to the fact that the modification was only the addition of new opportunities to the production, so the search space was widened. We can also see that `Comp1` and `Comp2`, which are the recipes for the joint production methods, are used as an alternative instead of the original options. There is still a little 6.25 units `P2` produced alone, though. The total production is also slightly different in this solution, with 86.88 units of `P1`, 103.13 units of `P2` and 10 units of `P3`.

5.7 Order fulfillment

Now, we have a complete model for the production problem of arbitrary recipes, with which we can easily implement and solve a wide range of problems, including the diet problem. We further extend the model with a new feature, which has practical importance: orders. These allow to buy and sell materials in a bulk, which is possibly a more lucrative option but can only be acquired entirely or ignored at all.

So far in the production problem topic, we only had real-valued variables, therefore the models were all LP models. Now, we will have integer variables, making the model an MILP model. We must note that MILP models can be easily implemented, but the solution procedure can be unacceptably long if there are *too many binary variables*. The limit on what is considered too many depends on the model: in some problems we can only have several dozens of integer variables, while sometimes several hundreds or even thousands will be working. Anyways, this limit is definitely lower than for the number of ordinary real-valued variables in LP models. Unfortunately in many situations, the limitations can only slightly be pushed up by choosing stronger equipment, better solver or modeling techniques, because integer programming is an NP-Complete problem. Nevertheless, using integer variables is a very powerful modeling tool, which allows a significantly wider range of problems to be possibly modeled by MILP than those by LP.

We will see an example how a GNU MathProg model can be extended by new features *while maintaining compatibility with old data files.* We will see how we can **filter in an indexing expression** in GNU MathProg.

An **order** is a fixed amount of several raw materials that are purchased and/or products which are sold when ready by the producing party. An order can also either cost money, or gain income as money, and payment may happen either before or after all the production takes place. The same order may be served multiple times.

The general problem definition for arbitrary recipes and order fulfillment is the following.

Problem 19.

Solve Problem 17, the production problem with arbitrary recipes, where production goes as usual, but there are ***orders we may acquire****. Orders are optional but can only be acquired (and subsequently fulfilled) completely, not partially. Each order consists of the following.*

- *A fixed amounts of raw materials and products. If a raw material is in an order, it means that by acquiring the order, we obtain that raw material in the given amount before production. If a product is in an order, it means that we deliver it after produced, in the given amount.*
- *Price of the order. It can either be a gain or a payment. Payment happens either before or after the production takes place.*
- *Maximum count. An order can be acquired multiple times, with an upper limit.*

Raw materials must either be purchased from the market, as usual, or purchased via acquiring an order. Leftover raw materials after production took place are lost without compensation.

Products must either be sold on the market, or delivered via an order. The only way of obtaining products is by producing it. Fulfilling orders is mandatory once acquired.

Minimum and maximum usage limitations still apply as before. Limitations correspond to the total amounts that are in possession at the same time.

The ***total costs*** *include incomes and expenses of those orders for which the payment is due before production, plus the total costs of raw materials that are purchased from the market in the ordinary way. The total costs are limited: there is a fixed amount of initial funds which we cannot exceed.*

The ***total revenue*** *includes incomes and expenses of those orders for which the payment is due after production, plus the total revenue from selling products at the market in the ordinary way.*

Optimize for ***profit****, which is the difference of the total revenue and total costs.*

The first observation shall be the point that although there is much description in this problem definition, if we suppose that there are no orders in the problem, then we get exactly to the production problem with arbitrary recipes. Let us see why this is true. If there were no orders, then the only way of getting raw materials is by purchasing from the market, and the only way to gain revenue is by selling products at the market. Of course, we purchase exactly as many raw materials as needed, and sell all the products we produced. There is no potential loss of materials, or alternatives.

For this reason, we expect the new model to be working with data files of the „old" model of arbitrary recipes. In short, *compatibility shall be maintained.*

First, let us see how the extra data can be implemented in the model as sets and parameters.

```
set Orders, default {};

param Order_Material_Flow {o in Orders, m in Materials}, >=0, default 0;
param Order_Cash_Flow {o in Orders}, default 0;
param Order_Count {o in Orders}, >=0, integer, default 1;
param Order_Pay_Before {o in Orders}, binary, default 1;
```

There is one additional set, `Orders`. Note that in this case, a default value of `{}` is used. This is an empty one-dimensional set in GNU MathProg. By this default value we can ensure that the original data files will work as they are already written, even if we do not mention the `Orders` set.

There are four parameters that describe orders.

The first one is `Order_Material_Flow`. The meaning of this parameter is similar to the meaning of `Recipe_Ratio` in the arbitrary recipes implementation. If `m` is a raw material, then the order denotes a purchase in the given amount. If `m` is a product, then the order denotes a delivery. By default, a material does not appear in an order, which is indicated by zero material flow.

The `Order_Cash_Flow` parameter denotes the cash flow associated with the order. Note that this is the only parameter which can be negative. A positive value means a cost that must be paid for acquiring the order. A negative value means a revenue from the order in cash, if acquired, and of course, if the production goals are met.

Note that a zero `Order_Cash_Flow` still has a practical relevance. The other party may only want to exchange raw materials with products.

On the other hand, there is no point in orders where `Order_Material_Flow` values are zero for all materials, because cash flow only occurs once, either before or after production. Nevertheless, it would be easy to implement cash flows both before and after production, and in this case, orders with no material flows may represent *investments*: expenses at present, which give more income in the future.

Remember that orders can be acquired multiple times, multiplying all the material flows, but also the prices and revenues. The parameter `Order_Count` is for denoting how many times and order can be acquired. We also set this parameter to only take nonnegative integer values by the `>=0` bound and `integer` keyword. This prevents fractional values to be provided in data sections. If the value of the `Order_Count` parameter is 4, for example, then we can completely ignore the order, or acquire it 1, 2, 3 or 4 times, multiplying all its effects on cash and material flow. The default value is 1, which means that the order is either acquired once or not at all.

Finally, the Order_Pay_Before parameter denotes whether cash flow occurs before or after the production takes place. It is important because the total funds are limited before the production, but profit is calculated afterwards. We used the binary keyword to denote that this parameter may only take the values 0 or 1. As a convention, a value of 1 means a *true*, that is, payment is due before production, and a 0 means that it occurs after production was done. The default is 1, so cash flow occurs before production.

Note that both the binary and integer keywords indicate the same value restriction for parameters and variables.

After defining all our necessary data definitions in our model section, we define decision variables which express our freedom of choice in the optimization model.

```
var volume {c in Recipes}, >=0;
var total_costs, <=Initial_Funds;
var total_revenue;
var profit;
var ordcnt {o in Orders}, integer, >=0, <=Order_Count[o];
```

The volume main variable for recipes, and auxiliary variables total_costs, total_revenue and profit are left unchanged.

A new ordcnt variable is introduced to denote how many times a given order is acquired. This is limited by the Order_Count parameter value of each order, but always nonnegative, and most importantly, it is an *integer* variable. It must be an integer, because we either fulfill an order (1), or ignore it (0), but we cannot fulfill an order partly. Therefore ordcnt can take 0 or 1 (and larger integers), but cannot take fractional values in between.

The ordcnt is the only variable that makes the model an MILP model instead of LP. The complexity of the model depends on how many orders are there and how many times they may be acquired.

The usage of the raw materials and products remains to be defined. In the implementation of the arbitrary recipes, it was easy, one single usage variable was sufficient for all raw materials and products. But now, the case is a bit more complicated because materials can come from and go to different sources. Let us collect the possibilities.

A raw material appears in the model in the following roles.

- It can be obtained as an input from the market, as usual.
- It can be obtained as an input by acquiring orders.
- It can be used up by production, as usual.
- There can be leftover amounts of raw materials which are obtained but not used up. These are wasted.

We might question why it is possible to have leftover raw materials. In the original problem without the orders, it was impossible. Or, precisely, it was possible in reality, but the model did not allow it as usage was directly calculated based on production requirements. The reason is that there was no point in purchasing more raw materials from the market in the beginning than the amount actually used by production, therefore the amount purchased and consumed was assumed to be equal.

But in this case, there are orders. It may happen that an order contains a large amount of raw materials that cannot be used up by production, meanwhile the order is still needed for other purposes. The fact that orders can only be acquired in a whole makes it possible to have leftover materials.

A product appears in the model in the following ways.

- It can only be obtained by production, as usual.
- It can be turned into revenue by selling it at the market, as usual.
- It can be turned into revenue also by delivery, by fulfilling orders.

In contrast with the raw materials, there is no practical relevance of leftovers in case of products. The reason is that the only way a product can be obtained is by production, which can be freely scaled and does not have an integer nature as raw materials obtained from orders. Or, for a second and more obvious reason, there is no point in keeping a product as leftover instead of selling it at the market.

If we are unsure about the possibilities, it is a good method of modeling to introduce a variable for every quantity appearing, and then specify their relations through constraints. For both raw materials and products, there is activity by orders and market, and also by production. However, only raw materials may have leftovers.

As the problem definition mentioned, usages refer to the total amounts present at the same time. So it is a good idea to introduce a total usage variable as well, for which the appropriate bounds can be specified. The remaining usage variables are the following.

```
var usage_orders {m in Materials}, >=0;
var usage_market {m in Materials}, >=0;
var usage_production {m in Materials}, >=0;
var usage_leftover {r in Raws}, >=0;
var usage_total {m in Materials}, >=Min_Usage[m], <=Max_Usage[m];
```

The model can surely be implemented with less variable definitions, but it is generally easier to introduce more variables than needed, implement a correct model, and if we are not satisfied, then we may think about how the model can be simplified.

Note that all variables are set as nonnegative. This is important, every quantity of materials must be nonnegative. Otherwise, practically infeasible solutions may be reported.

Now materials balance can be established as constraints. The total amounts of raw materials used up at production and products produced both can be calculated the same way. For all recipes, the volume in which it is utilized must be multiplied by the material appearing. Similarly, the amounts of raw materials obtained and products delivered can also be calculated in a common way. We have to add the material flow per order, multiplied by the number of times the order is acquired. These two constraints together means that `usage_production` and `usage_orders` are exactly calculated.

```
s.t. Material_Balance_Production {m in Materials}: usage_production[m] =
    sum {c in Recipes} Recipe_Ratio[c,m] * volume[c];
s.t. Material_Balance_Orders {m in Materials}: usage_orders[m] =
    sum {o in Orders} Order_Material_Flow[o,m] * ordcnt[o];
```

There are a few other usage constraints that must be established. The calculation of the total usage can be done two ways for both raw materials and products. The total usage of raw materials equals the total obtained, which comes from orders and from the market. On the other hand, the total usage of a raw material also equals the amount consumed plus the leftover. Similarly, the total usage of a product can be determined two different ways that lead to equations. First, all the products are obtained by production, so these two variables are equal. On the other hand, all the products are either delivered as an order or sold at the market.

```
s.t. Material_Balance_Total_Raws_1 {r in Raws}:
    usage_total[r] = usage_orders[r] + usage_market[r];
s.t. Material_Balance_Total_Raws_2 {r in Raws}:
    usage_total[r] = usage_production[r] + usage_leftover[r];
s.t. Material_Balance_Total_Products_1 {p in Products}:
    usage_total[p] = usage_production[p];
s.t. Material_Balance_Total_Products_2 {p in Products}:
    usage_total[p] = usage_orders[p] + usage_market[p];
```

We can analyze whether the constraints are correct.

As mentioned, `usage_production` and `usage_orders` are calculated based on other variables of the model (recipes and orders). The total usage calculation is different for raw materials and products.

The equation `Material_Balance_Total_Raws_1` expresses that the total amount must be at least the amount obtained by orders. This is true because `usage_market` is a nonnegative variable. Therefore the total amount is first coming from orders, and then we may additionally purchase from the market. Note that `usage_orders` is a calculated auxiliary variable, but `usage_market` is „free".

The equation `Material_Balance_Total_Raws_2` describes that whichever total amount we obtained by orders and market, it must be used up by production or it is a leftover. The amount for production is calculated, but the leftover amount „can be selected freely" to match the total amount obtained. Again, it is very important that all these variables are nonnegative, for which reason neither the amount used at production nor the amount of leftovers can exceed the total.

The two equations for the products are a bit easier.

Constraint `Material_Balance_Total_Products_1` expresses that the total available amount is the production. This effectively makes `usage_total` itself a calculated auxiliary variable. Then, `Material_Balance_Total_Products_2` ensures that this total must be at least the amount that is delivered, and anything that remains can be sent to the market. Again, ensured by the nonnegativity bounds. Note that `usage_leftover` is a value that we do not care about. It still plays an important role in the model solely because it must be nonnegative. Also, if the problem definition would change, for example to limit, penalize or give a refund for leftover raw materials, it would be easy to implement such a modification because the leftover amount is already represented in a variable.

Remember that it is possible to implement this model with fewer variables and material balance constraints, even at first, but would require some expertise.

There is another set of constraints left, which is the calculation of the total costs, revenue and profit, as for the original model. However, in this case, it is a bit more complicated because orders contribute to costs and revenue.

```
s.t. Total_Costs_Calc: total_costs =
    sum {r in Raws} Value[r] * usage_market[r] +
    sum {o in Orders: Order_Pay_Before[o]} Order_Cash_Flow[o] * ordcnt[o];
s.t. Total_Revenue_Calc: total_revenue =
    sum {p in Products} Value[p] * usage_market[p] -
    sum {o in Orders: !Order_Pay_Before[o]} Order_Cash_Flow[o] * ordcnt[o];
s.t. Profit_Calc: profit = total_revenue - total_costs;
```

The total value marketed is a cost for raw materials, and a revenue for products. However, we must add cash flow from orders. For good reason, we did not model orders with cash gain and payment as different, but simply the sign of the `Order_Cash_Flow` shows us whether it has a cost, or it pays. But there is one thing which cannot be modeled by signs: the timing of cash flow.

- If payment occurs *before* production (denoted by the `Order_Pay_Before` parameter being 1), then the cash flow amount must be added *to the total costs*.

- If payment occurs *after* production (denoted by the `Order_Pay_Before` parameter being 0), then the cash flow amount must be added *to the total revenue*. Note that raw material costs paid after the production are treated as negative revenues instead of costs.

It does not matter whether we gain or pay with an order. If an order is an expense before production, we simply add the positive value of `Order_Cash_Flow` to the total costs. If we have an income before production, then this amount must be *subtracted* from the total costs, because this increases our initial funds that can be spent for other purposes before production. For this reason, the meaning of the variable `total_costs` slightly changes. It does not represent the actual total costs anymore, but the total balance of incomes and expenses before production. But because `Order_Cash_Flow` is negative for incomes and positive for expenses, we simply add its value to the total, in both cases.

Similarly, cash flow by an order after production can either be an income or an expense. If it is an income, it must be added, if it is an expense, it must be subtracted from the total revenue. Because of the sign of `Order_Cash_Flow`, we have to *subtract* this value in both cases.

Finally, let us see how the selective addition is solved in GNU MathProg. The problem is that not all orders must be added to either the costs or the revenue, but only their subset (where `Order_Pay_Before`) has the appropriate value). Therefore, a condition is included into the indexing expression of the sum. This is called **filtering**. It means that instead of iterating over the full index set, only those of them are taken into account for which a given logical expression evaluates true.

```
sum {o in Orders: Order_Pay_Before[o]} Order_Cash_Flow[o] * ordcnt[o];
```

Here the original index set is the set of `o` orders from the `Orders` set. The logical expression must appear after all the indices. Now this expression is `Order_Pay_Before[o]` which evaluates to a number, either 0 or 1, and 1 is the true value. The opposite is done for the revenues where those orders are taken into account where `!Order_Pay_Before[o]` is true, therefore it must be 0.

```
sum {o in Orders: !Order_Pay_Before[o]} Order_Cash_Flow[o] * ordcnt[o]
```

In an indexing expression, even though there can be multiple indices, only one filtering logical expression is allowed, after all indices are listed, following a colon. We can have more or complex filters by logical operator like `&&`, `||` or `!`, these also have more readable aliases `and`, `or` and `not`. There are other operators as well. The criteria is that the expression must evaluate to a constant numeric value. For instance, the filtering expression can only refer to variables after the `solve` statement where variables are determined.

Filtering, as could be guessed, can be used on all indexing expressions. Examples other than the `sum` just so far included `param`, `set`, `var`, `s.t.`, `for` and `check` (there are more statements that can be indexed). The meaning is always the same, the index set is restricted according to the logical expression.

Note that it is valid in GNU MathProg for an indexing expression to be evaluated over an empty set, for example if the logical condition excludes all the indices. In this case, the expression which is indexed is treated similarly as if it was not present in its context at all. For example, a `sum` over an empty set evaluates to zero, or a `s.t.` statement over an empty set does not define any constraint.

Finally, the objective of the model is the profit, and we are ready. In the post-processing work, we can print out some of the values to overview the result. The ultimate model section, for **the production problem with arbitrary recipes and orders**, is the following.

```
set Raws;
set Products;
set Recipes;
set Orders, default {};

check card(Raws inter Products) == 0;
set Materials := Products union Raws;

param Min_Usage {m in Materials}, >=0, default 0;
param Max_Usage {m in Materials}, >=Min_Usage[m], default 1e100;
param Value {m in Materials}, >=0, default 0;
param Recipe_Ratio {c in Recipes, m in Materials}, >=0, default 0;
param Initial_Funds, >=0, default 1e100;
param Order_Material_Flow {o in Orders, m in Materials}, >=0, default 0;
param Order_Cash_Flow {o in Orders}, default 0;
param Order_Count {o in Orders}, >=0, integer, default 1;
param Order_Pay_Before {o in Orders}, binary, default 1;

var volume {c in Recipes}, >=0;
var total_costs, <=Initial_Funds;
var total_revenue;
var profit;
var ordcnt {o in Orders}, integer, >=0, <=Order_Count[o];
var usage_orders {m in Materials}, >=0;
var usage_market {m in Materials}, >=0;
var usage_production {m in Materials}, >=0;
var usage_leftover {r in Raws}, >=0;
var usage_total {m in Materials}, >=Min_Usage[m], <=Max_Usage[m];

s.t. Material_Balance_Production {m in Materials}: usage_production[m] =
    sum {c in Recipes} Recipe_Ratio[c,m] * volume[c];
s.t. Material_Balance_Orders {m in Materials}: usage_orders[m] =
    sum {o in Orders} Order_Material_Flow[o,m] * ordcnt[o];
s.t. Material_Balance_Total_Raws_1 {r in Raws}:
    usage_total[r] = usage_orders[r] + usage_market[r];
s.t. Material_Balance_Total_Raws_2 {r in Raws}:
    usage_total[r] = usage_production[r] + usage_leftover[r];
s.t. Material_Balance_Total_Products_1 {p in Products}:
    usage_total[p] = usage_production[p];
s.t. Material_Balance_Total_Products_2 {p in Products}:
    usage_total[p] = usage_orders[p] + usage_market[p];

s.t. Total_Costs_Calc: total_costs =
    sum {r in Raws} Value[r] * usage_market[r] +
    sum {o in Orders: Order_Pay_Before[o]} Order_Cash_Flow[o] * ordcnt[o];
s.t. Total_Revenue_Calc: total_revenue =
    sum {p in Products} Value[p] * usage_market[p] -
    sum {o in Orders: !Order_Pay_Before[o]} Order_Cash_Flow[o] * ordcnt[o];
s.t. Profit_Calc: profit = total_revenue - total_costs;

maximize Profit: profit;

solve;
```

```
printf "Total Costs: %g\n", total_costs;
printf "Total Revenue: %g\n", total_revenue;
printf "Profit: %g\n", profit;

for {o in Orders}
{
  printf "Acquiring order %s: %dx\n", o, ordcnt[o];
}
for {c in Recipes}
{
  printf "Volume of recipe %s: %g\n", c, volume[c];
}
printf "Raw materials (orders + market -> production + leftover):\n";
for {r in Raws}
{
  printf "Consumption of raw %s: %g + %g -> %g + %g (total: %g)\n",
      r, usage_orders[r], usage_market[r], usage_production[r],
      usage_leftover[r], usage_total[r];
}
printf "Products (production -> orders + market):\n";
for {p in Products}
{
  printf "Production of product %s: %g -> %g + %g (total: %g)\n",
      p, usage_production[p], usage_orders[p],
      usage_market[p], usage_total[p];
}

end;
```

Remember that the order of constraints which is presented here is only important for us to understand what the model does. The order of constraints and variables does not affect the model itself and its solutions (although it might possibly affect the solution algorithm, but definitely not the final answer). On the other hand, it is useful to establish some logical order of the variables and/or constraints, to understand the model we implement. For example, a logical order can be the following which explains the model code.

1. We decide how many times each order is acquired (variable `ordcnt`).
2. Then, we calculate the amount obtained or delivered as part of orders (variable `usage_order`), based on our former decisions on `ordcnt`.
3. We decide how much production we want to perform (variable `volume`).
4. Then, we calculate the amount of raw materials and products present in the production process (variable `usage_production`), based on our former decisions on `volume`.
5. We adjust the amounts of raw materials purchased and products sold at the market (variable `usage_market`) and then calculate the total amounts (variable `usage_total`) so that there is enough raw material, and all products are sold, based on our former decisions on production, orders, and market activity.
6. Then, we calculate the leftover amount of raw materials (variable `usage_leftover`) based on decisions of other usages.

7. Finally, we calculate the total costs, the total revenue, and the objective function, which is the profit. Their values are based on former decisions.

Meanwhile, if one of the appearing quantities are limited (like usage, or order count), then a corresponding constraint or bound must be formulated.

The optimization procedure is making these decisions to maximize the profit obtained at the final step. But for the technical, solution algorithmic point of view, the above logical order does not matter, statements in GNU MathProg can be defined in any order, provided that each time we refer to a model object, then its definition is before that reference.

Now try the model with some valid data. The first example is the problem instance introduced to demonstrate raw material costs, Problem 14. This was solved before.

```
data;

set Raws := A B C D;
set Products := P1 P2 P3;
set Recipes := MakeP1 MakeP2 MakeP3;

param Min_Usage :=
  B  21000
  D  200
  P2  100
  ;

param Max_Usage :=
  A  23000
  B  31000
  C  450000
  D  200
  P3  10
  ;

param Value :=
  A  1
  B  0.07
  C  0.013
  D  8
  P1  252
  P2  89
  P3  139
  ;

param Recipe_Ratio:
           A    B     C  D  P1  P2  P3 :=
  MakeP1  200   25  3200  1   1   0   0
  MakeP2   50  180  1000  1   0   1   0
  MakeP3    0   75  4500  1   0   0   1
  ;

end;
```

The optimal solution is the same as for the original model for arbitrary recipes, 25.48 units of

P1, 164.52 units of P2 and 10 units of P3, yielding a profit of 1577.45. What is important here is that the data section is exactly the same as for the original model. The extension with orders also works if no orders are defined. However, the output is different because the post-processing work changed.

```
Total Costs: 20876.4
Total Revenue: 22453.9
Profit: 1577.45
Volume of recipe MakeP1: 25.4839
Volume of recipe MakeP2: 164.516
Volume of recipe MakeP3: 10
Raw materials (orders + market -> production + leftover):
Consumption of raw A: 0 + 13322.6 -> 13322.6 + 0 (total: 13322.6)
Consumption of raw B: 0 + 31000 -> 31000 + 0 (total: 31000)
Consumption of raw C: 0 + 291065 -> 291065 + 0 (total: 291065)
Consumption of raw D: 0 + 200 -> 200 + 0 (total: 200)
Products (production -> orders + market):
Production of product P1: 25.4839 -> 0 + 25.4839 (total: 25.4839)
Production of product P2: 164.516 -> 0 + 164.516 (total: 164.516)
Production of product P3: 10 -> 0 + 10 (total: 10)
```

A second example, still without orders, is Problem 18, where arbitrary recipes were introduced. In this data section we also included the newly introduced parameters and sets, with empty values. This is valid, as the `Orders` set is explicitly set to empty. So that all the other parameters have an empty index set. They need not be present in the data section, but it is not a mistake to make it so. At least it reminds us that the data file may provide additional parameters.

```
data;

set Raws := A B C D;
set Products := P1 P2 P3;
set Recipes := MakeP1 MakeP2 MakeP3 Comp1 Comp2;

param Min_Usage :=
  B  21000
  D  200
  P2  100
  ;

param Max_Usage :=
  A  23000
  B  31000
  C  450000
  D  200
  P3  10
  ;

param Value :=
  A  1
  B  0.07
  C  0.013
  D  8
  P1  252
  P2  89
  P3  139
  ;

param Recipe_Ratio:
            A    B     C  D  P1  P2  P3 :=
  MakeP1  200   25  3200  1   1   0   0
  MakeP2   50  180  1000  1   0   1   0
  MakeP3    0   75  4500  1   0   0   1
  Comp1   240  200  4400  2   1   1   0
  Comp2    51  250  5400  2   0   1   1
  ;

param Initial_Funds :=;

set Orders :=;
param Order_Material_Flow :=;
param Order_Cash_Flow :=;
param Order_Count :=;
param Order_Pay_Before :=;

end;
```

The solution is again the same as before, an optimal profit of 1965.62, with 86.88 units of `P1`, 103.13 units of `P2` and 10 units of `P3` produced, using the joint production options `Comp1` and `Comp2`.

```
Total Costs: 30495
Total Revenue: 32460.6
Profit: 1965.63
Volume of recipe MakeP1: 0
Volume of recipe MakeP2: 6.25
Volume of recipe MakeP3: 0
Volume of recipe Comp1: 86.875
Volume of recipe Comp2: 10
Raw materials (orders + market -> production + leftover):
Consumption of raw A: 0 + 21672.5 -> 21672.5 + 0 (total: 21672.5)
Consumption of raw B: 0 + 21000 -> 21000 + 0 (total: 21000)
Consumption of raw C: 0 + 442500 -> 442500 + 0 (total: 442500)
Consumption of raw D: 0 + 200 -> 200 + 0 (total: 200)
Products (production -> orders + market):
Production of product P1: 86.875 -> 0 + 86.875 (total: 86.875)
Production of product P2: 103.125 -> 0 + 103.125 (total: 103.125)
Production of product P3: 10 -> 0 + 10 (total: 10)
```

And finally let us see a new problem, with orders introduced. The starting point is the previous problem.

Problem 20.

Solve Problem 18, a production problem example with arbitrary recipes, with the following modifications.

- *Maximum usages for raw materials are reset. Instead of* 23000 *units of* `A`*,* 31000 *units of* `B` *and* 450000 *units of* `C` *and* 200 *units of* `D`*, the new max usage limits are* 50000 *units of* `A`*,* 120000 *units of* `B`*,* 1000000 *units of* `C` *and* 1500 *units of* `D`*.*
- *Initial funds are capped at* 35000*.*
- *There are three available orders with the following properties.*

	*Ord*1	*Ord*2	*Ord*3
payment before	*no*	*no*	*yes*
expense	10000	10000	–
income	–	–	500
maximum count	10	10	30
obtain A	20000	15000	190
obtain B	10000	10000	20
obtain C	300000	400000	3000
obtain D	500	500	–
*deliver P*1	40	45	1
*deliver P*2	80	70	–
*deliver P*3	–	6	–

The data section implementation and the solution output are the following.

```
data;

set Raws := A B C D;
set Products := P1 P2 P3;
set Recipes := MakeP1 MakeP2 MakeP3 Comp1 Comp2;

param Min_Usage :=
  B  21000
  D  200
  P2  100
  ;

param Max_Usage :=
  A  50000
  B  120000
  C  1000000
  D  1500
  P3  10
  ;

param Value :=
  A  1
  B  0.07
  C  0.013
  D  8
  P1  252
  P2  89
  P3  139
  ;

param Recipe_Ratio:
            A    B     C  D  P1  P2  P3 :=
  MakeP1  200   25  3200  1   1   0   0
  MakeP2   50  180  1000  1   0   1   0
  MakeP3    0   75  4500  1   0   0   1
  Comp1   240  200  4400  2   1   1   0
  Comp2    51  250  5400  2   0   1   1
  ;

param Initial_Funds := 35000;

set Orders := Ord1 Ord2 Ord3;
param Order_Material_Flow:
            A      B       C    D  P1  P2  P3 :=
  Ord1  20000  10000  300000  500  40  80   0
  Ord2  15000  10000  400000  500  45  70   6
  Ord3    190     20    3000    0   1   0   0
  ;

param Order_Cash_Flow := # negative means income
  Ord1  10000
  Ord2  10000
  Ord3  -500
  ;

param Order_Count :=
  Ord1  10
  Ord2  10
  Ord3  30
  ;

param Order_Pay_Before :=
  Ord1  0
  Ord2  0
  Ord3  1
  ;

end;
```

```
Total Costs: 26497.3
Total Revenue: 47202.7
```

```
Profit: 20705.4
Acquiring order Ord1: 1x
Acquiring order Ord2: 0x
Acquiring order Ord3: 30x
Volume of recipe MakeP1: 0
Volume of recipe MakeP2: 553.716
Volume of recipe MakeP3: 0
Volume of recipe Comp1: 89.1554
Volume of recipe Comp2: 10
Raw materials (orders + market -> production + leftover):
Consumption of raw A: 25700 + 23893.1 -> 49593.1 + 0 (total: 49593.1)
Consumption of raw B: 10600 + 109400 -> 120000 + 0 (total: 120000)
Consumption of raw C: 390000 + 610000 -> 1e+06 + 0 (total: 1e+06)
Consumption of raw D: 500 + 252.027 -> 752.027 + 0 (total: 752.027)
Products (production -> orders + market):
Production of product P1: 89.1554 -> 70 + 19.1554 (total: 89.1554)
Production of product P2: 652.872 -> 80 + 572.872 (total: 652.872)
Production of product P3: 10 -> 0 + 10 (total: 10)
```

As there are much larger maximum usages for raw materials, and production is generally profitable, the extra capacities mean a significant increase in the objective. Also, there are orders used. We can see that the optimal profit is 20705.4, with 89.16 units of `P1`, 553.72 units of `P2`, and 10 units of `P3` produced altogether. We acquire `Ord1` once, and `Ord3` 30 times, which is the maximum available. Actually `Ord3` is a lucrative way to produce and deliver product `P1`. There are significant amounts obtained by orders for each raw material, and there are also purchased from the market. For this reason, there are no leftovers after production. Some of the products are delivered, but in all three products there are also amounts sold on the market.

Let us make an experiment with the integer nature of this model. As mentioned before, this is an MILP model because of the order acquiring decisions, which can only be modeled by integer variables. However, there is a built-in `--nomip` option for `glpsol` to **relax** all integer variables. By relaxing we mean variables are not forced to be integers, just to be between their defined lower and upper bounds, if any. By relaxing all integer variables of an MILP model, we obtain its **LP relaxation**.

```
glpsol -m model.mod -d data.dat --nomip
```

If this option is used, the model is now treated as an LP. This can be solved much faster, but integer variables are allowed to attain fractional values. If this happens, the solution is infeasible in reality. Using the option in the aforementioned problem instance, the following result is reported.

```
Total Costs: 12163
Total Revenue: 33560
Profit: 21397
Acquiring order Ord1: 1x
Acquiring order Ord2: 0x
Acquiring order Ord3: 30x
Volume of recipe MakeP1: 0
Volume of recipe MakeP2: 550
Volume of recipe MakeP3: 0
Volume of recipe Comp1: 90
Volume of recipe Comp2: 10
```

```
Raw materials (orders + market -> production + leftover):
Consumption of raw A: 35700 + 13910 -> 49610 + 0 (total: 49610)
Consumption of raw B: 15600 + 103900 -> 119500 + 0 (total: 119500)
Consumption of raw C: 540000 + 460000 -> 1e+06 + 0 (total: 1e+06)
Consumption of raw D: 750 + 0 -> 750 + 0 (total: 750)
Products (production -> orders + market):
Production of product P1: 90 -> 90 + 0 (total: 90)
Production of product P2: 650 -> 120 + 530 (total: 650)
Production of product P3: 10 -> 0 + 10 (total: 10)
```

What we can see is that the optimal solution of the LP relaxation is 21397, which is better than for the MILP, 20705.4. This is natural, because the LP relaxation has a wider search space, including all the solutions of the MILP and possibly others where integer variables take fractional values. The LP relaxation has a practical importance not in the modeling, but the algorithmic solution point of view. It is usually fast to solve the LP relaxation of an MILP, and if done, its optimal solution is guaranteed to be a bound for the optimal solution of the MILP itself. This is useful information as the MILP itself was hard to impossible to be solved to global optimality. But any times, the actual optimal solution lies between the currently found best solution and a bound. The LP relaxation is a suitable bound.

If we closely investigate the solution for the LP relaxation, it surprisingly seems „more integer" than that of the MILP. However, because the objectives are different, we can be certain that some integer variable has a fractional value. Actually, there are partially acquired orders, `Ord1` is acquired 1.5 times. We can deduce this fact by looking at the material amounts related to orders. The reason it was not reported is simply that order count was printed by the `%d` format specifier, which rounds the fractional value before printing it out as an integer. Therefore the post-processing work is not precise if the integer variables are relaxed. Alternatively `%f` or `%g` could be used.

5.8 Production problem – Summary

We started from the simplest case of production problems: where only available amounts of raw materials are given, and we have to decide production composition to achieve the highest revenue. We demonstrated how this problem may include upper and lower limits on production and consumption, how to optimize by a different objective if needed, and how we can take raw material costs into consideration.

The diet problem is where optimal amount of certain food types shall be determined to satisfy nutrition requirements. This, seemingly quite different problem resulted in a very similar implementation as the production problem. The two can actually be interpreted as special cases of a more general production problem where recipes may consist of multiple raw materials and products at the same time.

Finally, the concept of orders was introduced in the ultimate model of the chapter, which gives it an integer nature, as we can acquire and fulfill orders only fully or not at all, but not partially. This extension gives the raw material supply and final product demand side new possibilities, and our problem is still linear. Additional data files are easily implemented if needed, from the basic to the more complex problems at the same time, and can be solved with the same single model file.

Chapter 6

Transportation problem

Another common optimization problem, the so-called **transportation problem** is presented. Given some supply points with known resource availability, and demand points with known resource requirement, the goal is to arrange transportation of the resource between supplies and demands.

Note that this problem also has very fast algorithmic solution techniques [15]. Again, we are interested in the modeling techniques instead. As we will see, an LP/MILP formulation can easily be adapted if the problem definition changes. This can be substantially more difficult for solution algorithms.

Some additional capabilities of GNU MathProg are presented here. The aim of this chapter is to show how different cost functions can be handled while remaining in the class of LP and sometimes MILP models. Finally, an additional level of transportation is added to the problem definition to show an example of how separately modeled parts of a system can be incorporated into a single model.

6.1 Basic transportation problem

The basic transportation problem, in general, can be described as follows.

Problem 21.

Given a single material, a set of ***supply points****, and a set of* ***demand points****. Each supply point has a nonnegative* ***availability****, and each demand point has a nonnegative* ***requirement*** *for the material. The material can be transported from supply point to any demand point, in any amounts. The unit cost for transportation is known for each such pair. Find the transportation amounts so that the following holds:*

- *Available amounts at supply points are not exceeded.*
- *Required amounts at demand nodes are satisfied.*
- *The total transportation cost is minimal.*

For the sake of simplicity, supply points and demand points are simply called **supplies** and **demands**. Note that the term material can be replaced by any other resource as well (e.g. electricity, water, funds, manpower).

The network of supplies and demands can be described by a directed graph (see Figure 1), where **nodes** represent the supplies and demands, and **arcs** represent connections in between. The direction of an arc represents the direction of material flow, which is always from supply to demand.

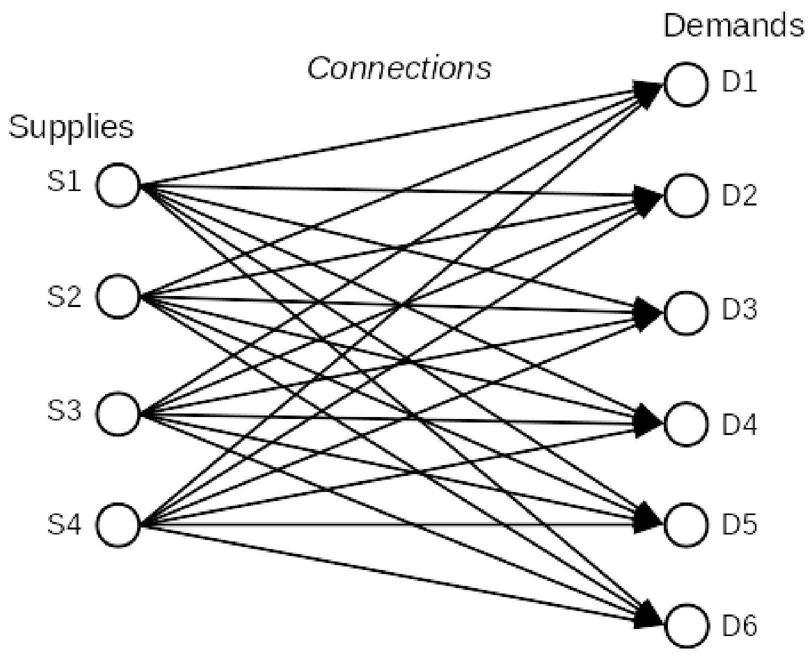

Figure 1: Graph representing the original transportation problem.

As in case of the production problem, we provide an example for the transportation problem. The following problem data will be used throughout this chapter.

Problem 22.
There are four supplies named `S1` *to* `S4` *and six demands named* `D1` *to* `D6`*. The amount of available materials at each supply, required materials at each demand, and unit transportation costs between each pair of a supply and a demand, are all summarized in the following table.*

	D1	*D2*	*D3*	*D4*	*D5*	*D6*	*Available*
S1	5	10	3	9	5	12	100
S2	1	2	6	1	2	6	250
S3	6	5	1	6	4	8	190
S4	9	10	6	8	9	7	210
Required	120	140	170	90	110	120	

(Numbers in the last row and column refer to material amounts, elsewhere numbers refer to costs per amount of material transported.)
Transport materials from the supplies to the demands to satisfy requirements at minimal cost.

As discussed in Section 5.6, the first task when considering mathematical programming is to find suitable decision variables. However, in GNU MathProg, we can start with the implementation of parameters and sets for problem data.
In this problem, there are two sets of arbitrary size each: the set of supplies and the set of demands. There are three kinds of numeric parameters determining the transportation problem, namely the following.

- The availability, provided for each supply.
- The requirement, provided for each demand.
- The unit transportation cost, provided for each pair of a supply and a demand.

Therefore the implementation can be the following. Note that all parameters are indicated as nonnegative to prevent wrong data to be provided. The names of the sets and parameters are self-explanatory.

```
set Supplies;
set Demands;
param Available {s in Supplies}, >=0;
param Required {d in Demands}, >=0;
param Cost {s in Supplies, D in Demands}, >=0;
```

The decisions to be made in a transportation problem are the exact transportation amounts. This amount shall be decided for each pair of a supply and a demand, and these decisions are independent. For example, the problem mentioned has 4 supplies and 6 demands, therefore 24 individual decisions shall be made about transported amounts between supplies and demands. Each of these decisions corresponds to a single variable in the model. With indexing, this can be expressed in a single `var` statement. The name of the variable is `tran`.

```
var tran {s in Supplies, D in Demands}, >=0;
```

Transportation amounts are nonnegative. A negative transportation amount could model a *transportation in backwards direction* (i.e. from a demand to a supply). This may be justified in some practical applications, but definitely prohibited here.

We can see that the transportation amounts describe the situation fully. Based on them, we can easily calculate total activity at each supply and demand node, whether available and required amounts are not violated, and also calculate the costs. Therefore no more decision variables are needed.

Now the constraints in the model are formulated. Of course, not all possible nonnegative real values of the transportation amounts result in a feasible solution in reality, because there are two kinds of restrictions that must be taken into account. First, the total amount transported *from a supply* cannot exceed the availability at that supply. This is a constraint for each supply. Note that the summation goes through all demands, as there can be transportation to all demands from a particular supply.

```
s.t. Availability_at_Supply_Points {s in Supplies}:
  sum {d in Demands} tran[s,d] <= Available[s];
```

Similarly, the total amount transported *to a demand* cannot be less than the requirement at that demand. This constraint is for each demand, and the summation now goes through all supplies, as there can be transportation from all supplies to a particular demand.

```
s.t. Requirement_at_Demand_Points {d in Demands}:
  sum {s in Supplies} tran[s,d] >= Required[d];
```

The objective is the total cost. Each transportation amount must be multiplied by the unit cost on that particular connection, then summed over all connections between supplies and demands.

```
minimize Total_Costs:
    sum {s in Supplies, d in Demands} tran[s,d] * Cost[s,d];
```

At this point, let us think a bit about whether the problem has a solution at all. Two quantities are of special interest: the total amount of available supply, denote it by S, and the total amount of required demand, denote it by D. There are three cases.

- If $S < D$, then the problem is infeasible, as there is simply not enough supply anywhere to fit all demands.
- If $S = D$, then the problem is feasible, but all supply must be used up, and each demand shall get exactly the required amount, no more.
- If $S > D$, then there can (and will) be leftover material at supplies, or exceeding deliveries at demands, or both.

Without solving the model, we can check problem data whether the problem is feasible. This `check` statement is ideally positioned after the parameters but before variable and constraint definitions. This check is useful, because we get a dedicated error message referring to this `check` statement if a problem is infeasible.

```
check sum {s in Supplies} Available[s] >= sum {d in Demands} Required[d];
```

Note that in our model formulation, we allow both leftover materials at supplies, or exceeding deliveries at demands. If these were not allowed, the corresponding two constraints should be equations instead. Note that if transporting costs are positive, then there is no benefit in transporting over the demand, so that case is eliminated by optimization anyways.

In some formulations of the transportation problem, $S = D$ is assumed. Note that in a problem with $S > D$ we can introduce a **dummy demand** with requirement $S - D$, and zero transportation costs from all supplies. The meaning of the dummy demand is that leftover amounts are all transported here. Therefore this new problem is equivalent to the original one, but has equal total supplies and demands. In conclusion, the case $S > D$ is essentially not more general than $S = D$.

The model description for Problem 21, the transportation problem, is ready. The model section can be appended with `printf` statements to show the result once found. Note that we only show transportation amounts that are nonzero. The full model section is the following.

```
set Supplies;
set Demands;
param Available {s in Supplies}, >=0;
param Required {d in Demands}, >=0;
param Cost {s in Supplies, D in Demands}, >=0;

check sum {s in Supplies} Available[s] >= sum {d in Demands} Required[d];

var tran {s in Supplies, D in Demands}, >=0;

s.t. Availability_at_Supply_Points {s in Supplies}:
  sum {d in Demands} tran[s,d] <= Available[s];
s.t. Requirement_at_Demand_Points {d in Demands}:
  sum {s in Supplies} tran[s,d] >= Required[d];

minimize Total_Costs:
    sum {s in Supplies, d in Demands} tran[s,d] * Cost[s,d];

solve;

printf "Optimal cost: %g.\n", Total_Costs;
for {s in Supplies, d in Demands: tran[s,d] > 0}
{
  printf "From %s to %s, transport %g amount for %g (unit cost: %g).\n",
    s, d, tran[s,d], tran[s,d] * Cost[s,d], Cost[s,d];
}

end;
```

For sake of completeness, we also show the data section corresponding to Problem 22.

```
data;

set Supplies := S1 S2 S3 S4;
set Demands := D1 D2 D3 D4 D5 D6;

param Available :=
  S1  100
  S2  250
  S3  190
  S4  210
  ;
param Required :=
  D1  120
  D2  140
  D3  170
  D4  90
  D5  110
  D6  120
  ;
param Cost:
      D1  D2  D3  D4  D5  D6 :=
  S1   5  10   3   9   5  12
  S2   1   2   6   1   2   6
  S3   6   5   1   6   4   8
  S4   9  10   6   8   9   7
  ;

end;
```

Solving the problem with `glpsol` gives the following result.

```
Optimal cost: 2700.
From S1 to D1, transport 10 amount for 50 (unit cost: 5).
From S1 to D5, transport 90 amount for 450 (unit cost: 5).
From S2 to D1, transport 110 amount for 110 (unit cost: 1).
From S2 to D2, transport 140 amount for 280 (unit cost: 2).
From S3 to D3, transport 170 amount for 170 (unit cost: 1).
From S3 to D5, transport 20 amount for 80 (unit cost: 4).
From S4 to D4, transport 90 amount for 720 (unit cost: 8).
From S4 to D6, transport 120 amount for 840 (unit cost: 7).
```

The result of the transportation problem can be fit into the original data table, by substituting the unit costs with the decided transportation amounts. Zero amounts can be omitted at all.

	$D1$	$D2$	$D3$	$D4$	$D5$	$D6$	Available
$S1$	10				90		100
$S2$	110	140					250
$S3$			170		20		190
$S4$				90		120	210
Required	120	140	170	90	110	120	

This representation better explains the elements of the model. Decisions are represented by inner cells, while constraints are represented by the rightmost column and bottom row. Each constraint for a supply expresses that the sum in that row must be at most the number on the right. Each constraint for a demand expresses that the sum in that column is at least the number at the bottom. Since total supplies and demands are equal, these can only be true if all constraints hold as an equation. This is exactly the case throughout the table.

Investigating the results, we can see that the optimal solution tries to use the cheapest unit costs wherever possible, but not always. For example, `S4` to `D4` is chosen, despite not being the cheapest method, neither from supply `S4`, nor to demand `D4`, but proven to be a good choice because the rest of the transportation can be done cheaper. Note that with this single run of the model, we cannot be sure that this is the only optimal solution.

There are plenty of other examples for the transportation problem publicly available, see for example [16].

6.2 Connections as index set

Based on the complete solution presented in Section 6.1, we now enhance the implementation a bit. As mentioned before, in GNU MathProg we can introduce additional parameters and sets in order to simplify model formulation. These can either be defined on spot, read from a separate data section outside the model, or allowing both as a default value.

Observe that the indexing expression `s in Supplies, d in Demands` appears in four different contexts:

- In the `Cost` parameter, as it is defined for all such pairs.
- In the `tran` variable, as it is defined for all such pairs.
- In the objective, as it is a sum over all such pairs.
- In the post-processing work, because transport amounts are printed for all such pairs. Note that in this case, there is a filter that only allows nonzero amounts to be reported.

It would be fatal to make a mistake in this indexing, for example, if we exchanged the order of `Supplies` and `Demands`, resulting in a modeling error. To avoid such errors and decrease redundancy in the model formulation a bit, we can introduce a two-dimensional set of these pairs to be used later. Each supply and each demand are said to be in a **connection**. Therefore, the transportation problem involves the decision of transportation amounts for each connection. The set `Connections` can be introduced in the following way.

```
set Connections := setof {s in Supplies, d in Demands} (s,d);
```

A new GNU MathProg operator `setof` is used here. This is a general way of defining sets based on data previously defined in the model section. The `setof` operator is followed by an indexing expression, then by a simple expression. The resulting set is formed by the expression at the back evaluated for all possible indices in the indexing expression. It is similar to `sum`, but rather to adding elements up, a set of them is formed. Note that the indexing expression in `setof` can be filtered, which gives us a fine control over the set to be defined. As the result is a set, duplicates are removed, and the result can also be empty if everything is filtered out, but these are not the cases here.

In fact, this usage of `setof` is rather simple, just all possible pairs formed by two sets are collected. The set `Connections` is a Cartesian product of sets `Supplies` and `Demands`. There is another built-in operator called `cross` for Cartesian product of two sets, which allows a simpler definition of the set `Connections` as follows. Either definition can be used, they are equivalent.

```
set Connections := Supplies cross Demands;
```

The **dimension** of sets `Supplies` and `Demands` is 1 because they contain simple elements, whereas the dimension of `Connections` is 2, because it contains pairs. There can be as many as 20 dimensions in a set in GNU MathProg. If the set contains n-tuples, then its dimension is n. Sets with different dimensions cannot be mixed together with set operations like `union`, `inter` or `diff`. Even a one-dimensional empty set is considered different from a two-dimensional empty set.

The dimension of a set determines how it can be used in indexing expressions. One-dimensional sets are indexed by a single introduced symbol, like `s in Supplies` or `d in Demands`. However, two- (and more) dimensional sets are indexed by a tuple element, like `(s,d) in Connections`. In general, an indexing expression may contain many sets with different dimensions, each of which introducing one or more new index symbol that can be referred to.

In short, everywhere instead of `s in Supplies, d in Demands` in an indexing expression, we can write `(s,d) in Connections` as follows.

```
set Supplies;
set Demands;
param Available {s in Supplies}, >=0;
param Required {d in Demands}, >=0;
set Connections := Supplies cross Demands;
param Cost {(s,d) in Connections}, >=0;

check sum {s in Supplies} Available[s] >= sum {d in Demands} Required[d];

var tran {(s,d) in Connections}, >=0;

s.t. Availability_at_Supply_Points {s in Supplies}:
  sum {d in Demands} tran[s,d] <= Available[s];

s.t. Requirement_at_Demand_Points {d in Demands}:
  sum {s in Supplies} tran[s,d] >= Required[d];
```

```
minimize Total_Costs:
    sum {(s,d) in Connections} tran[s,d] * Cost[s,d];

solve;

printf "Optimal cost: %g.\n", Total_Costs;
for {(s,d) in Connections: tran[s,d] > 0}
{
  printf "From %s to %s, transport %g amount for %g (unit cost: %g).\n",
    s, d, tran[s,d], tran[s,d] * Cost[s,d], Cost[s,d];
}

end;
```

The resulting new model file is equivalent to the original one, therefore solving the same data file describing Problem 22 shall yield the same result.

To illustrate how it can be useful to introduce an index set like `Connections` explicitly in the model, consider the following new problem.

Problem 23.
Solve Problem 22, the original example transportation problem, with one modification: only those connections are allowed for transportation whose unit costs are not greater than 7*.*

We introduce a parameter to denote the unit cost limit.

```
param Max_Unit_Cost, default 7;
```

Note that by providing a default value of 7 instead of setting the parameter equal to 7, we allow the possibility to alter `Max_Unit_Cost` by providing a value in the data section, if we ever want to choose another limit.

One possible solution is to express a new constraint that explicitly finds each prohibited connection and sets the transported amount there to zero, effectively excluding the connection from the model.

```
s.t. Connections_Prohibited
    {s in Supplies, d in Demands: Cost[s,d] > Max_Unit_Cost}: tran[s,d] = 0;
```

However, in this case we use many variables in the model just to fix them at zero. It is possible not to include those variables in the model formulation at all, and this can be done by introducing a filter at the `setof` expression defining the `Connections` set.

```
set Connections :=
    setof {s in Supplies, d in Demands: Cost[s,d] <= Max_Unit_Cost} (s,d);
```

By this filter, we only include those connections in the `Connections` set that are allowed. Therefore without the modification of other parts of the model, the exclusion is implemented – just the indexing expressions `(s,d) in Connections` iterate over a smaller set in the background.

Note that care must be taken with this method, because now the following constraint would give an **out of domain error**.

```
s.t. Availability_at_Supply_Points {s in Supplies}:
  sum {d in Demands} tran[s,d] <= Available[s];
```

The problem is that `tran[s,d]` is iterated over all pairs of `s in Supplies` and `d in Demands`, but the variable is simply not defined for all such pairs now. This is in strong contrast with the first approach where they are defined, but explicitly set to zero. We must now ensure that the sum only considers allowed connections, and it can be done as follows.

```
s.t. Availability_at_Supply_Points {s in Supplies}:
  sum {(s,d) in Connections} tran[s,d] <= Available[s];
```

In this case, the role of `s` and `d` are different in the `sum` operator. Despite used as an index, `s` is a constant value. But `d` is introduced inside the indexing expression, therefore it can be freely chosen by the `sum`. The meaning of the indexing expression is that all `(s,d)` pairs are selected for which the `s` is a given value. This effectively sums over all demands that are allowed to be connected to the particular `s`, and the constraint works as desired.

The point is that an n-tuple index in an indexing expression *can have constant coordinates* as long as it contains at least one new, free symbol for a coordinate. In the GNU MathProg language documentation, an index symbol which is introduced by an indexing expression is called a **dummy index**. Dummy indices are freely selected by the indexing expression in all possible ways, and can be used as constants afterwards in the expression. Here `s` is a dummy index from the indexing of the constraint, whereas `d` is a dummy index from the indexing expression of the `sum` operator, but both can be referred to in the operand of the sum, which is `tran[s,d]` now.

The model contains another constraint, for the demands, this must also be updated similarly, by replacing `s in Supplies` with `(s,d) in Connections`, and our model section for the new problem is ready.

Solving it with the original problem data reports an optimal solution of 2790, slightly worse than the original solution of 2700. This is not surprising, as the original solution used a unit cost of 8. By excluding it, in theory it is possible to obtain the same objective another way, but it is not the case here. The moral of the story is that contrary to first thought, excluding too expensive connections can be a disadvantage in the transportation problem.

Note that although the transportation problem has feasible (and optimal) solutions if the total supplies are not less than the total demands, this is no longer guaranteed if certain connections are prohibited.

6.3 Increasing unit costs

So far, a linear transportation cost was supposed at each connection. The amount is simply multiplied by a constant to obtain the cost. The relation of the total amount transported and the total cost incurred can be schematically represented as in Figure 2. The red line represents the calculated total costs, but the area above that curve can be regarded as „feasible" too. The logic behind this is that we are allowed to pay more than needed, it just makes no sense.

The term **proportional cost** is also widely used. Proportional cost means such a cost or its component for which the ratio of total amounts and costs is a parameter constant.

In practice, the total cost or effort to be paid for some resource to be used is not always proportional to the amount of resource actually used. A few common examples are shown in this and the following sections, which can be modeled as an LP, or at least as an MILP model.

The first example is when the unit cost is a constant, but after some **threshold** amount is reached, it increases to a higher constant value. This is common in practice and the phenomenon is called the **law of diminishing returns**. This means if we spend an additional unit for costs, we

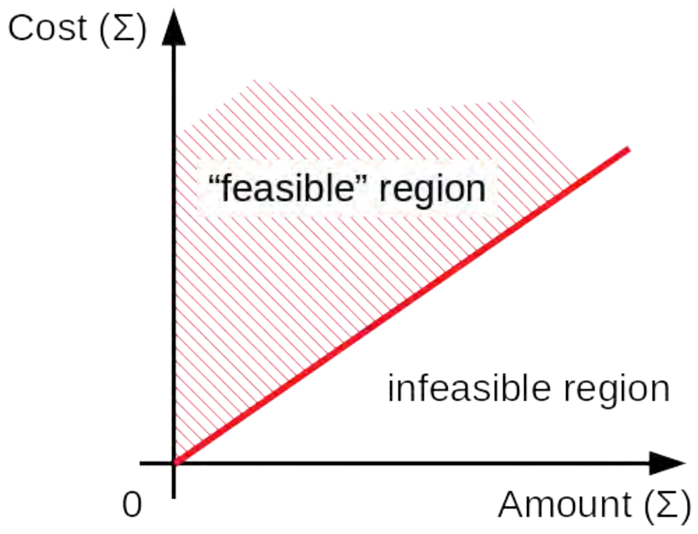

Figure 2: Linear transportation costs.

get less and less return. This is equivalent of having a unit price which increases with total amounts already obtained.

Problem 24.
Solve Problem 21, the transportation problem, with one modification: there are two unit costs for transportation, one below a given threshold amount transported, and a higher unit cost for surplus amounts above that threshold.

The example problem to be solved here is the following.

Problem 25.
Solve Problem 22, the original example transportation problem, with one modification: the given unit costs are only for transportation amounts below 100 *units. Above that limit, costs are increased by* 25% *per material units transported.*

Schematically, the cost function (in contrast with the simple linear case) can be represented as in Figure 3. Again, the region above the red curve denoting the total costs can be termed as feasible, because we can pay more if we want, it is just not advantageous and therefore will never happen.

In the example problem, all thresholds and increased cost are uniform across the connections. Note that this is not necessarily a case. In other problem definitions, each connection may have unique thresholds, basic and increased unit costs.

The first thing to do is to define the data in the model needed to calculate the alternative cost function. Two parameters are introduced: `CostThreshold` is the amount over which the unit costs increase, and `CostIncPercent` denotes the rate of increase, in percent. Note that the increase must be positive.

```
param CostThreshold, >=0;
param CostIncPercent, >0;
```

In the particular example, Problem 25 to be solved, the following values can be given in the data

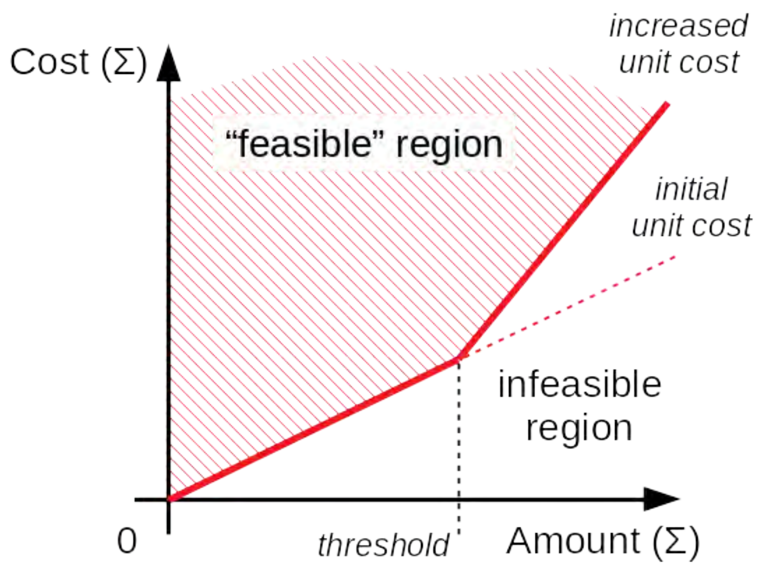

Figure 3: Transportation costs with an increased unit cost above a threshold.

section.

```
param CostThreshold := 100;
param CostIncPercent := 25;
```

We introduce another parameter `CostOverThreshold`, which calculates the increased unit cost by the following formula. Note that the original unit cost is `Cost[s,d]`.

```
param CostOverThreshold {(s,d) in Connections} :=
    Cost[s,d] * (1 + CostIncPercent / 100);
```

At this point, we have all the required data defined in the model. The question is how we could implement the correct calculation of total costs, regardless of being below or above the threshold. First, we can simply introduce two variables, `tranBase` for the amounts below, and `tranOver` for the amounts above the threshold.

```
var tranBase {(s,d) in Connections}, >=0, <=CostThreshold;
var tranOver {(s,d) in Connections}, >=0;
```

Note that both quantities are nonnegative. The amount to be transported over the threshold for the increased cost is unlimited, however, we can only transport a limited amount for the original cost. Therefore, `tranBase` gets an upper bound: the threshold itself. Note that the decision of transported amounts are individual for each connection, therefore all these variables are defined for each connection.

The amounts represented by `tranBase` and `tranOver` are just a separation of the total amount represented by `tran`. Therefore a constraint is provided to ensure that the sum of the former two equals `tran`, for each connection.

```
s.t. Total_Transported {(s,d) in Connections}:
  tran[s,d] = tranBase[s,d] + tranOver[s,d];
```

Instead of using `tran` in the objective, we can refer to the `tranBase` and `tranOver` part separately and multiply these amounts with their corresponding unit costs, which are `Cost` and `CostOverThreshold`.

```
minimize Total_Costs: sum {(s,d) in Connections}
    (tranBase[s,d] * Cost[s,d] + tranOver[s,d] * CostOverThreshold[s,d]);
```

Our model is ready, but think about one thing. Now, the `tranBase` and `tranOver` amounts can be freely set. For example, if the threshold is 100, then transporting 130 units in total can be done as `tranBase=100` and `tranOver=30`, but also for example as `tranBase=50`, `tranOver=80`. In the latter case, the total cost is not calculated correctly in the objective, but it is still allowed by the constraints.

But the model is still working, because the unit cost above the threshold is *strictly higher* than the cost below it. Therefore the optimal solution would not attempt transporting any amounts over the threshold unless all possible amount is transported below it, i.e. `tranBase=100` in the example. Therefore spending more is allowed as feasible solutions in the model, but these cases are eliminated by the optimization procedure. Consequently, in the optimal solution, there is either zero amount above the threshold, or full amount below it, for each connection. The costs are calculated correctly in both scenarios.

The full model section for the transportation problem with increasing rates is shown here.

```
set Supplies;
set Demands;
param Available {s in Supplies}, >=0;
param Required {d in Demands}, >=0;

set Connections := Supplies cross Demands;

param Cost {(s,d) in Connections}, >=0;

param CostThreshold, >=0;
param CostIncPercent, >0;

param CostOverThreshold {(s,d) in Connections} :=
    Cost[s,d] * (1 + CostIncPercent / 100);

check sum {s in Supplies} Available[s] >= sum {d in Demands} Required[d];

var tran {(s,d) in Connections}, >=0;
var tranBase {(s,d) in Connections}, >=0, <=CostThreshold;
var tranOver {(s,d) in Connections}, >=0;

s.t. Availability_at_Supply_Points {s in Supplies}:
  sum {d in Demands} tran[s,d] <= Available[s];

s.t. Requirement_at_Demand_Points {d in Demands}:
  sum {s in Supplies} tran[s,d] >= Required[d];
```

```
s.t. Total_Transported {(s,d) in Connections}:
  tran[s,d] = tranBase[s,d] + tranOver[s,d];

minimize Total_Costs: sum {(s,d) in Connections}
    (tranBase[s,d] * Cost[s,d] + tranOver[s,d] * CostOverThreshold[s,d]);

solve;

printf "Optimal cost: %g.\n", Total_Costs;
for {(s,d) in Connections: tran[s,d] > 0}
{
  printf "From %s to %s, transport %g=%g+%g " &
          "amount for %g (unit cost: %g/%g).\n",
    s, d, tran[s,d], tranBase[s,d], tranOver[s,d],
    (tranBase[s,d] * Cost[s,d] + tranOver[s,d] * CostOverThreshold[s,d]),
    Cost[s,d], CostOverThreshold[s,d];
}
```

Solving the model with a value of 100 for `CostThreshold` and 25 for `CostIncPercent`, we get the following result.

```
Optimal cost: 2772.5.
From S1 to D5, transport 100=100+0 amount for 500 (unit cost: 5/6.25).
From S2 to D1, transport 120=100+20 amount for 125 (unit cost: 1/1.25).
From S2 to D2, transport 130=100+30 amount for 275 (unit cost: 2/2.5).
From S3 to D2, transport 10=10+0 amount for 50 (unit cost: 5/6.25).
From S3 to D3, transport 170=100+70 amount for 187.5 (unit cost: 1/1.25).
From S3 to D5, transport 10=10+0 amount for 40 (unit cost: 4/5).
From S4 to D4, transport 90=90+0 amount for 720 (unit cost: 8/10).
From S4 to D6, transport 120=100+20 amount for 875 (unit cost: 7/8.75).
```

The optimal solution is 2772.5 units, and in some cases the thresholds are surpassed. The objective is slightly worse than the original 2700. This is not surprising, because the current and original problems only differ in the increased costs. If we closely look at the transportation amounts, we can observe that the increased unit costs over the 100 threshold do not only change the pricing, but the optimal transportation decisions as well.

6.4 Economy of scale

The previous Section 6.3 deals with the case of increased unit costs, but what happens when the unit costs do not increase, but decrease above the threshold?

The case when the larger amounts provide lower unit prices is called the **economy of scale**. This is also common in practice. In production environments, if more products are to be processed, the same investment and/or operating costs might share between more products, therefore the cost per product decreases, making production in larger volumes cost-efficient. In trade, it is possible we get a discount if we purchase in a larger volume.

Here, the simplest case of economy of scale is presented, where unit costs simply decrease above a threshold. As usual, we provide a general problem definition and an example problem.

Problem 26.

Solve Problem 21, the transportation problem, with one modification: there are two unit costs for transportation, one below a given threshold amount transported, and a lower unit cost for surplus amounts above that threshold.

Problem 27.

Solve Problem 22, the original example transportation problem, with one modification: the given unit costs are only for transportation amounts below 100 *units. Above that limit, costs are* ***decreased*** *by* 25% *per amount transported.*

Note that the only difference between this problems and the previous one is that the unit costs above the threshold are not higher, but lower. Even the threshold of 100 and the rate of 25% are the same. Therefore it is tempting to address this problem by only slightly modifying our code to calculate with decreasing costs instead. The parameter `CostIncPercent` is renamed to `CostDecPercent`, and the `CostOverThreshold` is calculated accordingly.

```
param CostDecPercent, >0, <=100;
param CostOverThreshold {(s,d) in Connections} :=
    Cost[s,d] * (1 - CostDecPercent / 100);
```

However, if we solve the model, the results do not reflect reality.

```
Optimal cost: 2025.
From S1 to D1, transport 10=0+10 amount for 37.5 (unit cost: 5/3.75).
From S1 to D5, transport 90=0+90 amount for 337.5 (unit cost: 5/3.75).
From S2 to D1, transport 110=0+110 amount for 82.5 (unit cost: 1/0.75).
From S2 to D2, transport 140=0+140 amount for 210 (unit cost: 2/1.5).
From S3 to D3, transport 170=0+170 amount for 127.5 (unit cost: 1/0.75).
From S3 to D5, transport 20=0+20 amount for 60 (unit cost: 4/3).
From S4 to D4, transport 90=0+90 amount for 540 (unit cost: 8/6).
From S4 to D6, transport 120=0+120 amount for 630 (unit cost: 7/5.25).
```

The optimal solution is 2025, way better than the original 2700. There is nothing wrong so far, because costs decreased. But the details show that there is zero transportation amounts below the threshold, the total amount is transported for the lower price, for amounts above the threshold. This is clearly not allowed, as we can only use the lower unit costs if we filled the thresholds first, after spending the higher unit cost for the amounts below.

The case where the cost above the threshold was higher, the model worked perfectly. Now, when the cost above the threshold is lower, the model is wrong. What makes this difference?

- If the unit costs *increase* above the threshold, the amounts above the threshold are filled first, and then we can optionally transport additional amounts above the threshold. We (and the model) are allowed to transport above the threshold first, but it does not give any benefit, therefore these approaches are ruled out by the optimization procedure.

- If the unit costs *decrease* above the threshold, however, the optimization prefers using the amounts above the threshold because of the lower cost. However, it does not work like a „free option”, but a „right” that must be obtained: we can only use the lower unit cost if we filled the amounts below the threshold first. This is not enforced by the constraints, and therefore the reported optimal solution is infeasible in reality.

To better understand the difference, observe the slightly modified schematic representation of economy of scale (see Figure 4). One might argue that the only difference is that the curve is going downward, not upward, but observe the feasible region.

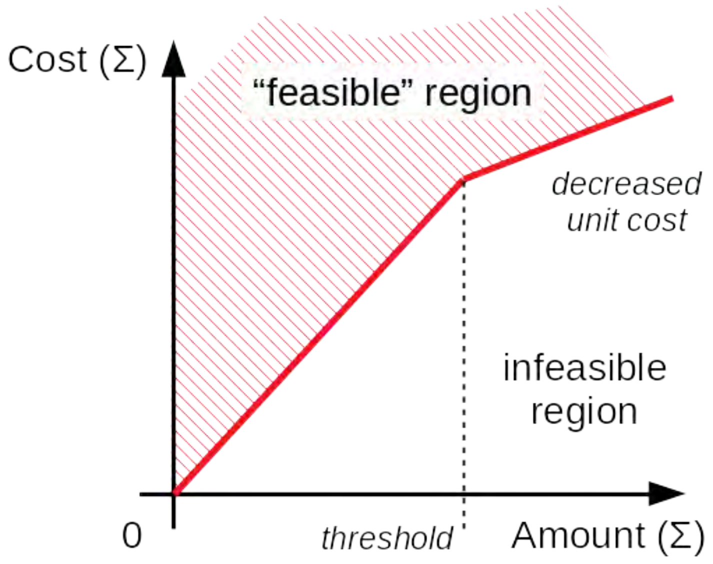

Figure 4: Transportation costs with an increased unit cost above a threshold.

One property of LP problems is that **their search space is always convex**. This means, if we have two feasible solutions, and make a **convex sum** of these, then the result is also a feasible solution. For example, if $x_1 = 3$, $y_1 = 1$ is one feasible solution, and $x_2 = 7$, $y_2 = 9$ is another, then let us obtain a convex sum of these with weights $\frac{3}{4}$ and $\frac{1}{4}$. We get $x_3 = \frac{3}{4}x_1 + \frac{1}{4}x_2 = 4$, $y_3 = \frac{3}{4}y_1 + \frac{1}{4}y_2 = 4$, which is guaranteed to be also feasible. If we investigate what general constraints are allowed in LP models, the convexity of the feasible region could be verified easily.

However, if we look at the schematic representation in Figure 4, the feasible region is not convex. Moreover, we cannot make it convex by cutting down parts from it. The reason is that we should allow feasibility on both of the two red line segments, but their convex sums fall into the infeasible region that must be prohibited by model constraints. We can see why this was not a problem in case if increasing unit costs: because the convex sums of the two red line segments fell into the feasible region – those cases did not make sense in reality though, but the model still included them, and overall the problem could be perfectly modeled by LP.

But the feasible region is clearly not convex, which suggests that **economy of scale cannot be modeled by LP**. However, we can model it by adding integer variables, making the model an MILP. Our goal is to ensure that the amounts below the threshold, denoted by `tranBase` are utilized in full whenever any amounts are transported above the threshold, denoted by `tranOver`.

For each connection, we introduce a binary variable `isOver`.

```
var isOver {(s,d) in Connections}, binary;
```

The meaning of `isOver` is that whether we are entitled ($= 1$) or not ($= 0$) to transport for the lower unit cost, for amounts above the threshold. This binary variable introduces a discrete nature to the problem, and now the search space is not convex, because no fractional values are allowed for `isOver` between 0 and 1. Depending on the value `isOver` takes, there are specific circumstances required.

- If `isOver=0`, then we cannot use the lower cost, hence the amounts above the threshold must be zero.
- If `isOver=1`, then we must get the right to use the lower costs, hence the amounts below the threshold must be maximal, i.e. equal to the threshold amount.

It is a common problem in mathematical programming that we have an ordinary linear constraint, like $A \geq B$, but we want this constraint to be active if and only if a certain condition is met. The condition is often represented by the value of a binary variable x. If $x = 1$, then constraint $A \geq B$ must hold. However, if $x = 0$, there is no restrictions about A or B.

The modeling technique used is called **big-M constraint**. The first step is to arrange the linear constraint into the form $B - A \leq 0$, then find a positive **upper bound for** $B - A$. This upper bound shall be large enough to include all possible values for all model variables appearing in the expression $B - A$. In practice, often a very large number, like $M = 1000000$ is selected as an upper bound, which is magnitudes larger than any sensible solution for the problem, is sufficient. This is the so-called **big-M** value.

Then, include the following constraint in the model formulation.

$$B - A \leq M \cdot (1 - x) \tag{12}$$

Now investigate what this constraint does in the model. Because x is an integer variable, there are two cases. If $x = 1$, then the constraint reduces to $B - A \leq 0$, which is exactly what we wanted to achieve if $x = 1$. If $x = 0$ however, the constraint reduces to $B - A \leq M$. Remember that M is magnitudes larger than any possible, sensible values of $B - A$. Therefore the constraint becomes *redundant*, as it does not impose further restrictions on model variables – again, what we exactly wanted for $x = 0$.

If the linear constraint is an equation, like $A = B$, then we can express it as two inequalities $A \geq B$ and $A \leq B$ and implement a big-M constraint for each.

If the condition is not $x = 1$ but $x = 0$ instead, then we can simply replace $(1 - x)$ by x in the model formulations as the negation of any binary value x is $(1 - x)$.

We can see that this is a very general technique which allows conditional inclusion of linear constraints in MILP models, provided that their condition can be expressed as a binary variable (or any other linear expression which results in a value that can only be 0 or 1, but nothing in between).

One might ask: if M is only required to be large enough, what is the most suitable M to be given? In the solution algorithmic point of view, the best choice is usually the **lowest possible** M. Be careful though, if M is too small, the constraint is not redundant when the condition is not met, and we risk excluding valuable solutions. (It might be OK to exclude feasible solutions, and even optimal ones, provided that at least one optimal solution remains.) Too large M values may also result in numerical errors in the solution procedure.

Now that the general technique of big-M constraints is introduced, let us head back to the economy of scale and implement the required constraints. There are two conditional constraints to implement.

- If `isOver=0`, then `tranOver=0` must hold.
- If `isOver=1`, then `tranBase=CostThreshold` must hold.

The first constraint can be formulated as follows.

```
param M := sum {s in Supplies} Available[s];
s.t. Zero_Over_Threshold_if_Threshold_Not_Chosen {(s,d) in Connections}:
  tranOver[s,d] <= M * isOver[s,d];
```

Note that when isOver[s,d]=0, then tranOver[s,d]<=0 is enforced, and because it is a non-negative variable, it will be zero. The value M must be selected to be larger than any sensible values of tranOver[s,d]. One easy possibility is to add up all total supplies in the problem. Clearly, no single transportation amount can be larger than this M, we introduced this value as a parameter. We could provide smaller values for each tranOver[s,d] if we wanted, but a single M will suffice.

The second constraint can be formulated as follows.

```
s.t. Full_Below_Threshold_if_Threshold_Chosen {(s,d) in Connections}:
    tranBase[s,d] >= CostThreshold - M * (1 - isOver[s,d]);
```

If isOver[s,d]=1, then tranBase[s,d]>=CostThreshold is enforced, and because parameter CostThreshold is also an upper bound for tranBase[s,d], they will be equal. If isOver[s,d]=0, then the constraint is redundant, because the right-hand side, CostThreshold-M is a negative number and tranBase[s,d] is a nonnegative variable. However, we can provide a more clever big-M for this constraint. The only requirement is that CostThreshold-M cannot be positive, otherwise tranBase[s,d]=0 would be excluded from the search space entirely, which is unacceptable. But the big-M in this constraint can be set as low as CostThreshold, in which case CostThreshold-M=0 and the constraint is still redundant when isOver[s,d]=1. However, for this particular big-M, the whole constraint can be rearranged into a more readable form, as follows.

```
s.t. Full_Below_Threshold_if_Threshold_Chosen {(s,d) in Connections}:
    tranBase[s,d] >= CostThreshold * isOver[s,d];
```

We can see that if isOver[s,d]=1, then tranBase[s,d]>=CostThreshold is enforced, but if isOver[s,d]=0, then the constraint is equivalent to tranBase[s,d]>=0, which is redundant.

In conclusion, we added the variable isOver and these two constraints to the previous model for increasing cost rates, and now our model for transportation problem with economy of scale is ready.

```
set Supplies;
set Demands;

param Available {s in Supplies}, >=0;
param Required {d in Demands}, >=0;

set Connections := Supplies cross Demands;

param Cost {(s,d) in Connections}, >=0;

param CostThreshold, >=0;
param CostDecPercent, >0, <=100;

param CostOverThreshold {(s,d) in Connections} :=
    Cost[s,d] * (1 - CostDecPercent / 100);

check sum {s in Supplies} Available[s] >= sum {d in Demands} Required[d];

var tran {(s,d) in Connections}, >=0;
var tranBase {(s,d) in Connections}, >=0, <=CostThreshold;
var tranOver {(s,d) in Connections}, >=0;
var isOver {(s,d) in Connections}, binary;
```

```
s.t. Availability_at_Supply_Points {s in Supplies}:
  sum {d in Demands} tran[s,d] <= Available[s];

s.t. Requirement_at_Demand_Points {d in Demands}:
  sum {s in Supplies} tran[s,d] >= Required[d];

s.t. Total_Transported {(s,d) in Connections}:
  tran[s,d] = tranBase[s,d] + tranOver[s,d];

param M := sum {s in Supplies} Available[s];

s.t. Zero_Over_Threshold_if_Threshold_Not_Chosen {(s,d) in Connections}:
  tranOver[s,d] <= M * isOver[s,d];

s.t. Full_Below_Threshold_if_Threshold_Chosen {(s,d) in Connections}:
#  tranBase[s,d] >= CostThreshold - M * (1 - isOver[s,d]);
  tranBase[s,d] >= CostThreshold * isOver[s,d];

minimize Total_Costs: sum {(s,d) in Connections}
    (tranBase[s,d] * Cost[s,d] + tranOver[s,d] * CostOverThreshold[s,d]);

solve;

printf "Optimal cost: %g.\n", Total_Costs;
for {(s,d) in Connections: tran[s,d] > 0}
{
  printf "From %s to %s, transport %g=%g+%g " &
          "amount for %g (unit cost: %g/%g).\n",
    s, d, tran[s,d], tranBase[s,d], tranOver[s,d],
    (tranBase[s,d] * Cost[s,d] + tranOver[s,d] * CostOverThreshold[s,d]),
    Cost[s,d], CostOverThreshold[s,d];
}
```

We get the following results for Problem 27.

```
Optimal cost: 2625.
From S1 to D1, transport 10=10+0 amount for 50 (unit cost: 5/3.75).
From S1 to D5, transport 90=90+0 amount for 450 (unit cost: 5/3.75).
From S2 to D1, transport 110=100+10 amount for 107.5 (unit cost: 1/0.75).
From S2 to D2, transport 140=100+40 amount for 260 (unit cost: 2/1.5).
From S3 to D3, transport 170=100+70 amount for 152.5 (unit cost: 1/0.75).
From S3 to D5, transport 20=20+0 amount for 80 (unit cost: 4/3).
From S4 to D4, transport 90=90+0 amount for 720 (unit cost: 8/6).
From S4 to D6, transport 120=100+20 amount for 805 (unit cost: 7/5.25).
```

Now the optimal solution is 2625, still better than the original 2700, but not significantly. We can observe that each time a lower cost is used, the threshold 100 is first reached. Unlike the case for the increasing unit costs, in Section 6.3, the reduced costs above the threshold now only affect the objective through pricing, but the optimal solution remains the same in terms of transported amounts.

Remember that the model contains binary variables, making it an MILP, instead of a pure LP model. If the number of binary variables in an MILP is large, computational time required for the final solution will eventually explode. This is not the case here only because the resulting MILP model with its 24 binary variables is still considered small.

6.5 Fixed costs

Different scenarios for unit costs and changes were considered. These kinds of costs were all **proportional costs**, because the calculation formula was always based on the amount in question, which was multiplied by some constant. In some cases, not only proportional costs, but **fixed costs** may be present.

Problem 28.
Solve Problem 21, the transportation problem, with the following modification: for each connection, there is a fixed cost to be paid once for establishing it, in order to use that connection to transport materials.

The term **fixed cost** means that its value does not depend on the actual amounts in question. We only have to pay the fixed cost once in order to utilize a feature. In this example, the feature is transportation through each connection. We may choose not to establish a connection, but then we cannot transport between that particular supply and demand.

The example problem to be solved here is the following.

Problem 29.
Solve Problem 28, the transportation problem with fixed costs, with data from the original example Problem 22, and the following fixed establishment costs per connection.

	D1	*D2*	*D3*	*D4*	*D5*	*D6*
S1	50	50	50	50	200	50
S2	200	50	50	50	200	50
S3	50	200	200	50	50	50
S4	50	50	50	200	200	200

The starting point for implementing our model is the general implementation from Section 6.2 where all connections were present. We add a new parameter `FixedCost` for the establishment costs per connection. The matrix in Problem 29 can then be implemented in the data section to provide values for `FixedCost`.

```
param FixedCost {(s,d) in Connections}, >=0;
```

```
param FixedCost:
       D1   D2   D3   D4   D5   D6 :=
  S1   50   50   80   50  200   50
  S2  200   50   50   50  200   50
  S3   50  200  200   50   50   50
  S4   50   50   50  200  200  200
  ;
```

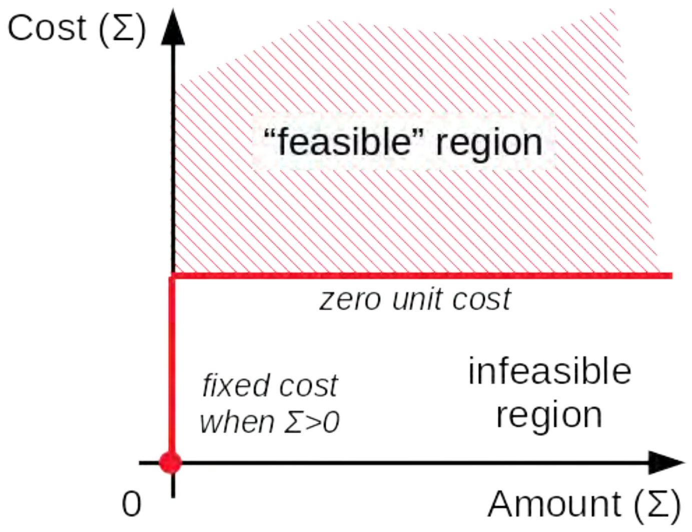

Figure 5: Fixed establishment cost when any positive amount of transportation is utilized.

If we observe the schematic representation of the fixed cost (see Figure 5), then we can see that the curve is again not convex, as in the case of economy of scale. The non-convex nature comes from the single point where the total amount *and* the total cost are zero, a single point which cannot be directly connected to the horizontal line describing the fixed cost for any amounts above zero, without intersecting the infeasible region.

As in case of the economy of scale, we need a binary decision variable to distinguish the two cases. Here, `tranUsed` equals one if we want to establish the connection and be able to send any positive amount of materials over it. If `tranUsed` is zero, then transportation on that particular connection must also be zero.

```
var tranUsed {(s,d) in Connections}, binary;
```

Two things must be enforced about the fixed cost and the `tranUsed` variable. The first is that if we do not establish the connection, then there must be no amount transported there. This can be done with a single big-M constraint.

```
param M := sum {s in Supplies} Available[s];

s.t. Zero_Transport_If_Fix_Cost_Not_Paid {(s,d) in Connections}:
  tran[s,d] <= M * tranUsed[s,d];
```

The coefficient `M` must be large in order not to exclude legitimate solutions from the search space. Precisely, it must be at least the largest possible value of `tran[s,d]` in the optimal solution to be found. The total of all available supplies, denoted by `M` is a single valid candidate for all big-M constraints, for any supply and demand. However, if we want stricter constraints that might result in better solver running times, we can provide a particular value for each supply and demand, which is the minimum of the available amount at the supply and the required amount at the demand. There cannot be more traffic between the two points.

```
s.t. Zero_Transport_If_Fix_Cost_Not_Paid {(s,d) in Connections}:
  tran[s,d] <= min(Available[s],Required[d]) * tranUsed[s,d];
```

The second thing to implement is the inclusion of the fixed establishment cost into the objective.

```
minimize Total_Costs: sum {(s,d) in Connections}
    (tran[s,d] * Cost[s,d] + tranUsed[s,d] * FixedCost[s,d]);
```

Our implementation is ready. Variable `tranUsed[s,d]` is either zero, in which case there is no fixed cost but also no transportation over that connection, or `tranUsed[s,d]` is one, the cost is paid and there can be a positive transportation amount over that connection. Simultaneously zero transportation while still establishing the connection is allowed as a feasible solution in the model, but optimization rules these cases out. The full model section is the following.

```
set Supplies;
set Demands;

param Available {s in Supplies}, >=0;
param Required {d in Demands}, >=0;

set Connections := Supplies cross Demands;

param Cost {(s,d) in Connections}, >=0;
param FixedCost {(s,d) in Connections}, >=0;

check sum {s in Supplies} Available[s] >= sum {d in Demands} Required[d];

var tran {(s,d) in Connections}, >=0;
var tranUsed {(s,d) in Connections}, binary;

s.t. Availability_at_Supply_Points {s in Supplies}:
  sum {d in Demands} tran[s,d] <= Available[s];

s.t. Requirement_at_Demand_Points {d in Demands}:
  sum {s in Supplies} tran[s,d] >= Required[d];

s.t. Zero_Transport_If_Fix_Cost_Not_Paid {(s,d) in Connections}:
  tran[s,d] <= min(Available[s],Required[d]) * tranUsed[s,d];

minimize Total_Costs: sum {(s,d) in Connections}
    (tran[s,d] * Cost[s,d] + tranUsed[s,d] * FixedCost[s,d]);

solve;

printf "Optimal cost: %g.\n", Total_Costs;
for {(s,d) in Connections: tran[s,d] > 0}
{
  printf "From %s to %s, transport %g " &
         "amount for %g (unit cost: %g, fixed: %g).\n",
    s, d, tran[s,d],
    tran[s,d] * Cost[s,d] + FixedCost[s,d], Cost[s,d], FixedCost[s,d];
}

end;
```

The original Problem 22 with zero fixed costs and the new Problem 29 mentioned here are both solved by the model. The original optimal solution is 2700, and is reproduced for the model with fixed costs if these fixed costs are simply zero. If nonzero fixed costs are present, the following output is reported.

```
Optimal cost: 3640.
From S1 to D1, transport 100 amount for 550 (unit cost: 5, fixed: 50).
From S2 to D1, transport 20 amount for 220 (unit cost: 1, fixed: 200).
From S2 to D2, transport 140 amount for 330 (unit cost: 2, fixed: 50).
From S2 to D4, transport 90 amount for 140 (unit cost: 1, fixed: 50).
From S3 to D3, transport 80 amount for 280 (unit cost: 1, fixed: 200).
From S3 to D5, transport 110 amount for 490 (unit cost: 4, fixed: 50).
From S4 to D3, transport 90 amount for 590 (unit cost: 6, fixed: 50).
From S4 to D6, transport 120 amount for 1040 (unit cost: 7, fixed: 200).
```

The objective is 3640. The connections used are not the same in the two cases. The establishment cost paid is in total is 850, note that it is smaller than the increment of 940 of the objective. This is natural, because not only the extra cost must be paid, but the new transportation solution obtained is not necessarily optimal according to the original objective anymore.

6.6 Flexible demands

Several transportation cost functions were introduced, including increasing, decreasing proportional and fixed costs. One additional example is mentioned now, but in a slightly different context.

The common property of transportation costs was that total cost increased (or remained constant) as the total transported amount increased. But what happens when the curve is not monotonically increasing? As this is not so practical for transportation costs, we rather demonstrate it throughout the demands. So far, the required amount at each demand point was a **strict constraint**. It means that those solutions for the problem in which demands are not met are infeasible and eliminated from the search. Any constraints so far were strict.

In some cases constraints are **soft constraints** instead. It means that we are interested in solutions which violate them. In that case, the extent of violation shall be minimized, which is another objective of the optimization procedure. Considering multiple objectives is under the field of **goal programming** [17], and is not covered here. However, there are techniques to merge different objective functions into a single one, resulting in an LP/MILP model again. One technique is the usage of **penalties**, which are terms added to the objective function and are proportional to the extent a constraint is violated. Penalties are practical: failing to comprehend to a desired goal can sometimes be measured or estimated as costs which can be added to the objective if it is profit maximization or cost minimization.

In this section, the starting point is the original transportation problem again, and is expanded with **flexible demands**.

Problem 30.

Solve Problem 21, the transportation problem with the addition of flexible demands. For each demand point, we do not require the exact requirement to be served. Instead, a penalty is introduced *into the objective which is proportional to the extent of difference between required amount at a demand point and actually served amount.*

Two example problems will be solved, which differ in the parameter constants only.

Problem 31.
Solve Problem 22, the originally proposed transportation problem, with flexible demands added. There is a surplus and a shortage penalty constant. The penalty for each demand is calculated by multiplying the difference of actual transported materials and the requirement, by a corresponding penalty constant.
There are two different scenarios to be evaluated.

- *The shortage penalty constant is* 3*, and the surplus penalty constant is* 1*. This is the „small" penalty scenario.*
- *The shortage penalty constant is* 15*, and the surplus penalty constant is* 10*. This is the „large" penalty scenario.*

The starting point for implementation is the model with connections, from Section 6.2. Data content is expanded by two parameters, let us call them `ShortagePenalty` and `SurplusPenalty`. These can be given values in the data section according to the scenarios in Problem 31.

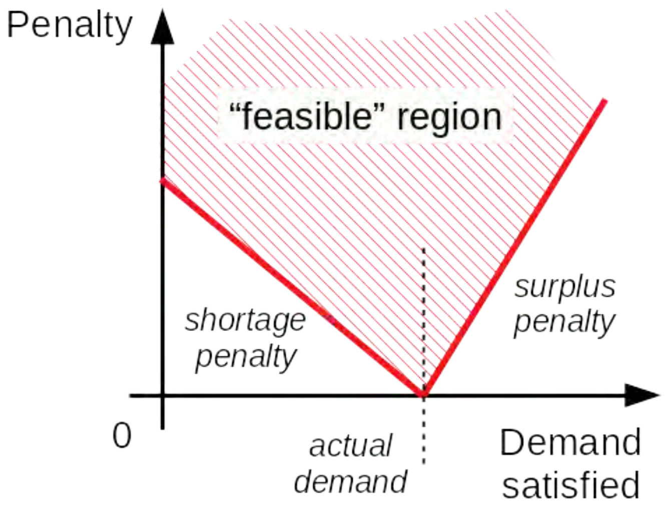

Figure 6: Penalty as a function of total delivered amount. The minimum point zero, and the requirement is exactly met there.

Let us investigate penalties as a function of total materials delivered at a given demand point (see Figure 6). Linear penalty functions are assumed in both directions. We can observe that this penalty can be considered as a special cost function, which decreases below and increases above the actual requirement for the demand point. If the requirement is exactly met, the penalty is zero. Although the function curve is „broken" in a manner similar to the absolute value function, the feasible region is still convex, which suggests a pure LP model.

The model is extended with an auxiliary variable `satisfied`. This is defined for each demand, and denotes the total amount of materials delivered to that demand. This helps in the formulation as penalty for a demand `d` is based on the difference between `satisfied[d]` and `Required[d]`.

```
var satisfied {d in Demands}, >=0;

s.t. Calculating_Demand_Satisfied {d in Demands}:
  satisfied[d] = sum {s in Supplies} tran[s,d];
```

From this point, two different methods are shown, which are applicable to penalties in general and are considered equally effective.

1. The first method introduces a single variable for the penalty itself.
2. The second method introduces two variables for the shortage and the surplus of materials.

The first method relies on the idea that the feasible region of the problem is determined by two lines: one for penalty below, and one for penalty above the requirement. This means that two linear constraints can achieve what we want.

```
var penalty {d in Demands}, >=0;

s.t. Shortage_Penalty_Constraint {d in Demands}:
  penalty[d] >= ShortagePenalty * (Required[d] - satisfied[d]);

s.t. Surplus_Penalty_Constraint {d in Demands}:
  penalty[d] >= SurplusPenalty * (satisfied[d] - Required[d]);
```

The first constraint ensures that the penalty is at least as defined for shortage. Note that this constraint is redundant if the demand is satisfied, as the RHS becomes negative.

The second constraint ensures that the penalty is at least as defined for surplus. Similarly, this constraint becomes redundant if the demand is not met, as the RHS becomes negative.

The `penalty` variable can be directly included into the objective function, and the model formulation for the first method is ready. Note that the model allows penalty values higher than described as feasible solutions, but these are eliminated by the optimization procedure. That means, in the optimal solution, one of the constraints will be strict, or both if the requirement is exactly met and the penalty is zero.

```
minimize Total_Costs:
  sum {(s,d) in Connections} tran[s,d] * Cost[s,d] +
  sum {d in Demands} penalty[d];
```

Let us see the second method for implementing flexible demands. This alternative way uses more variables, but less constraints. The exact shortage and surplus amounts are introduced as variables in this formulation.

```
var surplus {d in Demands}, >=0;
var shortage {d in Demands}, >=0;

s.t. Calculating_Exact_Demands {d in Demands}:
  Required[d] - shortage[d] + surplus[d] = satisfied[d];
```

Note that the exactly transported amount, termed as `satisfied[d]` is now the required amount, minus the shortage, plus the surplus – provided that either variable is zero, as common sense dictates. This is expressed by the constraint shown.

However, the model allows both the shortage and the surplus value to be increased by the same amount, and everything else remains the same. But these cases will be eliminated by the optimization procedure, as the new variables are introduced into the objective with their corresponding penalty constants as factors. This completes the implementation of the second method. Note that the optimization procedure ensures that either the shortage or the surplus is zero.

```
minimize Total_Costs:
  sum {(s,d) in Connections} tran[s,d] * Cost[s,d] +
  sum {d in Demands} (shortage[d] * ShortagePenalty +
      surplus[d] * SurplusPenalty);
```

The full model section for the first method is presented below. Remember that the two methods are equivalent.

```
set Supplies;
set Demands;

param Available {s in Supplies}, >=0;
param Required {d in Demands}, >=0;

set Connections := Supplies cross Demands;

param Cost {(s,d) in Connections}, >=0;

check sum {s in Supplies} Available[s] >= sum {d in Demands} Required[d];

param ShortagePenalty, >=0;
param SurplusPenalty, >=0;

var tran {(s,d) in Connections}, >=0;
var satisfied {d in Demands}, >=0;
var penalty {d in Demands}, >=0;

s.t. Calculating_Demand_Satisfied {d in Demands}:
  satisfied[d] = sum {s in Supplies} tran[s,d];

s.t. Availability_at_Supply_Points {s in Supplies}:
  sum {d in Demands} tran[s,d] <= Available[s];

s.t. Shortage_Penalty_Constraint {d in Demands}:
  penalty[d] >= ShortagePenalty * (Required[d] - satisfied[d]);

s.t. Surplus_Penalty_Constraint {d in Demands}:
  penalty[d] >= SurplusPenalty * (satisfied[d] - Required[d]);

minimize Total_Costs:
  sum {(s,d) in Connections} tran[s,d] * Cost[s,d] +
  sum {d in Demands} penalty[d];

solve;
```

```
printf "Optimal cost: %g.\n", Total_Costs;
for {d in Demands}
{
  printf "Required: %s, Satisfied: %g (%+g)\n",
    d, satisfied[d], satisfied[d]-Required[d];
}
for {(s,d) in Connections: tran[s,d] > 0}
{
  printf "From %s to %s, transport %g amount for %g (unit cost: %g).\n",
    s, d, tran[s,d], tran[s,d] * Cost[s,d], Cost[s,d];
}

end;
```

Regardless of which method we choose, the same result is obtained. For the „small" penalty scenario, we can see that the total costs are cut by almost a half.

```
Optimal cost: 1450.
Required: D1, Satisfied: 120 (+0)
Required: D2, Satisfied: 40 (-100)
Required: D3, Satisfied: 170 (+0)
Required: D4, Satisfied: 90 (+0)
Required: D5, Satisfied: 0 (-110)
Required: D6, Satisfied: 0 (-120)
From S2 to D1, transport 120 amount for 120 (unit cost: 1).
From S2 to D2, transport 40 amount for 80 (unit cost: 2).
From S2 to D4, transport 90 amount for 90 (unit cost: 1).
From S3 to D3, transport 170 amount for 170 (unit cost: 1).
```

There are no surplus penalties, but vast shortages for D2, D5 and D6. The latter two demand sites do not receive any materials at all. The reason is that the unit cost of transportation exceeds the gain by satisfying the demand. In case of D2, the first 40 units can be transported for low costs, but the remaining part of the requirements cannot. The reason for not satisfying demands is that the shortage penalty constant 3 is very low, so it is worth not delivering any materials at all.

On the other hand, the „large" penalty scenario results in exactly the same solution as the original transportation problem without the penalties. There is no surplus again, but no shortages either, simply because the shortage penalty of 15 per unit material is more costly than transporting the material from any available supply. Therefore there are no shortages either.

There were no surplus deliveries in any case, which is not surprising. There had been no point in surplus delivery to any demand point in the original problem as well, because it just increases the transportation costs. The surplus penalty just makes surplus even less beneficial. The case would be different if surplus deliveries were somehow needed – for example, if available supplies were required to be transported completely.

6.7 Adding a middle level

In the former sections, different modeling techniques for cost functions and penalties were presented. Now, the transportation problem is expanded to a larger scale.

So far, the graph had been very simple: there were two sets of nodes in the network, the supplies and demands, and these were all connected. But what happens, when the network is more complex, for example, if there is a third set of nodes?

Problem 32.
Solve Problem 21, the transportation problem extended with a middle level of so-called ***centers*** *introduced.*

Material does not go directly from the supply to the demand. Instead, material goes from supply nodes to center nodes first, and then from center nodes to demand nodes. Connections between either supplies and centers, or centers and demands, have their own transportation costs as usual.

Moreover, centers are not necessarily used in transportation, but if are, an ***establishment cost*** *must be paid first if any transportation is planned there. Establishment costs are one-time, and given for each center.*

As already mentioned, the original transportation problem can be represented by a graph with two types of nodes: supplies and demands (see Figure 1 in Section 6.1). In contrast, the scheme of the new transportation problem can be seen in Figure 7. Connections from supplies to centers are termed as **A-type**, while connections from centers to demands are termed as **B-type**, as the Figure shows.

Note that in general, there could be any network for transportation. However, if cycles are present, care must be taken to correctly interpret material flow through these cycles. Throughout this chapter, there are no cycles, material only flows in one direction.

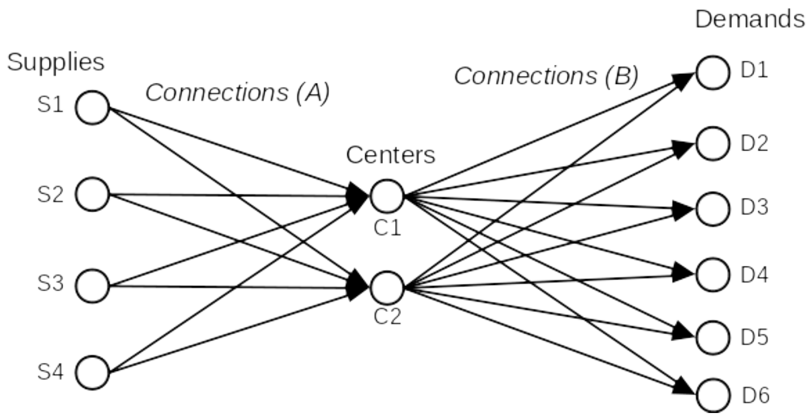

Figure 7: Graph representing the transportation problem with center nodes included.

Problem 33.
Solve Problem 21, the general transportation problem with centers, with the following problem data. There are four supplies, named `S1` *to* `S4`*, two centers,* `C1` *and* `C2`*, and six demands, named* `D1` *to* `D6`*. Transportation costs, available amounts for supplies, and required amounts for demands are summarized in the following table.*

	S1	*S2*	*S3*	*S4*	*D1*	*D2*	*D3*	*D4*	*D5*	*D6*
C1	5	10	3	10	10	3	9	8	1	6
C2	10	2	7	4	2	7	3	2	7	8
Amounts	100	250	190	210	120	140	170	90	110	120

Two scenarios must be evaluated.

- *In the first, „small" establishment cost scenario, using* `C1` *and* `C2` *require a fixed cost of* 1200 *and* 1400*, respectively.*
- *In the second, „large" establishment cost scenario, fixed costs are multiplied by* 10*, e.g.* 12000 *and* 14000*, respectively.*

Before model implementation starts, the data section describing Problem 33 can be implemented as follows. The only difference of the two scenarios is in the value of the `EstablishCost` parameter.

```
data;

set Supplies := S1 S2 S3 S4;
set Centers := C1 C2;
set Demands := D1 D2 D3 D4 D5 D6;

param Available :=
  S1  100
  S2  250
  S3  190
  S4  210
  ;

param Required :=
  D1  120
  D2  140
  D3  170
  D4  90
  D5  110
  D6  120
  ;

param CostA:
      C1  C2 :=
  S1   5  10
  S2  10   2
  S3   3   7
  S4  10   4
  ;

param CostB:
      D1  D2  D3  D4  D5  D6 :=
  C1  10   3   9   8   1   6
  C2   2   7   3   2   7   8
  ;

param EstablishCost :=
  C1  1200
```

```
    C2  1400
    ;
```

Sets `Supplies` and `Demands`, and parameters `Available` and `Required` have the same meaning as before. However, instead of a single parameter for transportation costs, two parameters named `CostA` and `CostB` are introduced for A-type and B-type connections, respectively. Finally, `EstablishCost` denotes the fixed cost of using a center node.

The corresponding definitions in the model section are the following. Note that the sets of all A-type and B-type connections are also defined.

```
set Supplies;
set Centers;
set Demands;

param Available {s in Supplies}, >=0;
param Required {d in Demands}, >=0;

set ConnectionsA := Supplies cross Centers;
set ConnectionsB := Centers cross Demands;

param CostA {(s,c) in ConnectionsA}, >=0;
param CostB {(c,d) in ConnectionsB}, >=0;
param EstablishCost {c in Centers}, >=0;
```

We can still check feasibility of the problem by asserting that the available amounts are larger than required amounts in total.

```
check sum {s in Supplies} Available[s] >= sum {d in Demands} Required[d];
```

Now the data are available in the model section, let us find out what the decision variables in the model shall be. At first glance, the problem seems like two independent transportation sub-problems: one from supplies to centers, and one from centers to demands. However, we cannot solve these two transportation stages separately, because the amounts at the center nodes are not fixed: these are part of the decision.

However, we can solve the two transportation sub-problems in a single model. The necessary decision variables for both transportation problems are simultaneously defined in the model: from supplies to centers, and from centers to demands. The only difference is that the amount of materials delivered to, and sent from the centers are not fixed values in the sub-problems, but variable quantities. Moreover, the amount delivered to and from a center node shall be equal. This is where the two sub-problems are connected, and can be easily established by constraints.

Because of these considerations, variables `tranA` and `tranB` are introduced to denote transportation amounts for connections of A-type and B-type, e.g. from supplies to centers and from centers to demands. In fact, these decisions are sufficient to determine everything about the resulting transportation system, however, additional variables are required. Variable `atCenter` denotes the amount of material going through each center node, which is exactly the amount going in, and going out. This is rather an auxiliary variable as not strictly required but makes implementation easier. However, binary variable `useCenter` denoting whether a center is used at all ($= 1$) or not ($= 0$) is required in order to implement the fixed establishment costs associated with center nodes.

```
var tranA {(s,c) in ConnectionsA}, >=0;
var tranB {(c,d) in ConnectionsB}, >=0;
var atCenter {c in Centers}, >=0;
var useCenter {c in Centers}, binary;
```

Four constraints ensure the consistency of material flow through the whole network. In order, the following rules are expressed.

1. Transportation from supply nodes in total shall not exceed the availability.
2. Transportation to center nodes in total is equal to the `atCenter` value of that center node.
3. Transportation from center nodes in total is equal to the `atCenter` value of that center node.
4. Transportation to demand nodes in total shall be at least the required amount.

```
s.t. Availability_at_Supply_Points {s in Supplies}:
  sum {c in Centers} tranA[s,c] <= Available[s];

s.t. Total_To_Center {c in Centers}:
  atCenter[c] = sum {s in Supplies} tranA[s,c];

s.t. Total_From_Center {c in Centers}:
  atCenter[c] = sum {d in Demands} tranB[c,d];

s.t. Requirement_at_Demand_Points {d in Demands}:
  sum {c in Centers} tranB[c,d] >= Required[d];
```

Note that as we express `atCenter` in two different ways, the two expressions that are equal to `atCenter` are made equal to each other.

A fifth constraint must be added to ensure that a center node can only receive or send materials if it established. The total availability is denoted by `M` which serves as the coefficient in the required big-M constraint.

```
param M := sum {s in Supplies} Available[s];

s.t. Zero_at_Center_if_Not_Established {c in Centers}:
  atCenter[c] <= M * useCenter[c];
```

The objective is the total of all costs, which has three components: A-type, B-type connections, and establishment costs of center nodes.

```
minimize Total_Costs:
  sum {(s,c) in ConnectionsA} tranA[s,c] * CostA[s,c] +
  sum {(c,d) in ConnectionsB} tranB[c,d] * CostB[c,d] +
  sum {c in Centers} useCenter[c] * EstablishCost[c];
```

We can include `printf` statements after the `solve` statement as usual to show the details of the optimal solution found. Our model section is ready.

```
set Supplies;
set Centers;
set Demands;

param Available {s in Supplies}, >=0;
param Required {d in Demands}, >=0;

set ConnectionsA := Supplies cross Centers;
set ConnectionsB := Centers cross Demands;

param CostA {(s,c) in ConnectionsA}, >=0;
param CostB {(c,d) in ConnectionsB}, >=0;
param EstablishCost {c in Centers}, >=0;

check sum {s in Supplies} Available[s] >= sum {d in Demands} Required[d];

var tranA {(s,c) in ConnectionsA}, >=0;
var tranB {(c,d) in ConnectionsB}, >=0;
var atCenter {c in Centers}, >=0;
var useCenter {c in Centers}, binary;

s.t. Availability_at_Supply_Points {s in Supplies}:
  sum {c in Centers} tranA[s,c] <= Available[s];

s.t. Total_To_Center {c in Centers}:
  atCenter[c] = sum {s in Supplies} tranA[s,c];

s.t. Total_From_Center {c in Centers}:
  atCenter[c] = sum {d in Demands} tranB[c,d];

s.t. Requirement_at_Demand_Points {d in Demands}:
  sum {c in Centers} tranB[c,d] >= Required[d];

param M := sum {s in Supplies} Available[s];

s.t. Zero_at_Center_if_Not_Established {c in Centers}:
  atCenter[c] <= M * useCenter[c];

minimize Total_Costs:
  sum {(s,c) in ConnectionsA} tranA[s,c] * CostA[s,c] +
  sum {(c,d) in ConnectionsB} tranB[c,d] * CostB[c,d] +
  sum {c in Centers} useCenter[c] * EstablishCost[c];

solve;

printf "Optimal cost: %g.\n", Total_Costs;
for {c in Centers: useCenter[c]}
{
  printf "Establishing center %s for %g.\n", c, EstablishCost[c];
}
for {(s,c) in ConnectionsA: tranA[s,c] > 0}
{
  printf "A: From %s to %s, transport %g amount for %g (unit cost: %g).\n",
    s, c, tranA[s,c], tranA[s,c] * CostA[s,c], CostA[s,c];
}
for {(c,d) in ConnectionsB: tranB[c,d] > 0}
{
  printf "B: From %s to %s, transport %g amount for %g (unit cost: %g).\n",
    c, d, tranB[c,d], tranB[c,d] * CostB[c,d], CostB[c,d];
}

end;
```

Now we can use the solver to find the optimal decisions for both scenarios in Problem 33. Recall that the small scenario assumes fixed costs of 1200 and 1400 for establishing `C1` and `C2`, while the large scenario assumes 12000 and 14000.

The results of the small scenario are the following.

```
Optimal cost: 7350.
Establishing center C1 for 1200.
Establishing center C2 for 1400.
A: From S1 to C1, transport 100 amount for 500 (unit cost: 5).
A: From S2 to C2, transport 250 amount for 500 (unit cost: 2).
A: From S3 to C1, transport 190 amount for 570 (unit cost: 3).
A: From S4 to C2, transport 210 amount for 840 (unit cost: 4).
B: From C1 to D2, transport 140 amount for 420 (unit cost: 3).
B: From C1 to D5, transport 110 amount for 110 (unit cost: 1).
B: From C1 to D6, transport 40 amount for 240 (unit cost: 6).
B: From C2 to D1, transport 120 amount for 240 (unit cost: 2).
B: From C2 to D3, transport 170 amount for 510 (unit cost: 3).
B: From C2 to D4, transport 90 amount for 180 (unit cost: 2).
B: From C2 to D6, transport 80 amount for 640 (unit cost: 8).
```

The results of the large scenario are the following.

```
Optimal cost: 21310.
Establishing center C2 for 14000.
A: From S1 to C2, transport 100 amount for 1000 (unit cost: 10).
A: From S2 to C2, transport 250 amount for 500 (unit cost: 2).
A: From S3 to C2, transport 190 amount for 1330 (unit cost: 7).
A: From S4 to C2, transport 210 amount for 840 (unit cost: 4).
B: From C2 to D1, transport 120 amount for 240 (unit cost: 2).
B: From C2 to D2, transport 140 amount for 980 (unit cost: 7).
B: From C2 to D3, transport 170 amount for 510 (unit cost: 3).
B: From C2 to D4, transport 90 amount for 180 (unit cost: 2).
B: From C2 to D5, transport 110 amount for 770 (unit cost: 7).
B: From C2 to D6, transport 120 amount for 960 (unit cost: 8).
```

As we can see, the optimal solution for the small scenario uses both center nodes, in an almost equal distribution of materials. Therefore establishing both centers for 2600 is advantageous. There is no need to deliver to both center nodes from the same supply, and there is only a single demand `D6` where delivery is made from both center nodes. It seems like establishment costs are paid, and then the cheapest possible routes are used from supplies to demands.

In contrast, the establishment costs are too high in the large scenario, so establishing both centers is not beneficial. However, we still need at least one center node for the network, and the optimal solution chooses `C2` with a fixed cost of 14000. Assuming this decision, transportation is straightforward: every supply is transported to `C2` and then from `C2` to demands. Note that `C2` is more expensive than `C1` by 2000, but choosing `C2` seems to be justified by larger transportation costs across `C1`.

6.8 Transportation problem – Summary

The basic case of the transportation problem was first presented, which involves a set of supply and demand nodes, and transportation costs between these. The implementation was later improved by the introduction of index sets and some additional features of GNU MathProg.

The transportation problem results in an LP model, and transportation costs for each connection are linear, e.g. can be described by a single cost coefficient, which multiplied by the transported amount gives the total cost. Four cases of more elaborate cost functions, example problems and their implementations were also presented, which are the following.

- *Increasing unit costs.* Two factors and a threshold describe the situation, but we still use an LP model.
- *Economy of scale.* Two factors and a threshold describes the function again, but the cost above the threshold is now less. Binary decision variables are required, making it an MILP model.
- *Fixed costs.* A fixed cost once paid instead of a proportional cost describes the cost function. A binary variable is used, the model is MILP.
- *Linear penalties.* Flexible demands were assumed, and any difference, either positive or negative, was penalized by a linear factor. This results in an LP model.

Finally, the transportation problem was extended by center nodes, which divides the whole problem into two related sub-problems, and a fixed cost decision problem is also involved.

Chapter 7

MILP models

In the previous chapters, proposed problems were mostly solvable by an LP model. Nevertheless, some advanced problem elements like order fulfillment in a whole, economy of scale and fixed costs required integer variables, resulting in an MILP model solution. But the main problems involved, like the production, the diet and transportation problems were **continuous** in nature. That is, variables can take any value in an interval of real numbers. Integer values were only required in specific circumstances.

In this chapter, we will see several optimization problems that involve **discrete** decisions, where the solution space is not a convex continuous region, but finite – or divided into finite parts. These problems naturally require integer, mostly binary variables to express discrete decisions, always resulting in MILP models. Complexity is usually substantially higher than for problems involving continuous decisions only. Some not too large MILP problem instances are difficult to impossible to be exhaustively solved. The exact limit depends on the problem itself, the MILP model applied, the solver software, its configuration and the machine used.

7.1 Knapsack problem

One of the simplest optimization problems that are discrete by nature is the **knapsack problem** [18], its definition follows.

Problem 34.
Given a set of items, each having a nonnegative weight and a gain value. A weight limit is known. Select some of the items so that their total weight does not exceed the given limit, and the total gain is maximal.

The problem is termed as knapsack problem because it can be explained as we have a huge knapsack which must be filled with some of the items, but the total weight we can carry is limited. Therefore we must carefully choose which items we carry, in order to obtain the highest possible gain.

As usual, a concrete knapsack problem instance is described for the general problem

Problem 35.
Solve Problem 34, the knapsack problem, with a weight capacity of 60*, and the following items.*

Item	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*
Weight	16.1	19.2	15.0	14.7	11.3	20.1	17.5	14.5	14.8	18.1
Gain	4.4	4.9	4.3	4.0	3.7	5.1	4.6	4.2	4.3	4.8

Starting with the GNU MathProg implementation, first we can define the sets and parameters describing problem data. The set `Items` denotes the items, and there are two parameters for each item: `Weight` and `Gain`. A single `Capacity` number describes the weight limit. We set all of these as nonnegative to avoid errors in the data section.

```
set Items;

param Weight {i in Items}, >=0;
param Gain {i in Items}, >=0;

param Capacity, >=0;
```

Implementing the data section according to Problem 35 and the sets and parameters described here is straightforward.

```
set Items := A B C D E F G H I J;

param Weight :=
  A  16.1
  B  19.2
  C  15.0
  D  14.7
  E  11.3
  F  20.1
  G  17.5
  H  14.5
  I  14.8
  J  18.1
  ;
param Gain :=
  A  4.4
  B  4.9
  C  4.3
  D  4.0
  E  3.7
  F  5.1
  G  4.6
  H  4.2
  I  4.3
  J  4.8
  ;

param Capacity := 60;
```

Now let us think about our freedom in choosing a solution for the knapsack problem. Any subset of all items is a possible solution, including none and all items selected. If the number of items is n, then the number of different solutions is 2^n. Note that not all of these subsets are feasible because of the weight limit, and we can rule out many non-optimal solutions as well. Nevertheless, the optimization problem for general weights and gains is NP-hard. Therefore, for a large n, an exhaustive solution can be practically impossible.

In mathematical programming, we only express our freedom in decision variables. This can be done easily in the form of a single binary variable per item. We name it `select`, and denotes whether the item is chosen into the knapsack ($= 1$) or not ($= 0$).

```
var select {i in Items}, binary;
```

There is only one restriction which can prevent a solution from being feasible: the total weight of the items. We express in a single constraint that the total weight of selected items is at most the limit given. The total weight of the knapsack is obtained by adding each `select` variable multiplied by the weight of the object. This gives the correct result, because if an item is selected in the knapsack, its weight is added, but if not selected, nothing happens in terms of total weight of the knapsack.

```
s.t. Total_Weight:
  sum {i in Items} select[i] * Weight[i] <= Capacity;
```

The objective is the total gain which is maximized. We add each `select` variable multiplied by the `Gain` value of the item.

```
maximize Total_Gain:
  sum {i in Items} select[i] * Gain[i];
```

After the `solve` statement, we add some printing work to show us the solution. Our model section is ready.

```
set Items;

param Weight {i in Items}, >=0;
param Gain {i in Items}, >=0;

param Capacity, >=0;

var select {i in Items}, binary;
# var select {i in Items}, >=0, <=1;

s.t. Total_Weight:
  sum {i in Items} select[i] * Weight[i] <= Capacity;

maximize Total_Gain:
  sum {i in Items} select[i] * Gain[i];

solve;

printf "Optimal Gain: %g\n", Total_Gain;
printf "Total Weight: %g\n",
  sum {i in Items} select[i] * Weight[i];
for {i in Items}
```

```
{
  printf "%s:%s\n", i, if (select[i]) then " SELECTED" else "";
}

end;
```

Solving Problem 35 reports an optimal gain of 17.1, which is obtained by selecting items C, E, I and J. The total weight of these is 59.2, which fits into the limit of 60 given, although not perfectly.

Note that we used a **conditional expression** to print the word `SELECTED` after each object `i` for which `select[i]` is 1, and nothing where it is 0. The condition of such an expression shall be a constant expression that can be interpreted as a logical value. Remember that variables only behave as constants after the `solve` statement in the model section. The output produced is the following.

```
Optimal Gain: 17.1
Total Weight: 59.2
A:
B:
C: SELECTED
D:
E: SELECTED
F:
G:
H:
I: SELECTED
J: SELECTED
```

One may think about an easy and straightforward heuristic for the knapsack problem: let us arrange the items in the **order of descending gain/weight ratios**, and select items in this order until we reach the limit. This **greedy** strategy seems good, because the weight limit works as a fixed resource, therefore we need to maximize the gain obtained per unit weight. The term greedy means that each time we make a decision, the most beneficial choice is made according to some heuristic. Items having a larger gain/weight ratio mean a more efficient usage of the weight limit.

Actually the only problem with this strategy is the **slack** weight. We may (and will usually) have some weight under the limit which is not used. However, we might probably obtain a better solution by sacrificing some items with a higher gain/weight ratio, in order to fill the weight limit better with other items. This is the small difference which makes the straightforward heuristic non-optimal, and the knapsack problem difficult.

Let us consider a „relaxed" version of the knapsack problem, where items are actually fluids. This means, we are allowed to select a fraction of an item into the knapsack, which means only the same fraction of its weight and gain is obtained. For this relaxation, the binary variable can be replaced by a continuous one, as follows.

```
var select {i in Items}, >=0, <=1;
```

Note that this effect can also be achieved if the original model is run with `glpsol`, with the `--nomip` option added, as follows.

```
glpsol -m knapsack.mod -d example.dat --nomip
```

Furthermore, we change the output method a bit. Instead of printing a `SELECTED` word, we print the actual value of the now continuous `select` variable, as follows.

```
for {i in Items}
{
  printf "%s: %g\n", i, select[i];
}
```

Solving the relaxed model with the exactly same data gives the following results.

```
Optimal Gain: 17.7025
Total Weight: 60
A: 0.273292
B: 0
C: 1
D: 0
E: 1
F: 0
G: 0
H: 1
I: 1
J: 0
```

We can observe that `C`, `E` and `I` are common, but instead of choosing `J` as the fourth and last selected item, the relaxed model chooses `H`, and then `A` in a fractional ratio. With this selection, a prefect weight limit utilization of 60 is obtained, and a slightly better objective of 17.7025 instead of 17.1. If we further analyze problem data, it turns out that the decreasing order of gain/weight ratio is `E`, `I`, `H`, `C`, `A`, `D`, `J`, `G`, `B`, `F`. Therefore the relaxed model does exactly what the greedy heuristic describes: it selects the first four items and then the fraction of the fifth (`A`) to fill the weight limit. Whereas the optimal solution to the actual, integer programming problem does not choose `H`, instead it chooses `J`, which is way down in the order, but fills better the weight limit by its larger weight and gain.

Now that we have seen what complications may arise in case of an integer problem, let us consider another, similar problem, the **multi-way number partitioning problem** [19].

Problem 36.
Given a set of $N \geq 1$ real numbers, divide them exhaustively into exactly $K \geq 1$ subsets so that the difference between the smallest and largest sum of numbers in a subset is minimal.

The multi-way number partitioning problem can be interpreted as a modified version of the knapsack problem where all N items must be packed into any one of K given knapsacks, and this distribution shall be as balanced as possible. There is no gain value in the multi-way number partitioning problem, (or, it can be interpreted as being equal to the weight of the item).

We solve one example problem.

Problem 37.
Distribute the $N = 10$ items described in Problem 35, the knapsack example problem, into $K = 3$ knapsacks so that the difference of the lightest and the heaviest knapsack is minimal.

Let us see how this can be implemented in GNU MathProg. First, there is an `Items` set and a `Weight` parameter as before, but there is no `Gain` parameter. Instead, we define a single integer parameter `Knapsack_Count`, which refers to the positive integer K in the problem description.

```
set Items;

param Weight {i in Items}, >=0;

param Knapsack_Count, integer;
set Knapsacks := 1 .. Knapsack_Count;
```

We also introduced the set `Knapsacks`. However, instead of reading this set from the data section, we denote each knapsack by numbers from 1 to K. The operator `..` defines a set by the smallest and largest integer element, enlisting all integers in between.

Note that we require `Weight` to be nonnegative, however, these parameters can be restricted to integers in general, or relaxed to take values. The model is exactly the same in all cases.

There are more decisions to be made as in the original knapsack problem. For each item, we do not only decide whether it goes to the knapsack or not, but we now decide which knapsack it goes into. This can be done by defining a binary decision variable `select` for each pair of an item and a knapsack, denoting whether the item goes into that particular knapsack or not. These decisions determine the situation well, but auxiliary and other variables are needed to express the objective function concisely. Therefore we also introduce an auxiliary variable `weight` for each knapsack, denoting its total weight, and `min_weight` and `max_weight` for the minimal and maximal knapsack weight.

```
var select {i in Items, k in Knapsacks}, binary;
var weight {k in Knapsacks};
var min_weight;
var max_weight;
```

There is only one constraint which establishes whether a decision about knapsacks is feasible: each item must go exactly into one knapsack, not more and not less. If we add some binary variables and set the sum equal to one, then its only feasible solution is when one binary variable is 1 and all others are 0. Therefore the constraint is the following.

```
s.t. Partitioning {i in Items}:
  sum {k in Knapsacks} select[i,k] = 1;
```

We provide three additional constraint statements, to express the calculation of the weight of each knapsack, a lower limit on all knapsack weights `min_weight` and an upper limit `max_weight`, respectively.

```
s.t. Total_Weights {k in Knapsacks}:
  weight[k] = sum {i in Items} select[i,k] * Weight[i];

s.t. Total_Weight_from_Below {k in Knapsacks}:
  min_weight <= weight[k];

s.t. Total_Weight_from_Above {k in Knapsacks}:
  max_weight >= weight[k];
```

The objective can be the difference of the upper and lower limit.

```
minimize Difference: max_weight - min_weight;
```

This design had been used before, for minimizing errors in equations (see Section 4.8), maximizing minimum production volumes (see Section 5.3), and some cost functions (see for example Sections 6.3 or 6.6). The key is that the solver is allowed not to assign the actual minimum and maximum weights to the variables `min_weight` and `max_weight` to obtain feasible solutions. It is just not beneficial to do, and therefore those solutions where these two bounds are not strict are automatically ruled out.

Finally, we print the contents of each knapsack after the `solve` statement, and our model section is ready.

```
set Items;

param Weight {i in Items}, >=0;

param Knapsack_Count, integer;
set Knapsacks := 1 .. Knapsack_Count;

var select {i in Items, k in Knapsacks}, binary;
var weight {k in Knapsacks};
var min_weight;
var max_weight;

s.t. Partitioning {i in Items}:
  sum {k in Knapsacks} select[i,k] = 1;

s.t. Total_Weights {k in Knapsacks}:
  weight[k] = sum {i in Items} select[i,k] * Weight[i];

s.t. Total_Weight_from_Below {k in Knapsacks}:
  min_weight <= weight[k];

s.t. Total_Weight_from_Above {k in Knapsacks}:
  max_weight >= weight[k];

minimize Difference: max_weight - min_weight;

solve;

printf "Smallest difference: %g (%g - %g)\n",
  Difference, max_weight, min_weight;
for {k in Knapsacks}
{
  printf "%d:", k;
  for {i in Items: select[i,k]}
  {
    printf " %s", i;
  }
  printf " (%g)\n", weight[k];
}
```

```
end;
```

Solving the example Problem 37 gives the following result.

```
Smallest difference: 2.5 (55.3 - 52.8)
1: C F J (53.2)
2: D E H I (55.3)
3: A B G (52.8)
```

The ten items could be divided into three subsets of roughly equal size. The largest knapsack is the second, with a weight of 55.3, and the smallest is the third, with 52.8, but note that the order of knapsacks is not important. Because all ten items are distributed, it is guaranteed that the sum of the three knapsacks is a constant, therefore their average is also constant, and must be in between the two limits.

Another similar, interesting, and better-known problem is the so-called **bin packing problem**, where all items are known, but the knapsack sizes are fixed, therefore the number of knapsacks (called bins) is to be minimized [20]. This could also be solved in GNU MathProg, but is not detailed here.

7.2 Tiling the grid

The knapsack and similar problems require items to „fit” somewhere. Fitting means that each item has a weight, and the sum of the weights is under a constraint. But what happens when fitting is more complex? For instance, considering *size* or shape of items and their container is also a common real-world question can lead to much more difficult optimization problems.

A two-dimensional example is the class of **tiling problems**, where copies of the same shape called **tiles** are used to cover a region in the plane. Tiles are usually forbidden to overlap, and the cover is often required to be perfect, i.e. all of the designated region is covered. If we do not require perfect covering, then one may ask what is the most area we can cover, which is an optimization problem. Provided that there is a single tile, the congruent copies of which are used, the optimization problem simplifies to maximizing the non-overlapping tiles that can be put into the region.

In this section, we restrict tiling to the rectangular grid of unit squares. The tile, its all possible positions, and the region to be covered all fit onto unit squares of the same grid. The tile will be the simplest **cross** consisting of five squares. For the sake of simplicity, the region to be covered is a rectangular area (see Figure 8). Tiles cannot cover area outside the region.

Problem 38.
Determine the maximum number of non-overlapping crosses in an $N \times M$ grid.

One might argue that the general problem itself is very special in the class of tiling problems. Indeed, many tiling problems are not solvable effectively by mathematical programming tools. Here, on the term general problem, we mean a general model section which is applicable to any problem instance.

The data for the problem is very short: only the dimensions of the rectangular area must be specified. Let us see some examples.

Problem 39.
Solve Problem 38, the rectangle tiling problem with crosses for the following rectangle sizes.

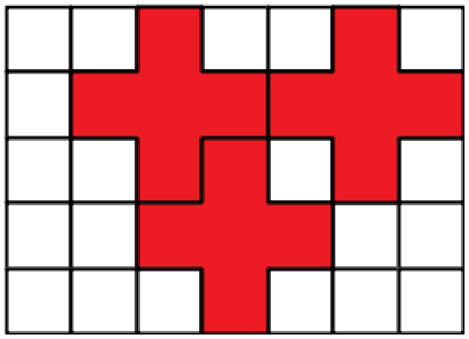

Figure 8: Rectangular area imperfectly covered by some five-square cross tiles.

- 6×4
- 10×10
- 30×30

Probably the shortest data sections so far are required for this problem. The following is for the smallest instance.

```
param Width := 6;
param Height := 4;
```

We use parameter names `Width` and `Height` to use for the dimensions of the rectangular region to be tiled. These must be positive integers. Parameter `Height` represents N, which is the number of rows, and `Width` represents M, which is the number of columns in the area. Rows are denoted by numbers from 1 to N, while columns are denoted by numbers from 1 to M. The set of all squares, called **cells** in the rectangular grid is the Cartesian product of the sets of rows and columns, having a size of $N \cdot M$. These are defined as shown below.

```
param Height, integer, >=1;
param Width, integer, >=1;

set Rows := 1 .. Height;
set Cols := 1 .. Width;
set Cells := Rows cross Cols;
```

At this point, the minimal requirement of data is defined in the model, but a concise implementation will require some more. Let us see first what kind of decisions we can make to describe a tiling.

First, we want to do something like „deciding for all possible positions of a cross tile whether to put a tile there or not". This can be done with a binary variable introduced for each such position as follows.

```
var place {(r,c) in Tiles}, binary;
```

Here, the set `Tiles` refer to some set not yet characterized, but listing all possible positioning of cross tiles. Decisions for this `place` variable well determine all required properties of the tiling, but an auxiliary variable will also be useful.

```
var covered {(r,c) in Cells}, binary;
```

Here, `covered` is defined for each cell, and decides whether the cell is covered by a placed tile or not. This helps us formulating constraints that each cell can only be covered at most once to avoid overlapping tiles.

As usual, the value of 1 means **yes**, and 0 means **no** for all of these binary variables. For instance, a `place` variable is 1 if the tile is placed on the particular position, and a `covered` variable is 1 if the particular cell is covered by some tile.

Before going forward, we must somehow characterize the set `Tiles` describing all possible tile positions. Luckily, the five-cell cross is symmetric: rotating or reflecting it does not yield a different orientation. That means the only difference between tile positions is shifting. In short, we might call it position. Therefore a binary variable is introduced for each possible position of the tile, and the set `Tiles` just contains these positions.

Note that if we do want to tile with asymmetric shapes where multiple orientations are possible, we must define different binary decision variables for each different orientation *and* position. In that case, the `Tiles` set would need a third index dimension to denote orientation, apart from the first two dimensions denoting position. But since the cross only has a single orientation, that index dimension is simply omitted from our model, and `Tiles` is just a two-dimensional set listing positions.

Finally, let us decide how two-dimensional coordinates may define positioning of tiles. For this purpose, we define an **anchor point** of the cross tile as the central cell of the tile. The set `Tiles` will enlist all possible anchor points of correctly positioned tiles. This is valid, because different positions of tiles have different anchor points.

The selection of the anchor point relative to the tile can be arbitrary, it just needs to be consistent throughout the model. It is also possible to be outside the tile, for example, it could also be the corner cell of the 3×3 square containing the cross tile.

The last thing which remains is to determine the anchor points. All correctly positioned cross tiles in the rectangular grid must be considered. Clearly, because the anchor point is a cell inside the tile, the anchor point of a correctly positioned tile must be in the rectangular grid as well. However, not all cells of the rectangular grid can be anchor points. For example, no cells at the edge of the rectangle are valid choices, as it cannot cover area outside the region. Moreover, the corner cells of a rectangular grid cannot even be covered by any correctly positioned cross tiles.

Based on an anchor `(r,c)`, we can define all cells of the cross tile if placed onto that anchor. This is done by the `CellsOf` set. Note that this `set` statement is itself indexed over all cells (that is, possible anchors), which mean that $N \times M$ different sets of 5 cells each are defined.

```
set CellsOf {(r,c) in Cells} :=
  {(r,c),(r+1,c),(r-1,c),(r,c+1),(r,c-1)};
```

The set `Tiles` of correct tile positions (anchor points) are those for which these 5 cells defined in `CellsOf` are all within the rectangular area.

```
set Tiles, within Cells :=
  setof {(r,c) in Cells: CellsOf[r,c] within Cells} (r,c);
```

Of course, we could have answered the question about correct anchor points right at the beginning: these are exactly the cells not on the edges. However, this approach is chosen for two reasons.

- Defining all cells of the placed tile, and keeping only those positions that are entirely in the region to be tiled is a very general approach. It would work not only on arbitrary tiles, but on arbitrary regions as well, on a finite grid.

- The sets `CellsOf` provide us a shorter model formulation, as we will see.

The definitions `CellsOf` and `Tiles` must be placed *before* the definition of variable `place`.

We need two things to be established by constraints. The first is the calculation of the auxiliary variable `covered` for each cell. The second is that each cell can only be covered at most once. Surprisingly, both can be done in a single constraint statement as follows.

```
s.t. No_Overlap {(r,c) in Cells}: covered[r,c] =
  sum {(x,y) in Tiles: (r,c) in CellsOf[x,y]} place[x,y];
```

In words, for each cell we add the `place` variables of all tiles that cover that cell. As `place` is an integer variable, the sum exactly yields the number of tiles covering that particular cell. This can either be zero, or one, but two or more is forbidden. The latter is implicitly assured because the sum equals the single binary variable `covered[r,c]` and it cannot be more than one, as it is a binary variable.

Note that in fact, variable `covered` does not even need to be formulated as binary. The constraint ensures that its value is integer, because it is obtained as a sum of integer variables. The only required property of variable `covered` is its upper bound, 1.

This knowledge might be useful if the complexity of the problem is analyzed. Generally, the more binary variables are there, the more difficult the model is. However, as `covered` is not necessarily binary, it is not expected to increase complexity that much. The true integer nature and difficulty of the problem is based on the `place` variable, denoting tile placements. Nevertheless, marking `covered` binary might alter the course of the solution algorithm.

The objective is the number of tiles placed in total.

```
maximize Number_of_Crosses:
  sum {(r,c) in Tiles} place[r,c];
```

After the `solve` statement, a good way to print the solution is to visualize the tiling itself with some character graphic.

```
for {r in Rows}
{
  for {c in Cols}
  {
    printf "%s",
      if (!covered[r,c]) then "."
      else if ((r,c) in Tiles) then (if (place[r,c]) then "#" else "+")
      else "+";
  }
  printf "\n";
}
```

The output is textual, so we must print each row in one line, and inside a row, we print each cell denoted by the column. A single character is printed for each cell so that the whole rectangular area looks correctly adjusted, for fixed-width fonts.

- If a cell is not covered, a dot (`.`) is put there.
- If a cell is an anchor point of a placed tile, it is denoted by a hash mark (`#`).
- Otherwise, if the cell is covered by a cross but not its center, then it is denoted by a plus sign (`+`).

Note that nested `if` operators are used to obtain the desired result. One might argue that there is an unnecessary `if`, because of the two cases both ending in a `+` sign. The point is that we first check whether `(r,c)` is a proper anchor point or not, and only if it is, then we check the value of the `place[r,c]` variable. We do this because if `place[r,c]` is referred for a non-anchor point `(r,c)`, then out of domain error occurs, as the variable `place` is only defined for anchor points. Note that if an `else` value is omitted, it is assumed to be zero.

The model section for Problem 38 is ready.

```
param Height, integer, >=1;
param Width, integer, >=1;

set Rows := 1 .. Height;
set Cols := 1 .. Width;
set Cells := Rows cross Cols;

set CellsOf {(r,c) in Cells} :=
  {(r,c),(r+1,c),(r-1,c),(r,c+1),(r,c-1)};

set Tiles, within Cells :=
  setof {(r,c) in Cells: CellsOf[r,c] within Cells} (r,c);

var covered {(r,c) in Cells}, binary;
var place {(r,c) in Tiles}, binary;

s.t. No_Overlap {(r,c) in Cells}: covered[r,c] =
  sum {(x,y) in Tiles: (r,c) in CellsOf[x,y]} place[x,y];

maximize Number_of_Crosses:
  sum {(r,c) in Tiles} place[r,c];

solve;

printf "Max. Cross Tiles (%dx%d): %g\n",
  Height, Width, Number_of_Crosses;

for {r in Rows}
{
  for {c in Cols}
  {
    printf "%s",
      if (!covered[r,c]) then "."
      else if ((r,c) in Tiles) then (if (place[r,c]) then "#" else "+")
      else "+";
  }
  printf "\n";
}

end;
```

Solving the model for the smallest instance of a 6×4 area shows that no more than two cross tiles can fit in such a small area. One possible construction is shown below, reported by the model output.

```
Max. Cross Tiles (4x6): 2
....+.
.+.+#+
+#+.+.
.+....
```

The medium instance of a 10×10 area is still solved very fast. At most 13 cross tiles fit in the area, the solution reported is the following.

```
Max. Cross Tiles (10x10): 13
.+...+..+.
+#+.+#++#+
.++..+.++.
.+#+.++#+.
..+++#+++.
.++#+++++#+
+#++.+#++.
.++..++.+.
.+#++#++#+
..+..+..+.
```

The large instance of 30×30 cells was included to show what happens when the model is really big, and thus cannot be solved fast. In such cases, the solver would take an eternity to complete. Therefore, a time limit of 60 seconds is provided. In command line, this can be done by adding the `--tmlim 60` as arguments to `glpsol`.

```
glpsol -m tiling.mod -d example.dat --tmlim 60
```

If the time limit option is omitted, there is no time limit. Note that the limit given this way is not a strong bound of running time actually available for the solver: `glpsol` tends to exceed this limit slightly, especially if preparing steps themselves are tedious.

Running `glpsol` with a one-minute time limit produced the following output.

```
GLPK Integer Optimizer, v4.65
901 rows, 1684 columns, 5604 non-zeros
1684 integer variables, all of which are binary
Preprocessing...
896 rows, 1680 columns, 4816 non-zeros
1680 integer variables, all of which are binary
Scaling...
 A: min|aij| =  1.000e+00  max|aij| =  1.000e+00  ratio =  1.000e+00
Problem data seem to be well scaled
Constructing initial basis...
Size of triangular part is 896
Solving LP relaxation...
GLPK Simplex Optimizer, v4.65
896 rows, 1680 columns, 4816 non-zeros
*     0: obj =  -0.000000000e+00 inf =   0.000e+00 (784)
Perturbing LP to avoid stalling [253]...
```

```
Removing LP perturbation [1896]...
*  1896: obj =   1.631480829e+02 inf =   0.000e+00 (0) 12
OPTIMAL LP SOLUTION FOUND
```

Note that the original model contained a vast 1684 binary variables, the preprocessing only removed four of them (probably the corner cells because those cannot by any means be covered, but we cannot verify this claim).

After that, the LP-relaxation of the MILP model is solved. The rows regarding perturbation tell us that the solver made countermeasures to avoid **stalling**, which is a possible infinite loop in the Simplex algorithm. The presence of this message indicates that the LP solved is itself difficult, very big or have special properties. The last line shows us that 163.15 was the optimal solution of the LP relaxation. This information is valuable, because we know that the optimal solution of the actual MILP model cannot be more than 163. The solution algorithm also uses the result of the LP relaxation.

```
Integer optimization begins...
Long-step dual simplex will be used
+  1896: mip =     not found yet <=              +inf        (1; 0)
+  4865: mip =     not found yet <=   1.630000000e+02        (44; 0)
+  9025: mip =     not found yet <=   1.630000000e+02        (102; 0)
+ 11252: >>>>>   1.380000000e+02 <=   1.630000000e+02  18.1% (176; 0)
+ 14948: mip =   1.380000000e+02 <=   1.620000000e+02  17.4% (200; 35)
+ 18811: mip =   1.380000000e+02 <=   1.620000000e+02  17.4% (266; 35)
+ 22402: mip =   1.380000000e+02 <=   1.620000000e+02  17.4% (330; 36)
+ 25362: mip =   1.380000000e+02 <=   1.620000000e+02  17.4% (384; 36)
+ 28609: mip =   1.380000000e+02 <=   1.620000000e+02  17.4% (457; 36)
+ 29844: >>>>>   1.400000000e+02 <=   1.620000000e+02  15.7% (484; 36)
+ 33482: mip =   1.400000000e+02 <=   1.620000000e+02  15.7% (492; 108)
+ 36688: mip =   1.400000000e+02 <=   1.620000000e+02  15.7% (557; 108)
+ 40050: mip =   1.400000000e+02 <=   1.620000000e+02  15.7% (639; 109)
+ 43326: mip =   1.400000000e+02 <=   1.620000000e+02  15.7% (687; 109)
Time used: 60.0 secs.  Memory used: 6.4 Mb.
+ 43866: mip =   1.400000000e+02 <=   1.620000000e+02  15.7% (698; 109)
TIME LIMIT EXCEEDED; SEARCH TERMINATED
Time used:   60.3 secs
Memory used: 6.7 Mb (7071653 bytes)
```

The integer optimization begins afterwards, which, in general, is usually a **Branch and Bound** procedure. Branching is used to check different values of integer variables one by one, and bounding is used to rule out branches before checking. We do not go into details here.

Rows of output are regularly printed, either periodically or after some important event, for example when a better solution is found.

The numbers on the left, ending in 43866, denote the number „steps" made by the solution algorithm.

The column right of it is the actually known best solution for the problem. If a solution branch is known to be unable to produce a better result (for example, if the relaxation of that branch is even worse), then that branch can be skipped without sacrificing solutions. This speeds up the procedure a lot. The best solution cannot decrease during the search. We can observe the following.

- Initially, the solver had not found any solutions.
- The first solution found involved 138 tiles.

- In 60 seconds this was improved to 140 tiles, then optimization stopped due to the time limit.

The column to the right denotes the current **best bound**. This is an upper limit on the objective of any feasible solutions. This cannot increase during the search, but can decrease, which means improvement. In the example, it is first 163 based on the LP relaxation, then slightly improved to 162. This means that although the solution procedure did not finish, there cannot be more than 162 tiles.

At any time, the optimal solution of the problem lies somewhere between the best solution and the best bound. The solution procedure therefore tries to constantly improve both of these values to eliminate the difference between them. If they are equal, then the optimal solution is found. Sometimes the current best solution is called a **lower bound**, because it is a lower bound for the final optimal solution. Note that if the objective is not maximized but minimized, the same notions are used, but all relations are the opposite.

The relative difference between the best solution and the best bound is called the **integrality gap**, which is usually given in percent. The integrality gap is used as a measure of our knowledge about the optimal solution. The solver tries to minimize the gap, eventually reaching zero.

Concluding the results of the large case, we have a solution for 140 tiles, but the actual optimal solution can be as large as 162, the gap after one minute is 15.7%. It is possible that 140 is indeed the optimal solution, but it is not proven by the solver for sure. Note that these results may be slightly different for each run, especially if different machines or `glpsol` versions are used.

What can we do about a large model, which `glpsol` failed to solve? There are different options based on the exact situation. Note that each option is limited: some problems are just naturally very difficult and simply cannot be solved fast.

- Spend more time solving. Unfortunately, improvement usually tends to slow down after some point. Finishing may take an unacceptably long time.

- Formulate a better model solving the same problem. There is a wide range of practices we can try, including choosing different decision variables, and applying **tightening constraints** which help the solver by cutting out parts of the search space, usually improving the model relaxation. Sometimes, a better formulation can improve efficiency in magnitudes.

- Choosing a different solver or machine. Note that there are more effective LP/MILP solvers than `glpsol`. Example free software are CBC and lpsolve, but even stronger commercial solvers exist. Note that `glpsol` supports exporting of the model into a commonly supported format like `CPLEX-LP`, which is supported by most MILP solvers. There can be surprising performance differences between solvers.

- Even `glpsol` has some options which alter the solution algorithms and can be helpful, including the usage of heuristics and non-default solution algorithms.

We will now demonstrate the last option: running `glpsol` again, but now with the following two heuristic configuration options enabled:

- **Feasibility pumping.** (`--fpump` option) This is a heuristic which aims to obtain good solutions early on in the optimization procedure. This can help the solver later by ruling out branches with too low objective values.

- **Cuts.** (`--cuts` option) This allows all the available cuts `glpsol` knows to be used in the model. Cuts are constraints automatically added to the model to improve the solution algorithm by trimming the search space. These are used for solving MILP models. The cuts enabled by the option are the Gomory, Mixed-Integer Rounding, Clique and Mixed Cover cuts, note that each of these can be enabled or disabled individually.

```
glpsol -m tiling.mod -d example.dat --fpump --cuts
```

The mentioned heuristics often help in solving the problem, sometimes dramatically. But again, there are cases where they do not help, furthermore, the overhead slows down the solution procedure.

The output produced by `glpsol` by solving the 30×30 instance, now with the `--fpump` and `--cuts` options is the following.

```
Integer optimization begins...
Long-step dual simplex will be used
Gomory's cuts enabled
MIR cuts enabled
Cover cuts enabled
Number of 0-1 knapsack inequalities = 784
Clique cuts enabled
Constructing conflict graph...
Conflict graph has 896 + 1004 = 1900 vertices
+  1896: mip =     not found yet <=              +inf        (1; 0)
Applying FPUMP heuristic...
Pass 1
Solution found by heuristic: 153
Pass 1
Pass 2
Pass 3
Pass 4
Pass 5
Cuts on level 0: gmi = 12; clq = 1;
+  2759: mip =   1.530000000e+02 <=   1.620000000e+02   5.9% (15; 0)
+  5134: mip =   1.530000000e+02 <=   1.620000000e+02   5.9% (44; 0)
+  7111: mip =   1.530000000e+02 <=   1.620000000e+02   5.9% (73; 1)
+  8009: mip =   1.530000000e+02 <=   1.620000000e+02   5.9% (89; 1)
+  9294: mip =   1.530000000e+02 <=   1.620000000e+02   5.9% (106; 1)
+ 10509: mip =   1.530000000e+02 <=   1.620000000e+02   5.9% (124; 1)
+ 12685: mip =   1.530000000e+02 <=   1.620000000e+02   5.9% (141; 2)
+ 14071: mip =   1.530000000e+02 <=   1.620000000e+02   5.9% (160; 2)
+ 15544: mip =   1.530000000e+02 <=   1.620000000e+02   5.9% (181; 2)
+ 16909: mip =   1.530000000e+02 <=   1.620000000e+02   5.9% (195; 3)
+ 17875: mip =   1.530000000e+02 <=   1.620000000e+02   5.9% (211; 3)
Time used: 60.2 secs.  Memory used: 10.8 Mb.
+ 18551: mip =   1.530000000e+02 <=   1.620000000e+02   5.9% (225; 3)
TIME LIMIT EXCEEDED; SEARCH TERMINATED
Time used:   60.5 secs
Memory used: 13.1 Mb (13730993 bytes)
```

Note that we can see traces of preprocessing and mid-time work associated with some of the cuts. However, the absolute improvement was due to the feasibility pumping, which obtained a feasible solution of 153 tiles. This is substantially better than the 140 before. The gap is much smaller, just 5.9% instead of 15.7%.

Note that if we are unsure about the complexity of our model, we can always omit time limits, and simply kill `glpsol` on demand, as the current best solution is periodically reported in the output anyways. But in this case, the best found solution itself is not reported, as any subsequent printing work after the `solve` statement is skipped. In contrast, if `glpsol` stops on itself before finding the optimal solution, due to time limit for example, then the variables are all set according to the current best solution instead, and that particular solution will be printed out. For example, this is the reported solution with 153 tiles placed.

```
Max. Cross Tiles (30x30): 153
..+....+.....+...+....+....+..
.+#+.++#+.+.+#+++#+.++#+.++#+.
..+++#++++#++++#++++#++++#+++.
.++#++++#+++#+.+++#++++#++++#+
+#++++#+++..+.++#++++#++++#++.
.+++#++++#+.++#++++#++++#+++..
.+#++++#++++#++++#++++#++++#+.
..+.+#++++#++++#++++#++++#+++.
..+..+++#++++#++++#++++#++++#+
.+#+++#++++#++++#++++#++++#++.
.+++#++++#++++#++++#++++#+++..
+#+.+++#++++#++++#++++#++++#+.
.++.+#++++#++++#++++#++++#+++.
.+#+.+++#++++#++++#++++#++++#+
..++.+#++++#++++#++++#++++#++.
.++#+.+++#++++#++++#++++#+++..
+#++.++#++++#++++#++++#++++#+.
.+.++#++++#++++#++++#++++#+++.
.++#++++#++++#++++#++++#++++#+
+#++++#++++#++++#++++#++++#++.
.+++#++++#++++#++++#++++#+++..
.+#++++#++++#++++#++++#++++#+.
..+++#++++#++++#++++#++++#+++.
.++#++++#++++#++++#++++#++++#+
+#++++#++++#++++#++++#++++#++.
.+++#++++#++++#++++#++++#++.+.
.+#++++#++++#++.+#++++#+++.+#+
..+++#++++#+++..++.+#++++#+++.
..+#++.+#++.+#++#+..+.+#+++#+.
...+....+....+..+......+...+..
```

We can observe that the inside of the 30×30 square is almost perfectly filled with cross tiles. The gap between the best found 153 and the possibly best 162 is left open for now.

7.3 Assignment problem

Another well-known optimization problem is presented in this section, which is the **assignment problem** [21].

Problem 40.
Given N workers and N tasks. For each worker and each task, we know how well that particular worker can execute that particular task, which is described by a cost value for that pair.
Assign each worker exactly one task so that the total cost is maximized.

As usual, the general problem is demonstrated through an example.

Problem 41.

Solve Problem 40, the assignment problem, with the following data. There are $N = 7$ the workers, named `W1` to `W7`, the tasks are named `T1` to `T7`, and the following matrix describes the costs that incurred whenever a particular task is assigned a particular worker.

	$T1$	$T2$	$T3$	$T4$	$T5$	$T6$	$T7$
$W1$	9	6	10	10	8	7	11
$W2$	7	12	6	14	10	5	5
$W3$	8	9	7	11	10	15	6
$W4$	4	10	2	10	6	4	7
$W5$	10	11	7	12	14	9	10
$W6$	5	9	8	9	13	3	8
$W7$	7	12	7	7	11	10	9

Defining the input sets and parameters and implementing the data section can be done in different ways. One possibility is to only read N, the number of workers and tasks, and use a fixed naming convention for them, for example, numbers from 1 to N. But now, we want to allow the naming of workers and tasks from the data section, and therefore two sets named `Workers` and `Tasks` are provided. The only requirement is that these sets must have the same size, otherwise the assignment is impossible. This is established by a `check` statement.

Next, we introduce the `Assignments` set for all possible assignments between workers and tasks. This helps in the formulation later. The `Cost` parameter is then defined for all possible assignments.

```
set Workers;
set Tasks;
check card(Workers)==card(Tasks);
set Assignments := Workers cross Tasks;

param Cost {(w,t) in Assignments};
```

Data for the example Problem 41 can be implemented as follows.

```
data;

set Workers := W1 W2 W3 W4 W5 W6 W7;
set Tasks := T1 T2 T3 T4 T5 T6 T7;

param Cost:
      T1  T2  T3  T4  T5  T6  T7 :=
  W1   9   6  10  10   8   7  11
  W2   7  12   6  14  10   5   5
  W3   8   9   7  11  10  15   6
  W4   4  10   2  10   6   4   7
  W5  10  11   7  12  14   9  10
  W6   5   9   8   9  13   3   8
  W7   7  12   7   7  11  10   9
  ;
```

```
end;
```

There is only one kind of decision to be made in the model: for each assignment, decide whether we assign that particular job to the worker or not. This is a binary variable, we name it `assign`.

```
var assign {(w,t) in Assignments}, binary;
```

There are two rules we must obey when choosing assignments: each worker must have exactly one task, and each task must have exactly one worker assigned. These are implemented as follows.

```
s.t. One_Task_Per_Worker {w in Workers}:
  sum {t in Tasks} assign[w,t] = 1;

s.t. One_Worker_Per_Task {t in Tasks}:
  sum {w in Workers} assign[w,t] = 1;
```

The objective is the total cost, which is obtained by adding each assignment variable multiplied by the associated cost.

```
minimize Total_Cost:
  sum {(w,t) in Assignments} assign[w,t] * Cost[w,t];
```

Finally, after the `solve` statement, we print the optimal total cost, and for each worker, the task assigned. The model section is ready.

```
set Workers;
set Tasks;
check card(Workers)==card(Tasks);
set Assignments := Workers cross Tasks;

param Cost {(w,t) in Assignments};

var assign {(w,t) in Assignments}, binary;

s.t. One_Task_Per_Worker {w in Workers}:
  sum {t in Tasks} assign[w,t] = 1;

s.t. One_Worker_Per_Task {t in Tasks}:
  sum {w in Workers} assign[w,t] = 1;

minimize Total_Cost:
  sum {(w,t) in Assignments} assign[w,t] * Cost[w,t];

solve;

printf "Optimal Cost: %g\n", Total_Cost;
for {w in Workers}
{
  printf "%s->", w;
  for {t in Tasks: assign[w,t]}
  {
    printf "%s (%g)\n", t, Cost[w,t];
  }
}

end;
```

Note that the task assigned to the worker is printed by a `for` loop iterating over all tasks. As each worker has exactly one task assigned, this loop is guaranteed to print exactly one task as desired.

The solution of the example problem is the following. The cost of each assignment is shown in parentheses.

```
Optimal Cost: 42
W1->T2 (6)
W2->T1 (7)
W3->T7 (6)
W4->T5 (6)
W5->T3 (7)
W6->T6 (3)
W7->T4 (7)
```

Therefore, the smallest possible sum of assignments is 42. As usual, it is not guaranteed that the shown solution is the only optimal one.

There are plenty of things we must note about the assignment problem.

- As we can see, the model formulation is relatively simple. The number of different feasible solutions for N workers is exactly $N!$ (factorial), because the assignment of tasks to workers can be regarded as a permutation of the tasks (or the workers). For $N = 7$, it is 5040, not too much to check even by brute-force. But the factorial function quickly rises, in fact, quicker than exponentially. For 20 tasks, the number of solutions is $20! > 10^{18}$, such number of steps is not considered feasible for average computers, for naive brute-force methods.

- With mathematical programming, we can solve assignment problems much larger in size. However, this is one of the few integer programming problems for which a polynomial time algorithmic solution exists. The Hungarian method [22] is specifically designed to the assignment problem, and is substantially faster than using a mathematical programming tool.

- The assignment problem also has a rare property: its LP relaxation yields the optimal result for the MILP model as well. Therefore, the integrality gap is guaranteed to be zero. This means that we do not even need binary variables, simple continuous variables with 0 lower and 1 upper bounds suffice. The reasons behind this property is out of scope of this Tutorial. Nevertheless, this is a useful knowledge because the assignment problem can be solved by LP, for which the practical limits in size are usually much larger than for MILP models.

- The assignment problem is also closely related to the transportation problem. In fact, it can be interpreted as a transportation problem with N sources and N demands, each having an availability and requirement of 1, and the cost matrix for the transportation problem is the same as for the assignment problem. The only difference is that the 1 unit of material cannot be split over multiple connections. But this is not even beneficial: as we previously remarked, the LP relaxation of the problem is sufficient to provide the optimal solution.

- In general, the assignment problem can be part of more complex optimization problems where finding a **bijection** between two sets is part of the decisions. A **bijection**, also called one-to-one correspondence relation between two sets of the same size is when exactly one element from the other set is assigned to each element.
- Another interesting property is that we can add the same number to any row, or any column of the cost matrix, without altering the optimal solution. The only thing which changes is the optimal objective value, which is changed exactly by the number added/subtracted from the row or column.
- This assignment problem shown here involves minimization, but maximization could also be a valid objective. These problems are equivalent. If costs are simply negated, we get the opposite problem, and the solution procedure remains the same.

Now, we will show one interesting extension of the problem. What happens when some decisions about assignments are already made?

Problem 42.
Solve Problem 40, the assignment problem, with the addition of a priori decisions: some possible assignments are explicitly declared to either be used or not used.

This extension is not specific to the assignment problem. Such considerations may arise for every real-world optimization problem. The technique we show here is also not specific.

Our primary goal is to support a priori decisions without compromising the already implemented functionality. That means, old data files not containing any decisions shall still work.

First, we define a new parameter named `Fixing` to express these a priori decisions, for all assignments.

```
param Fixing {(w,t) in Assignments}, in {0,1,2}, default 2;
```

This parameter has three possible values.

- The value 0 means we exclude the assignment from the possible choices.
- The value 1 means we must use this assignment in the solution.
- The value 2 means that we do not decide beforehand whether we want the assignment or not, this decision is rather left as freedom for optimization.

As the value 2 is set as the default for the parameter, if the `Fixing` variable is not mentioned at all in the data sections, then the model assumes that there are no a priori decisions. This addresses compatibility with old data files.

Next, we need these decisions to work. For all `Fixing` values that are either 0 or 1, we explicitly set the `assign` variable to the `Fixing` value. The indexing goes over all possible assignments, but those possible assignments that are left to be decided are filtered out and therefore not fixed.

```
s.t. Fixing_Constraints {(w,t) in Assignments: Fixing[w,t]!=2}:
  assign[w,t] = Fixing[w,t];
```

This completes our new model, which not only solves the assignment problem, but supports a priori decisions to be made, and only optimizes over the restricted set of cases. We now demonstrate its usage by two alternative trials.

First, we set the assignment `W6` to `T6` fixed as zero. This is the only possible assignment that is decided a priori. This case is related to the original example Problem 41, because its optimal solution involves this assignment. Therefore prohibiting this assignment forces the solver to find another solution. The data section modifications and the results are shown below.

```
param Fixing :=
  W6 T6 0
  ;
```

```
Optimal Cost: 42
W1->T2 (6)
W2->T6 (5)
W3->T7 (6)
W4->T5 (6)
W5->T3 (7)
W6->T1 (5)
W7->T4 (7)
```

It turns out that although the `W6` to `T6` assignment has a cost of 3 which is the second smallest in the whole cost matrix, there is still an alternative solution with the objective 42 which omits it.

Now for the second trial, set the assignment `W4` to `T3` as mandatory. This is the cheapest possible assignment in the whole matrix with a cost of 2, but was not included in either optimal solution previously reported. This is the only a priori decision in the second trial. The data added and results are the following.

```
param Fixing :=
  W4 T3 1
  ;
```

```
Optimal Cost: 43
W1->T2 (6)
W2->T7 (5)
W3->T5 (10)
W4->T3 (2)
W5->T1 (10)
W6->T6 (3)
W7->T4 (7)
```

Surprisingly, including assigning `W4` to `T3` turns out to be a bad idea, as the optimal solution with this decision is only 43 now, 42 cannot be obtained anymore.

Further than demonstrating the usage of a priori decisions made for a model, the results show that simple greedy heuristics like taking the smallest cost first does not work perfectly for the assignment problem.

Finally, note that the implementation of a priori decisions by manipulating problem data – particularly for the assignment problem – is also possible.

- If a possible assignment is assigned a very large positive cost, like a positive big-M, then it is not beneficial for the solver to use. Therefore that assignment is effectively excluded.

- If a possible assignment is assigned a very large negative cost, like a negative big-M, then the solver is effectively forced to use that assignment.

A priori decisions can easily make the problem infeasible, if those decisions are contradictory. In case we implement a priori decisions by very large negative and positive big-M costs, then the model will not be infeasible. Instead, it tries to select as few forbidden and as many mandatory possible assignments as possible. Assignments in the solution despite their positive big-M costs, or assignments missing despite their negative big-M costs indicate that the assignment is not possible as described by a priori decisions. Nevertheless, the solver does its best to minimize the objective anyways.

7.4 Graphs and optimization

Two basic problems from the field of graph theory are mentioned now which can be solved by MILP models. For readers interested in topics of graph theory, we offer a good introductory book [23], but the definitions and theorems needed for the two problems are presented in this Tutorial.

A **simple graph** has **nodes** and undirected **edges**, where each edge connects two different nodes, and any two nodes are connected by at most one edge. This is in contrast with non-simple graphs where loops and multiple edges are allowed, and with directed graphs, where edges are substituted by **arcs** directing from one node to another.

Some further definitions are introduced that are required to understand the upcoming problem definitions.

- A **path** in a graph is a sequence of distinct nodes, where the adjacent nodes in the sequence are connected by an edge. If the first and last member of a path are connected by an edge, then together with that edge, the path forms a **cycle** (see Figure 9).
- A graph is **connected** if any two of its nodes can be connected with a path. Otherwise, it is **disconnected** (see Figure 10). Informally, a connected graph is „one big component", and a disconnected graph consists of multiple components.
- A **tree** is a connected graph with no cycles.
- A **spanning subgraph** of a graph is another graph with the *all* its nodes, and the same or fewer edges.
- A **spanning tree** of a graph is a spanning subgraph which is a tree (see Figure 11). Spanning trees are obtained by erasing edges of the graph until it contains no more cycles, but is still connected.

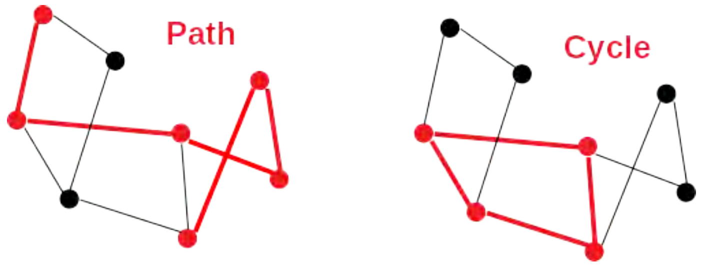

Figure 9: A path and a cycle in the same graph.

Simple graphs are useful for representing the scheme of a network of connections, but sometimes the graph is extended by additional data. One common extension is edge weight, which is a number assigned to each edge. This leads to a **weighted simple graph**. From now on, we assume **weights are positive**. The two problems to be solved are the following.

Problem 43.

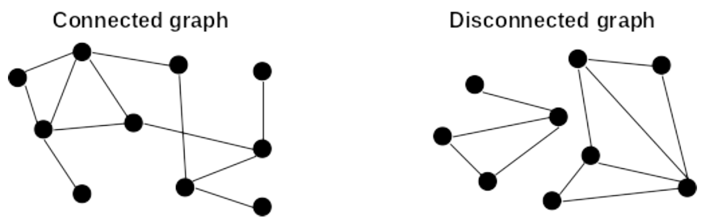

Figure 10: A connected and a disconnected graph.

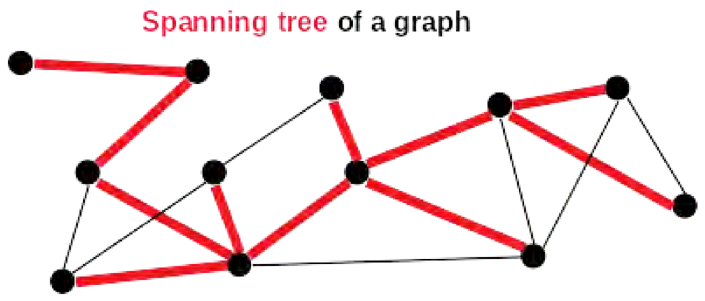

Figure 11: A spanning tree of a graph connects all nodes but does not contain cycles.

*Solve the **shortest path problem** [24] on an arbitrary simple weighted graph, specified as follows. Given two nodes, find a path connecting these two nodes so that the total weight of edges on the path is minimal.*

Problem 44.
*Solve the **minimum weight spanning tree** [25] problem on an arbitrary simple weighted graph: find the spanning tree of the graph with the minimal total edge weight.*

In the shortest path problem, edge weights represent distances between two nodes. This is common in practice, for example in navigation. In the minimum weight spanning tree problem, weights may represent connection establishment costs. This problem may arise in real-world situations where a minimal-cost connected network is to be established between a set of nodes, because the **minimal connected spanning subgraphs are always spanning trees** – we omit the proof here. But this idea is a key observation: we are looking for the minimum weight connected graph on the set of nodes, and this eventually coincides with the minimum weight spanning tree problem.

Before going on, we must note that these problems have very efficient algorithms solving them, running in polynomial time.

- The shortest path problem is solved by Dijkstra's algorithm [26].
- The minimum weight spanning tree problem is solved, for example, by Kruskal's algorithm [27], or Prim's algorithm.

Other approaches are also available. Our aim here is to show how MILP models can be utilized for graphs. Although a specific algorithm can be superior in efficiency, but a mathematical programming model can be much easier to formulate, and to adapt to more complex problems if the problem definition changes.

Figure 12 shows an example of a simple weighted graph, which is used for demonstration.

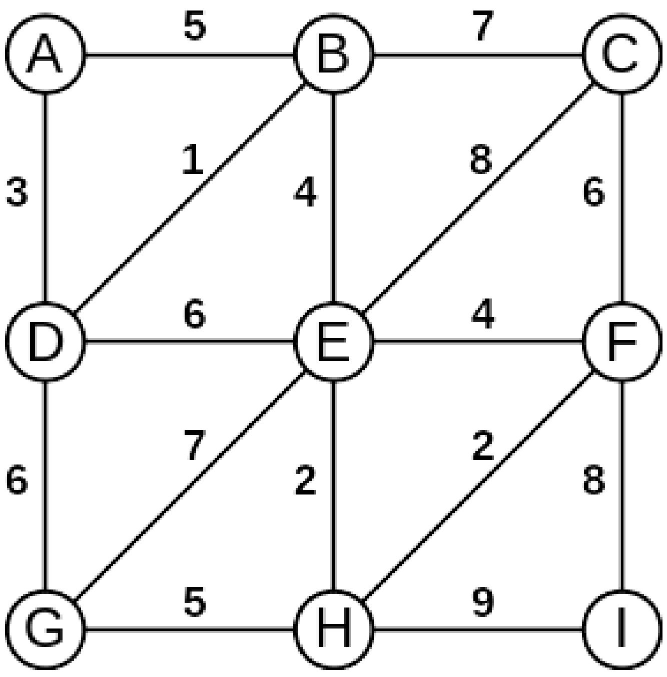

Figure 12: A simple weighted graph with 9 nodes and 16 weighted edges.

Problem 45.
On the graph depicted in Figure 12, find the shortest path between nodes A and I, and find the minimum weight spanning tree.

First, we analyze the problems in terms of feasibility. If a graph is disconnected, then there are nodes which cannot be by moving along edges, and there is no connecting path. Disconnected graphs have no spanning trees either. On the other hand, if a graph is connected, there should be paths form any node to any other, and also a spanning tree. We omit proofs of these claims here.

The graph in Figure 12 is connected and clearly have paths and spanning trees, so feasibility is guaranteed.

The main idea about modeling graphs with mathematical programming tools is that *edges are defined by pairs of nodes.* Therefore, edges can be indexed by two-dimensional indices where both dimensions refer to the set of nodes.

We first implement a data section describing the graph under question. The set of `Nodes` has two important elements, the two nodes between which the shortest path is to be found. We name these as `Start` and `Finish` nodes. Note that the two roles are interchangeable. Finally, a `Weight` parameter describes edge weights. Note that only those pairs of nodes are mentioned for which an edge is defined in the graph, and only in one order. For example `A B 5` is in the list, but `B A 5` is not. More on that later.

```
data;

set Nodes := A B C D E F G H I;
param Start := A;
param Finish := I;
param Weight :=
  A B 5
  B C 7
  A D 3
  B E 4
  C F 6
  D E 6
  E F 4
  D G 6
  E H 2
  F I 8
  G H 5
  H I 9
  B D 1
  C E 8
  E G 7
  F H 2
  ;

end;
```

Parameters `Start` and `Finish` have special characteristics. They do not take numeric values as parameters usually do, but values from the set `Nodes`. For this reason, we mark these parameters as `symbolic`. For safety reasons, we also add `in Nodes` to the definitions so that the data section can only provide names from the `Nodes` set previously defined. Also, we assert that the two nodes are different, by a `check` statement.

```
set Nodes;
param Start, symbolic, in Nodes;
param Finish, symbolic, in Nodes;
check Start!=Finish;
```

The notion of edges in simple graph only allows a single, undirected edge between two nodes. That means, edge AB and BA are the same. However, in mathematical programming, it is more convenient to refer to edges as (A, B) ordered pairs, because it is easy to index: both A and B can be any node. There are two approaches to resolve this confusion.

- We could allow only one direction of edges. For example, we can make a convention that for each $X < Y$, XY is considered as an edge, but YX is not. Here $X < Y$ refers to some kind of ordering, like lexicographical. Note that lexicographical comparison of `symbolic` values (strings) is supported in GNU MathProg.

- We could allow all ordered pairs, and therefore work with directed **arcs** instead of undirected edges. Later we can identify two arcs as the same edge if needed, by constraints.

We choose the latter option for two reasons. First, only allowing specific node orders for edges would mess up with the data to be provided, as we cannot exchange the two nodes in the description of an edge. Second, and more importantly, edge direction will be used in the model implementation anyways.

A parameter `Infty` is introduced to serve as a very high edge cost. In both the shortest path and the minimum weight spanning tree problem, if an edge has such a cost, it is not beneficial for the solver to select. This effectively eliminates those edges from the search.

For this reason, we set `Infty` as the default value for the `Weight` parameter. This makes edges not mentioned in the data section automatically excluded from the search by their very large cost.

```
param Infty, default 99999;
param Weight {a in Nodes, b in Nodes}, >0, default Infty;
param W {a in Nodes, b in Nodes} := min(Weight[a,b],Weight[b,a]);
```

Finally, a parameter `W` is introduced for the weight of an edge used in the model, which will be used in the model formulation. The `min` operator ensures the following, for all edges XY:

- If neither of XY or YX have data provided, the weight `W` is `Infty`, therefore the edge is practically excluded.
- If only one of XY or YX have date provided, then `W` will be equal to that given weight. Therefore we only have to mention each edge once in the data section, in arbitrary order of the two nodes it connects.
- If both XY and YX have data provided, then the minimum of these weights is used for both. Note that *this is not an intended functionality*, we should not provide both weights in the data. This could also be asserted by a `check` statement.

The idea for the shortest path problem is the following. We put one imaginary **droplet of material** into the `Start` node, and it will **flow** through the arcs of the graph to finally reach the `Finish` node. We expect the droplet to draw the path we look for. This is the point where the usage of arcs is more convenient than the usage of edges, because the direction of flow is very important.

A binary variable `flow` is introduced for each possible arc `(a,b)`, denoting whether the droplet flows through that arc from `a` to `b` or not. This variable shall be binary, as the droplet cannot split.

```
var flow {a in Nodes, b in Nodes}, binary;
```

No more variables are needed, not even auxiliary ones. The question is what property of the arcs must satisfy in order to actually form a path between the starting and finishing node?

It is a basic idea to check the **material balance** at each node of a graph with flows. In short, material balance ensures the connection between the total amount of material coming in and going out. This concept was implicitly used for the center nodes in the transportation problem, where all materials coming into a center node must have left it towards the demand nodes (see Section 6.7). We also refer to some constraints as material balance constraints if they express a relation between different material amounts, as in Chapter 5.

Now let us see how the material balance works for a single droplet. The quantity under investigation is *the times the droplet enters the node, minus the times it leaves*, we will call this the **balance** at the node.

- From the starting node, the droplet must go out. We may may allow it getting back there, but then it shall go out again each time. Therefore the balance at the starting node is -1.
- To the ending node, the droplet must arrive. We may allow it leaving, but then it must come back again each time. Therefore, the balance at the ending node is 1.
- For any other nodes, the droplet shall arrive several times, but leave so many times as well. Therefore the balance in any other node is 0.

With a single `s.t.` statement, we establish these balances with the following code.

```
subject to Path_Balance {x in Nodes}:
  sum {a in Nodes} flow[a,x] - sum {b in Nodes} flow[x,b] =
  if (x==Start) then -1 else if (x==Finish) then 1 else 0;
```

Now let us understand why this single constraint ensures that the model works, and no more are needed. If a set of arcs is selected by these balance rules, these arcs will form some directed graph inside the original one. Let us start the droplet and travel around a **trail** along the selected arcs. A trail is a sequence of nodes, where adjacent nodes in the sequence are connected by an arc (or edge, in simple graphs).

We start from the starting node, let it be called A_0. Because of the balance, we can get to at least one other node, A_1. Because of the balance there, either we arrived at the finish node, or there must be another node A_2 the droplet can go out. And so on. The balance constraints ensure that each time we arrive at any node, *we are either at the finishing node, or we can select a new arc not already travelled to go out.* The trail is using up the arcs, therefore it cannot be infinite, it must end at some point. And by the balances, the only possible node the trail can end is the finishing point. Therefore, the `flow` variables provide a *feasible trail* between the `Start` and `Finish` nodes.

The objective is the total weight of the selected arcs.

```
minimize Total_Weight:
  sum {a in Nodes, b in Nodes} flow[a,b] * W[a,b];
```

Note that there are two discrepancies between a path to be found and the set of arcs selected in the model.

- The balance constraint only ensures an existing trail, but more arcs are allowed to be selected.
- The trail is not necessarily a path, as trails may visit the same node multiple times.

However, as we have seen many times before, optimization will eventually rule out these differences and will find an actual path. First, it is not beneficial to select additional edges because of their positive weight. Second, trails can be trimmed to paths, by cutting out cycles between visiting the same node multiple times. In practice, it sounds like: „If I can travel from A to B by visiting some places more than once, then I can surely travel without doing so."

We print out the used arcs after the `solve` statement, and our model is now ready.

```
set Nodes;
param Start, symbolic, in Nodes;
param Finish, symbolic, in Nodes;
check Start!=Finish;

param Infty, default 99999;
param Weight {a in Nodes, b in Nodes}, >0, default Infty;
param W {a in Nodes, b in Nodes} := min(Weight[a,b],Weight[b,a]);

var flow {a in Nodes, b in Nodes}, binary;

subject to Path_Balance {x in Nodes}:
  sum {a in Nodes} flow[a,x] - sum {b in Nodes} flow[x,b] =
  if (x==Start) then -1 else if (x==Finish) then 1 else 0;

minimize Total_Weight:
  sum {a in Nodes, b in Nodes} flow[a,b] * W[a,b];
```

```
solve;

printf "Distance %s-%s: %g\n", Start, Finish, Total_Weight;
for {a in Nodes, b in Nodes: flow[a,b]}
{
  printf "%s->%s (%g)\n", a, b, W[a,b];
}

end;
```

Now, let us find the shortest path from A to I in the given graph. We get the following result.

```
Distance A-I: 19
A->D (3)
B->E (4)
D->B (1)
E->H (2)
H->I (9)
```

This means the shortest path is 19, and the path itself is $ADBEHI$. The arcs show the direction of the path from A to I well, but unfortunately the arcs are not in order. Printing them in order is possible in GNU MathProg, but would require extensive workarounds spoiling the simplicity of this model formulation.

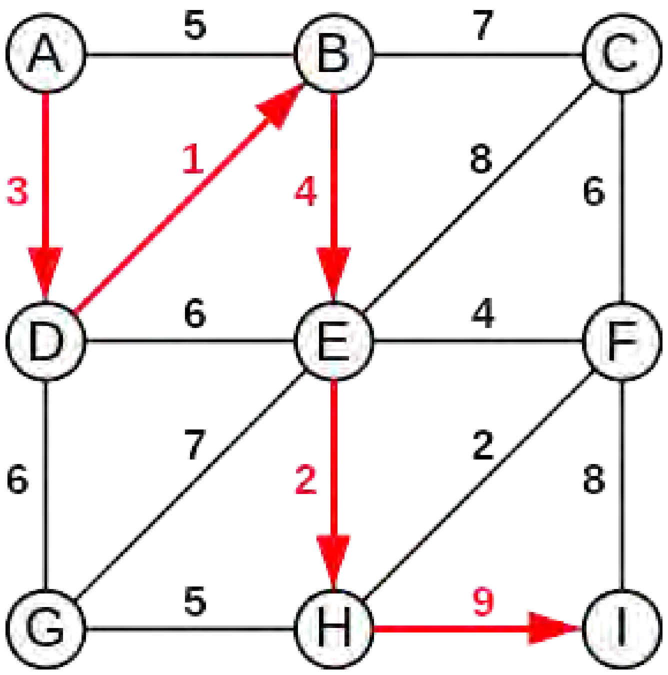

Figure 13: Shortest path of length 19 found from node A to node I.

The path can be seen in Figure 13. We can observe that the shortest path is not the path with the minimal number of edges, and is not always going into the „cheapest" direction.

We make two additional notes on the shortest path problem.

- We could impose stricter constraints, to only allow at most one incoming and outgoing arc at each node. This would eliminate trails visiting nodes multiple times, but would not eliminate unnecessary arcs to be included: note that a cycle between unrelated nodes meets the requirements of even the strict balance constraints.
- The shortest path problem in this form has the same nice property as the assignment problem (see Section 7.3): the LP relaxation of the problem yields the optimal solution. So the model could be an LP instead. This is not surprising: if the droplet is split, then all of its pieces shall still go simultaneously on the shortest path to travel the minimal distance in total.

Now let us solve the minimum weight spanning tree problem on the same graph. This time, we start with the strategy.

Recall the note that the minimal connected spanning subgraph is always a tree. Therefore the optimization has nothing to do with cycles and tree definitions. The objective shall be the total weight of selected edges, and the **constraints shall only ensure that the graph is connected**. Then, optimization will find the minimum weight connected spanning subgraph, which eventually will be a tree.

Now the real question is how we can assure by constraints that some edges form a connected graph of the nodes? We again use the idea of flows. Let us put a single droplet in each node, and these droplets may flow through edges of the graph, eventually arriving into a single designated **sink** node.

If the graph formed by the selected edges is connected, then droplets can reach all nodes from any. If the graph is disconnected, then only part of the nodes is reachable and the flow is infeasible.

Now let us start implement the model based on this idea. The data section can be almost the same, the only difference is that we do not define a `Start` and `Finish` node, only a single `Sink`. The selection of the `Sink` node is arbitrary, and we could use the model section itself to automatically choose one, but providing it in the data section is easier. The role of the `Sink` node is to serve as the common endpoint of all droplets. We select `A` as the `Sink` node.

```
param Sink := A;
```

Edge weights are determined exactly the same way as for the shortest path problem. Therefore the rest of the data section remains unchanged.

```
set Nodes;
param Start, symbolic, in Nodes;

param Infty, default 99999;
param Weight {a in Nodes, b in Nodes}, >0, default Infty;
param W {a in Nodes, b in Nodes} := min(Weight[a,b],Weight[b,a]);
```

We use two kinds of decision variables this time. Variable `use` is defined for each arc, and denotes whether the droplets may flow through that direction ($= 1$) or not ($= 0$). Variable `flow` is continuous, and denotes how much material (eventually, how many droplets) flow through that arc. Note that `flow` can be negative, more on that later.

```
var use {a in Nodes, b in Nodes}, binary;
var flow {a in Nodes, b in Nodes};
```

Without proof, we note that edges of a spanning tree can be uniquely directed towards any one of its nodes. The variable `use` will denote if the edge is selected, *and* it points towards the `Sink` node.

Although both variables are defined for arcs, there is no point for droplets to flow in both directions through an edge. The reason is that their endpoint is the same. Therefore the optimization will select at most one of the directions for each edge. A flow of k droplets in one direction can be regarded as a flow of $-k$ droplets in the opposite direction. The following constraint ensures that if there is flow between two nodes, then one is some positive $k > 0$, which is the actual direction, and the other direction is its exact opposite, $-k < 0$. Alternatively, both flows can be zero as well.

```
subject to Flow_Direction {a in Nodes, b in Nodes}:
  flow[a,b] + flow[b,a] = 0;
```

A positive flow of droplets is only allowed in the arcs selected by the `use` variable. This is established by a big-M constraint. Note that the coefficient is the number of nodes minus one. This is the maximum number of droplets in motion, as the droplet put directly on the `Sink` node does not need to move. Observe that two constraints are generated for a single edge, which represents the two directions, but the constraint for the negative flow is redundant.

```
subject to Flow_On_Used {a in Nodes, b in Nodes}:
  flow[a,b] <= use[a,b] * (card(Nodes) - 1);
```

Finally, establish the material balance of the droplets, which is the following at each node.

- If the node is the `Sink`, then it shall receive the number of nodes minus one droplets.
- For any other node, it must send one droplet. Note that droplets may enter and exit the same node, but these do not contribute to the balance of any node.

```
subject to Material_Balance {x in Nodes}:
  sum {a in Nodes} flow[a,x] - sum {b in Nodes} flow[x,b] =
  if (x==Sink) then (1-card(Nodes)) else 1;
```

It can be proven that the arc and flow selection satisfying the balance constraints provide a connected graph. The logic is similar to the case of the shortest paths: we can start trails at each node and move through positive flows until reaching the `Sink`. The sink is the only node where trails can end. Therefore, all nodes must be connected to the `Sink`, and consequently to each other as well.

The objective is the total weight of arcs where the flow is positive. This minimizes the arcs to be selected, resulting in a spanning tree, all its edges directed towards the `Sink`.

```
minimize Total_Weight:
  sum {a in Nodes, b in Nodes} use[a,b] * W[a,b];
```

We print the selected arcs again after the `solve` statement, and our model section is ready.

```
set Nodes;
param Sink, symbolic, in Nodes;

param Infty, default 99999;
param Weight {a in Nodes, b in Nodes}, >0, default Infty;
param W {a in Nodes, b in Nodes} := min(Weight[a,b],Weight[b,a]);

var use {a in Nodes, b in Nodes}, binary;
var flow {a in Nodes, b in Nodes};
```

```
subject to Flow_Direction {a in Nodes, b in Nodes}:
  flow[a,b] + flow[b,a] = 0;

subject to Flow_On_Used {a in Nodes, b in Nodes}:
  flow[a,b] <= use[a,b] * (card(Nodes) - 1);

subject to Material_Balance {x in Nodes}:
  sum {a in Nodes} flow[a,x] - sum {b in Nodes} flow[x,b] =
  if (x==Sink) then (1-card(Nodes)) else 1;

minimize Total_Weight:
  sum {a in Nodes, b in Nodes} use[a,b] * W[a,b];

solve;

printf "Cheapest spanning tree: %g\n", Total_Weight;
for {a in Nodes, b in Nodes: use[a,b]}
{
  printf "%s->%s (%g)\n", a, b, W[a,b];
}

end;
```

Solving the example with `Sink` node A gives the following results, also visible in Figure 14.

```
Cheapest spanning tree: 31
A<-D (3)
B<-E (4)
D<-B (1)
E<-H (2)
F<-C (6)
F<-I (8)
H<-F (2)
H<-G (5)
```

We can observe that all directed paths unambiguously lead to the designated `Sink` node A. If another `Sink` was selected, then the solution graph could be the same (or another one with exactly the same objective), but the direction of the arcs would be different.

Note that droplets may be split as flows are not integers, but it is not beneficial and therefore not happening in the optimal solution. This is a similarity to the shortest path problem. However, in this case the MILP model cannot be relaxed into an LP, because the selection of the arcs is a mandatory integer part. It does not matter whether only a single or all droplets travel through an edge, its full weight must be calculated. Therefore variable `use` needs to remain binary.

One final note about these MILP formulations: how are loops treated? For the sake of easier indexing, for all nodes `a in Nodes` and `b in Nodes`, the arc `(a,b)` was included in both models. But this does not only allow two directions for the same edge, but also loops. These are edges in the form `(a,a)`, silently present throughout all of the formulation: parameters, variables, constraints and the objective. We can verify that loops are either not beneficial to be used, or using them does not make a difference, in all instances they are present.

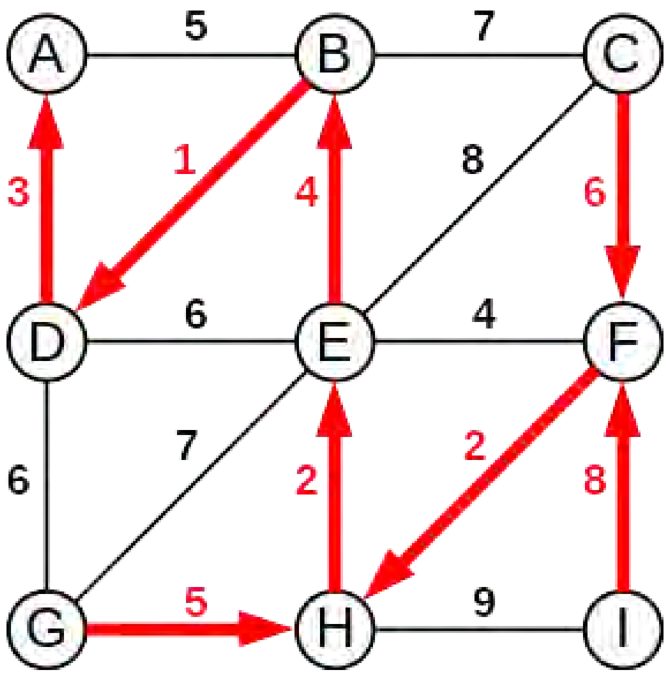

Figure 14: Minimum weight spanning tree (weight is 31), directed towards `Sink` node A.

7.5 Travelling salesman problem

A well-known and notoriously difficult optimization problem and its GNU MathProg model implementation is presented here. This is the **travelling salesman problem**, or **TSP** in short [28].

Problem 46.
A set of nodes and distances between any two are given. Find the shortest cycle visiting all nodes.

The motivation behind TSP is that an agent must visit a set of targets in the least amount of time or least distance travelled, then go back to the starting point. The route, if optimal, will be a cycle. Note that a cycle visiting all nodes of a graph is also called a **Hamiltonian cycle**. The starting node in TSP can be arbitrary, as the cycle visits all nodes anyway.

Comparing the TSP to the knapsack problem, although both are NP-hard in general, the knapsack tends to be solvable for much larger number of items than the number of nodes in a TSP.

In general, the distances can be different between two nodes depending on direction, but we assume they are equal throughout this section.

For the sake of simplicity, we solve the TSP problem for nodes situated on the plane (see Figure 15), and an arbitrary starting point is also determined. The distances are Euclidean, e.g. the ordinary definition of distance between two points on the plane.

Problem 47.
Find the shortest route starting and ending at the green node and visiting all red nodes depicted in Figure 15.

For this particular problem, rather than calculating distance matrix, we implement a model which accepts the planar positions of the nodes instead and calculates Euclidean distances accordingly.

For this reason, a new kind of data section formulation is introduced: the nodes and their

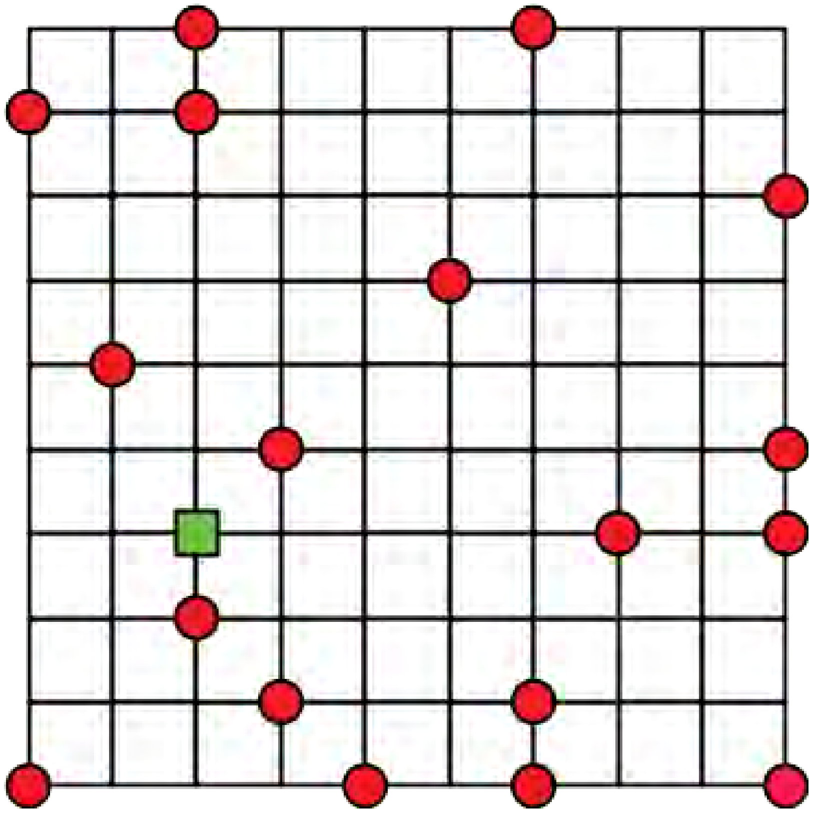

Figure 15: Set of nodes on the plane, for which TSP is to be solved. The green node is designated as an arbitrary starting point for travelling.

positions are listed in a single `set` statement named as `Node_List`.

Another parameter `Start` is for the arbitrary starting node. Its role is similar to the arbitrary sink node in the minimum weight spanning tree problem (see Section 7.4) – it could be avoided, but makes implementation easier.

```
data;

set Node_List :=
  P00 0 0
  P08 0 8
  P15 1 5
  P22 2 2
  P23 2 3
  P28 2 8
  P29 2 9
  P31 3 1
  P34 3 4
  P40 4 0
  P56 5 6
  P60 6 0
  P61 6 1
  P69 6 9
  P73 7 3
  P90 9 0
  P93 9 3
  P94 9 4
  P97 9 7
  ;
param Start := P23;

end;
```

Let us see how such a data format can be implemented in the model file. What we first need is a three-dimensional `set` called `Node_List`. Note that here we must add `dimen 3` to the definition to indicate that each element of this set has three coordinates. These coordinates are the name of the node, its X and Y coordinates in plane, respectively.

```
set Node_List, dimen 3;
set Nodes := setof {(name,x,y) in Node_List} name;
check card(Nodes)==card(Node_List);
param X {n in Nodes} := sum {(n,x,y) in Node_List} x;
param Y {n in Nodes} := sum {(n,x,y) in Node_List} y;
```

After this point, the set `Nodes` is derived from the list, by picking each name mentioned there. Node names shall be unique in the data section. This can be asserted by a `check` statement telling the `Nodes` set has the same size as `Node_List`. If there are duplicates, the `Nodes` set is smaller.

The numeric parameters `X` and `Y` denote the planar coordinates of the nodes. Technically, this is obtained by a `sum`. Since each node name appears in the list exactly once, the sum will result in selecting the corresponding coordinate of that particular node. The `Node_List` set is no longer needed.

The arbitrary `Start` can be any node from the `Nodes` set.

```
param Start, symbolic, in Nodes;
```

The parameter `W` denoting the Euclidean distances are calculated based on the `X` and `Y` parameters. The formula of the Euclidean distance between two points $P_1\left(x_1, y_1\right)$ and $P_2\left(x_2, y_2\right)$ is the following. The implementation in GNU MathProg uses the `sqrt` built-in function and the operator `^` for exponentiation.

$$P_1P_2 = \sqrt{\left(x_1 - x_2\right)^2 + \left(y_1 - y_2\right)^2} \tag{13}$$

```
param W {a in Nodes, b in Nodes} := sqrt((X[a]-X[b])^2+(Y[a]-Y[b])^2);
```

We have all the required data defined in the model, now the solution strategy and appropriate decision variables must be determined.

First, observe that TSP is in close connection with the assignment problem (see Section 7.3). For each node in the TSP, we must decide the next node in the cycle. Therefore the solution of the TSP is an assignment from the set of nodes to itself.

The variable `use` for each arc, and the two constraints ensure that each node have exactly one „next”, and one „previous” neighbor.

```
var use {a in Nodes, b in Nodes}, binary;

subject to Path_In {b in Nodes}:
  sum {a in Nodes} use[a,b] = 1;

subject to Path_Out {a in Nodes}:
  sum {b in Nodes} use[a,b] = 1;
```

Applying the logic of the shortest paths, we can start a trail and go into the direction of the only selected arc outside, obtaining a sequence $A_0A_1A_2\dots$ of nodes. For each $i \geq 1$, the arc going into A_i is $A_{i-1}A_i$, and the arc going out is A_iA_{i+1}. Therefore it is impossible for the sequence to run from A_i into an already visited node A_j, $1 \leq j < i$, because A_j has only one arc going in, and it is $A_{j-1}A_j$, not A_iA_j. This means that the only way the trail can end is running into A_0 and the cycle closes. Ideally, we would obtain a single cycle of the nodes this way (see Figure 16).

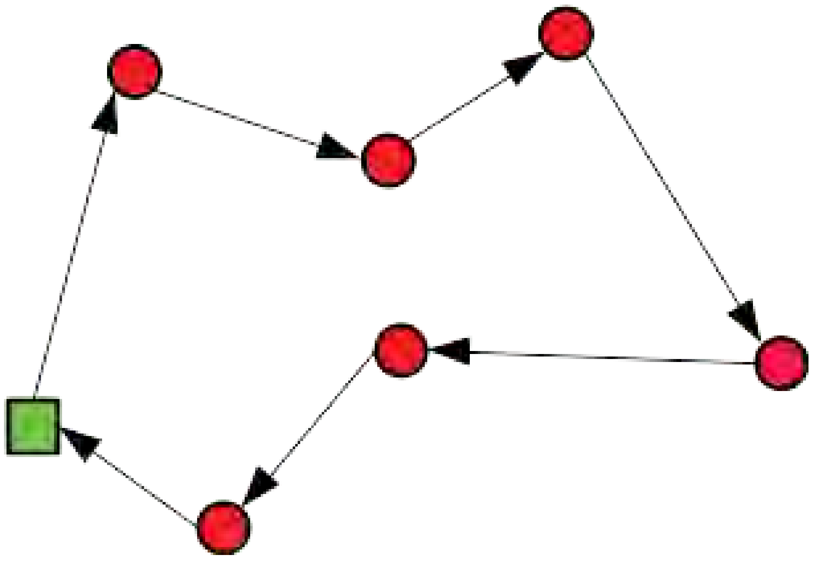
Figure 16: Cycle containing all nodes, representing a feasible solution for TSP.

Unfortunately, this simple solution is not sufficient. The mistake in the above consideration is that the cycle starting from and ending at the `Start` node does not necessarily include all nodes. The assignment of nodes to other nodes can possibly form not a single Hamiltonian cycle, but two or several smaller cycles (see Figure 17).

To prevent the case of more than one cycle, we must ensure *connectivity* of the graph. And this is where the technique shown for the minimum weight spanning tree (see Section 7.4) is reused: a single droplet is put at each node, and they shall flow through selected arcs into the `Start` node. This is possible if and only if the assignment is a single cycle.

Note that the resulting graph is not a tree, but the new variable `flow` and the corresponding constraints guarantee connectivity only, as they did for the minimum weight spanning tree problem.

```
var flow {a in Nodes, b in Nodes};

subject to Flow_Direction {a in Nodes, b in Nodes}:
  flow[a,b] + flow[b,a] = 0;
```

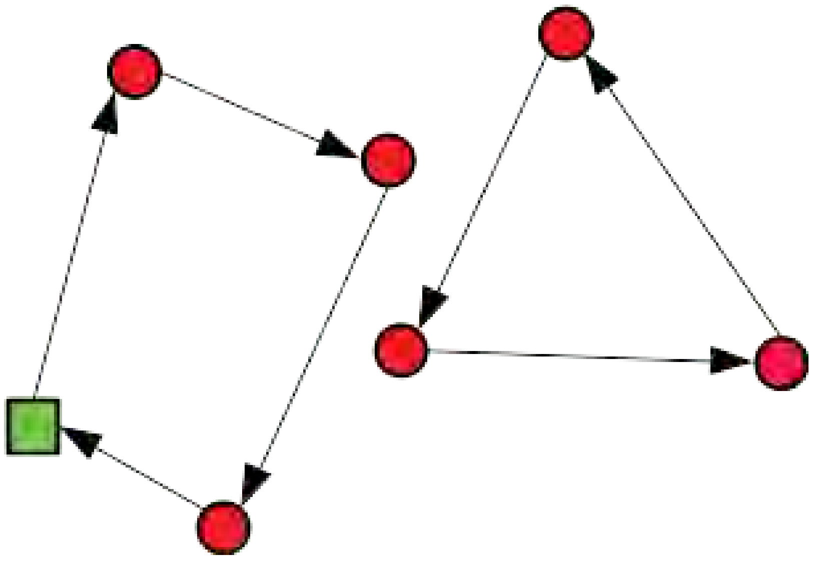

Figure 17: Feasible assignment of nodes but infeasible for TSP.

```
subject to Flow_On_Used {a in Nodes, b in Nodes}:
  flow[a,b] <= use[a,b] * (card(Nodes) - 1);

subject to Material_Balance {x in Nodes}:
  sum {a in Nodes} flow[a,x] - sum {b in Nodes} flow[x,b] =
  if (x==Start) then (1-card(Nodes)) else 1;
```

The objective is the total weight of the selected arcs.

```
minimize Total_Weight:
  sum {a in Nodes, b in Nodes} use[a,b] * W[a,b];
```

We finished the model implementation. As we can see, the TSP model is just the „combination" of the assignment problem and the minimum weight spanning tree problem. The optimization procedure does not end up with a spanning tree, but a Hamiltonian cycle because of the assignment rules. Note that because the assignment problem can be solved by an LP, *the assignment problem can be used as a relaxation of the TSP problem.* Solution techniques for TSP may exploit this fact.

We can again print out the used edges one by one after the `solve` statement, likely not in their correct order.

```
printf "Shortest Hamiltonian cycle: %g\n", Total_Weight;
for {a in Nodes, b in Nodes: use[a,b]}
{
  printf "%s->%s (%g)\n", a, b, W[a,b];
}
```

Solving example Problem 47 gives the following result.

```
Shortest Hamiltonian cycle: 44.5948
P00->P31 (3.16228)
P08->P15 (3.16228)
P15->P34 (2.23607)
P22->P00 (2.82843)
P23->P22 (1)
P28->P29 (1)
P29->P08 (2.23607)
P31->P40 (1.41421)
P34->P23 (1.41421)
P40->P60 (2)
P56->P28 (3.60555)
P60->P61 (1)
P61->P90 (3.16228)
P69->P56 (3.16228)
P73->P93 (2)
P90->P73 (3.60555)
P93->P94 (1)
P94->P97 (3)
P97->P69 (3.60555)
```

Note that although the number of nodes is only 19, it still takes some time to be solved. The output of `glpsol` shows clearly how the solution was gradually improved to optimality.

```
Integer optimization begins...
Long-step dual simplex will be used
+   545: mip =     not found yet >=              -inf        (1; 0)
+  1859: >>>>>   5.969014627e+01 >=   2.300457897e+01  61.5% (48; 1)
+  4111: >>>>>   4.777953551e+01 >=   2.623268810e+01  45.1% (75; 7)
+  7223: >>>>>   4.576632755e+01 >=   2.722682823e+01  40.5% (90; 32)
+ 15676: >>>>>   4.459475467e+01 >=   3.434024294e+01  23.0% (154; 77)
+ 39752: mip =   4.459475467e+01 >=   4.136099600e+01   7.3% (501; 612)
+ 52815: mip =   4.459475467e+01 >=     tree is empty   0.0% (0; 2407)
INTEGER OPTIMAL SOLUTION FOUND
Time used:   9.4 secs
Memory used: 4.7 Mb (4954539 bytes)
```

The optimal cycle length is 44.59. Our manual output tells everything about the solution, but is difficult to interpret. Instead, a visualization is produced, this time by the model section itself. We print out the solution in **Scalable Vector Graphics (SVG)** format [29].

SVG is a vector-graphical image representation format. Instead of storing pixels, the primitives making up the image are described. The SVG relies on XML data format. Therefore, we only need to print properly formatted XML containing the grid, the nodes and the found TSP solution. This whole work goes after the `solve` statement.

Producing SVG with GNU MathProg has more spectacular demonstrations publicly available [30], we basically use the cited approach here. The syntax of neither SVG nor XML is explained in this Tutorial.

First, the particular TSP, Problem 47 uses X and Y coordinates from 0 to 9, therefore the image would fit in a 500×500 image with a 50 pixels distance between grid lines, the edges being padded by 25 pixels. In the SVG to be produced, we refer to x and y coordinates from the top left corner, increasing right and down. First, we translate coordinates in the TSP problem into coordinates in the SVG image as follows.

```
param PX {n in Nodes} := 25 + 50 * X[n];
param PY {n in Nodes} := 475 - 50 * Y[n];
```

We also define an `SVGFILE` parameter to contain the name of the SVG file to be produced. This is `solution.svg` by default, but can be overridden in a data section.

```
param SVGFILE, symbolic, default "solution.svg";
```

We first print an empty string by `printf`. But this is not printed into the result of the `glpsol`, but the file we specify, by adding `>SVGFILE` at the end of the `printf` statement. If we want to append to the end of the file instead, we should write `>>SVGFILE`, similarly to redirection in command line.

```
printf "" >SVGFILE;
```

The first `>SVGFILE` erases the file if it was present, but all subsequent `printf` statements shall end with `>>SVGFILE` to keep previously written content.

The SVG is an XML file, and a properly formatted header is needed.

```
printf "<?xml version=""1.0"" standalone=""no""?>\n" >>SVGFILE;
printf "<!DOCTYPE svg PUBLIC ""-//W3C//DTD SVG 1.1//EN"" " >>SVGFILE;
printf """http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd"">\n" >>SVGFILE;
```

Note that GNU MathProg does not allow too long symbolic names and character strings in code (not more than 100 characters), therefore the work is separated into multiple `printf` statements.

Afterwards, XML tags will be inserted. The SVG content starts with the opening of an `svg` tag, with the width and height of the image. Note that we have to use a lot of `"` characters in the SVG code. These must be escaped as `""` if we want to print such characters with `printf` in GNU MathProg.

```
printf "<svg width=""500"" height=""500"" version=""1.0"" " >>SVGFILE;
printf "xmlns=""http://www.w3.org/2000/svg"">\n" >>SVGFILE;
```

First, a „canvas" is printed, which is a rectangle having the same size as the whole image, providing a uniform pale yellow background. We can use the `&` operator in GNU MathProg to concatenate strings together, to separate a single string into multiple lines.

```
printf "<rect x=""0"" y=""0"" width=""500"" height=""500"" " &
    "stroke=""none"" fill=""rgb(255,255,208)""/>\n" >>SVGFILE;
```

First, the grid is drawn, consisting of 10 vertical and 10 horizontal, narrow black lines at appropriate positions.

```
for {i in 0..9}
{
  printf "<line x1=""%g"" y1=""%g"" x2=""%g"" y2=""%g"" " &
      "stroke=""black"" stroke-width=""1""/>\n",
      25+50*i, 475, 25+50*i, 25 >>SVGFILE;
  printf "<line x1=""%g"" y1=""%g"" x2=""%g"" y2=""%g"" " &
      "stroke=""black"" stroke-width=""1""/>\n",
      25, 25+50*i, 475, 25+50*i >>SVGFILE;
}
```

Next, the TSP solution is drawn. A wide, blue line segment is put between each pair of nodes that appear in the optimal cycle. The direction is not represented, it does not matter anyways.

```
for {a in Nodes, b in Nodes: use[a,b]}
{
  printf "<line x1=""%g"" y1=""%g"" x2=""%g"" y2=""%g"" " &
      "stroke=""blue"" stroke-width=""3""/>\n",
      PX[a], PY[a], PX[b], PY[b] >>SVGFILE;
}
```

Finally, the nodes are printed upon the grid and the blue cycle. All nodes apart from `Start` are printed as small black circles with red fill.

```
for {n in Nodes: n!=Start}
{
  printf "<circle cx=""%g"" cy=""%g"" r=""%g"" " &
      "stroke=""black"" stroke-width=""1.5"" fill=""red""/>\n",
      PX[n], PY[n], 8 >>SVGFILE;
}
```

The `Start` node is a small black rectangle with green fill.

```
printf "<rect x=""%g"" y=""%g"" width=""16"" height=""16"" " &
    "stroke=""black"" stroke-width=""1.5"" fill=""green""/>\n",
    PX[Start]-8, PY[Start]-8 >>SVGFILE;
```

Note that the mentioned order of elements to be printed is important, as it determines how they cover each other in the result. Finally, the `svg` tag is closed.

```
printf "</svg>\n" >>SVGFILE;
```

By running `glpsol` to solve TSP, Problem 47, we do not only get the textual output, but the SVG image file which shows the drawing (see Figure 18).

Note that the TSP problem has more effective implementations than the one shown here, and even this one could be improved significantly to find solutions faster for larger TSP problem instances. The focus was on the relation to the assignment problem and the connectivity constraints. Here we show the full model section, note that the MILP model logic almost as large as the code responsible for the textual and SVG output.

```
set Node_List, dimen 3;
set Nodes := setof {(name,x,y) in Node_List} name;
check card(Nodes)==card(Node_List);
param X {n in Nodes} := sum {(n,x,y) in Node_List} x;
param Y {n in Nodes} := sum {(n,x,y) in Node_List} y;

param Start, symbolic, in Nodes;

param W {a in Nodes, b in Nodes} := sqrt((X[a]-X[b])^2+(Y[a]-Y[b])^2);
```

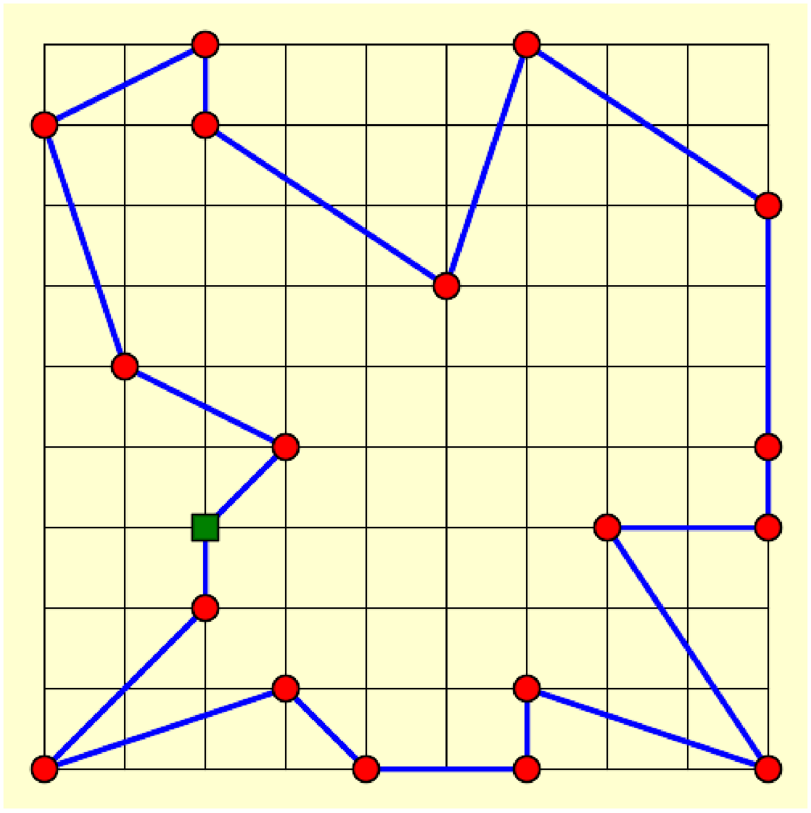
Figure 18: Optimal solution of TSP, Problem 47.

```
var use {a in Nodes, b in Nodes}, binary;
var flow {a in Nodes, b in Nodes};

subject to Path_In {b in Nodes}:
  sum {a in Nodes} use[a,b] = 1;

subject to Path_Out {a in Nodes}:
  sum {b in Nodes} use[a,b] = 1;

subject to Flow_Direction {a in Nodes, b in Nodes}:
  flow[a,b] + flow[b,a] = 0;

subject to Flow_On_Used {a in Nodes, b in Nodes}:
  flow[a,b] <= use[a,b] * (card(Nodes) - 1);

subject to Material_Balance {x in Nodes}:
  sum {a in Nodes} flow[a,x] - sum {b in Nodes} flow[x,b] =
  if (x==Start) then (1-card(Nodes)) else 1;

minimize Total_Weight:
  sum {a in Nodes, b in Nodes} use[a,b] * W[a,b];

solve;

printf "Shortest Hamiltonian cycle: %g\n", Total_Weight;
for {a in Nodes, b in Nodes: use[a,b]}
{
  printf "%s->%s (%g)\n", a, b, W[a,b];
}

param PX {n in Nodes} := 25 + 50 * X[n];
param PY {n in Nodes} := 475 - 50 * Y[n];

param SVGFILE, symbolic, default "solution.svg";
printf "" >SVGFILE;
printf "<?xml version=""1.0"" standalone=""no""?>\n" >>SVGFILE;
printf "<!DOCTYPE svg PUBLIC ""-//W3C//DTD SVG 1.1//EN"" " >>SVGFILE;
printf """http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd"">\n" >>SVGFILE;
printf "<svg width=""500"" height=""500"" version=""1.0"" " >>SVGFILE;
printf "xmlns=""http://www.w3.org/2000/svg"">\n" >>SVGFILE;

printf "<rect x=""0"" y=""0"" width=""500"" height=""500"" " &
    "stroke=""none"" fill=""rgb(255,255,208)""/>\n" >>SVGFILE;

for {i in 0..9}
{
  printf "<line x1=""%g"" y1=""%g"" x2=""%g"" y2=""%g"" " &
      "stroke=""black"" stroke-width=""1""/>\n",
      25+50*i, 475, 25+50*i, 25 >>SVGFILE;
  printf "<line x1=""%g"" y1=""%g"" x2=""%g"" y2=""%g"" " &
      "stroke=""black"" stroke-width=""1""/>\n",
      25, 25+50*i, 475, 25+50*i >>SVGFILE;
}

for {a in Nodes, b in Nodes: use[a,b]}
{
  printf "<line x1=""%g"" y1=""%g"" x2=""%g"" y2=""%g"" " &
      "stroke=""blue"" stroke-width=""3""/>\n",
      PX[a], PY[a], PX[b], PY[b] >>SVGFILE;
}

for {n in Nodes: n!=Start}
{
  printf "<circle cx=""%g"" cy=""%g"" r=""%g"" " &
      "stroke=""black"" stroke-width=""1.5"" fill=""red""/>\n",
      PX[n], PY[n], 8 >>SVGFILE;
}

printf "<rect x=""%g"" y=""%g"" width=""16"" height=""16"" " &
    "stroke=""black"" stroke-width=""1.5"" fill=""green""/>\n",
    PX[Start]-8, PY[Start]-8 >>SVGFILE;

printf "</svg>\n" >>SVGFILE;

end;
```

7.6 MILP models – Summary

Several optimization problems where shown where MILP models are an adequate solution technique, while further capabilities of the GNU MathProg language were demonstrated.

- The knapsack and the multi-way number partitioning problem are easy examples for models involving discrete decisions and requiring integer variables.
- Tiling can also be addressed by MILP approaches, provided that our choices on tile placement are finite. This was shown on the tiling of an arbitrary rectangle with cross-shaped tiles. Some `set` and `param` definitions allowed a more concise model formulation.
- The assignment problem is a well-known optimization problem, and was implemented as an MILP model, turned out to be not more difficult than its LP relaxation. The functionality of making a priori decisions for a model was demonstrated on the assignment problem.
- Some problems on weighted graphs were also addressed. The shortest path problem and the minimum weight spanning tree problem were both formulated as an MILP model, with similar techniques. The idea was the management of imaginary material flows through the edges of the graph.
- Finally, the travelling salesman problem was solved as a combination of the assignment problem and the connectivity constraints from the minimum weight spanning tree problem. Visual output was also produced by the GNU MathProg code itself, in SVG format.

The examples shown here included some of the most common linear programming techniques and their possible implementations. These can be part of more complex real-world optimization problems. Integer programming techniques make a much wider range of problems to be solvable than pure LP models, but at a price of exploding computational complexity.

Chapter 8

Solution algorithms

So far, the LP and MILP modeling techniques and GNU MathProg implementation details were in focus. Mathematical programming models are solved by dedicated solvers. Usually the modeling procedure can be performed without knowing how the solvers work.

Nevertheless, an expertise in the solution algorithms themselves can be valuable when modeling. It may help designing models, improving their efficiency, choosing the appropriate solver tools and configurations, interpreting results and solver outputs. Therefore, this chapter is dedicated to give an insight into how the models we describe in a modeling language are solved in the background.

Operations research has a very rich literature from the seventies regarding solution algorithms, we propose some books for the interested readers. First of all, book of W. L. Winston [31] is a very commonly used handbook, considering the main models and algorithms. We recommend also the book of István Maros about Linear Programming [32], and the book of George Dantzig [33].

In fact, Linear Programming (LP) is a main technique within operations research. There are several solution methods, and within them we must mention the Simplex Method (and its different versions). The first version called Primal Simplex Method was introduced by George Dantzig in 1947. In the next chapter we briefly introduce it by solving a simple production problem.

8.1 The Primal Simplex Method

Given the next table with the data of a production problem.

	P_1	P_2	P_3	P_4	P_5	Cap
Res_1	1	2	1	3	0	24
Res_2	0	1	1	5	1	43
Res_3	1	0	0	2	2	18
Profit	19	23	15	42	33	

The meaning of the data is analogous to production problems mentioned before (see Problem 9 from Chapter 5).

We have a small factory where we may produce several products ($P_1, ..., P_5$). We have three resources for the production, a column belongs for each product, showing how many units from the resources must be used, respectively, for producing one unit of the corresponding product. For example, if we produce one unit from product P_4, we will use 3 units from resource Res_1, 5 units from resource Res_2, and 2 units from resource Res_3. A resource can be in fact some kind of raw material, or a human resource, electricity, etc. We can produce as many from the products as we want, except that the production will consume the capacities of the resources, and these capacities must not be exceeded. For example, if we produce 8 units from P_2 and 1 unit from P_4, then in total we will consume as many as $8 \cdot 2 + 1 \cdot 3 = 19$ units from the first resource. The usage from the other resources can be calculated similarly. We will gain certain amount of profit for the production, this is again a linear combination of the *production plan* and the vector of coefficients of the profit. If the production vector is as before, $x(0, 8, 0, 1, 0)$, introducing vector $c(19, 23, 15, 42, 33)$ for the profit, the gained profit would be $c \cdot x = (19, 23, 15, 42, 33) \cdot (0, 8, 0, 1, 0) = 226$. Our goal is to find a production plan where all constraints are satisfied and the gained profit is as large as possible. Denoting the matrix of the coefficients in the rows of resources and columns of products by A, the vector of capacities of resources by b, our problem can be written as below:

$$\begin{aligned} & z = c \cdot x \to \max \\ \text{st.} \quad & Ax \leq b, x \geq 0. \end{aligned}$$

We can see that this is a general format for LP models. We show how we can solve it by the Primal Simplex Method. First let us see the model in a detailed form:

$$\begin{array}{rrrrrl} x_1 & +2x_2 & +x_3 & +3x_4 & & \leq 24 \\ & +x_2 & +x_3 & +5x_4 & +x_5 & \leq 43 \\ x_1 & & & +2x_4 & +2x_5 & \leq 18 \\ & & & & x_i & \geq 0, 1 \leq i \leq 5 \\ 19x_1 & +23x_2 & +15x_3 & +42x_4 & +33x_5 & = z \to \max \end{array}$$

First we change the inequalities by adding so-called *slack variables* to the left hand side, gaining equations, like below.

$$\begin{array}{rrrrrrrrl} x_1 & +2x_2 & +x_3 & +3x_4 & & +s_1 & & & = 24 \\ & x_2 & +x_3 & +5x_4 & +x_5 & & +s_2 & & = 43 \\ x_1 & & & +2x_4 & +2x_5 & & & +s_3 & = 18 \\ & & & & & & & \mathbf{x} \geq \mathbf{0}, & \mathbf{s} \geq \mathbf{0} \\ 19x_1 & +23x_2 & +15x_3 & +42x_4 & +33x_5 & & & & = z \to \max \end{array}$$

In the above model we write $\mathbf{x} \geq \mathbf{0}$ instead of writing the nonnegativity constraints one by one for the variables but the meaning is the same. The same holds for the $\mathbf{s}$ vector. Instead of the system of equations we can use a brief form to handle the data. This brief form is called **simplex tableau**, which is shown below.

B	x_B	a_1	a_2	a_3	a_4	a_5	u_1	u_2	u_3
u_1	24	1	**2**	1	3	0	1	0	0
u_2	43	0	1	1	5	1	0	1	0
u_3	18	1	0	0	2	2	0	0	1
z	0	-19	-23	-15	-42	-33	0	0	0

We note that there are several versions of the simplex tableau; and all are equivalent. Here we use that version where the basis solution is put to the left hand side, and the extra row belonging to the objective is put in the bottom. These special choices have no importance in fact.

Here B means **basis**, which is a maximal linearly independent set of vectors. Now $B = \{u_1, u_2, u_3\}$, the basis is composed now (initially) from the three unit vectors. The corresponding **basis solution** is $x_B = (0, 0, 0, 0, 0, 24, 43, 18)$. This is a solution of the linear system, moreover each variable is nonnegative (i.e. the solution is feasible), and any nonzero component in the vector belongs to a basis vector.

The tableau contains all data, and it also has an extra row.

How do we define the extra row of the objective function? Here simply the negatives of the coefficients of the c vector appear.

The next theorem is crucial:

Theorem 1 *Exactly one of the following options happens.*
a, There is no negative entry in the last row. In this case the corresponding basis solution is optimal.
b, There is a negative value in the last row, such that there is no positive value in its column. In this case there is no optimal solution, as the objective value is not bounded from above.
c, Otherwise (there is a negative value in the last row, but for any such value there exists a positive value in its column) we can perform a basis transformation so that the corresponding value of the objective will not be smaller.

Since we give only a brief overview here about the simplex method, we do not prove the theorem, the interested reader can find it in any proposed book. But we show how the transformation goes. Each transformation exchanges one vector in the basis to another one not already there.

Let us suppose that we choose the column of a_2 for the vector that enters the basis. We are not allowed to choose the vector that will leave the basis, this choice must be made by a rule that is called the **minimum rule**. This is as follows. We take the following minimum: $\min\left\{\frac{24}{2}, \frac{43}{1}\right\}$, here the fractions are created so that the numerator is taken from the basis solution, and the denominator is taken from the same row and the chosen column. We cannot divide by zero, and we do not want to divide by negative value. The minimum is found as 24/2, this means that the vector of the row of 24 must be chosen as leaving vector. So u_1 leaves the basis. We call the 2 value as **pivot** number, we wrote it by bold letter in the tableau.

How do we perform the transformation? The row of the pivot value is divided by the pivot value (i.e. by 2), and 1/2 times of the row of the pivot value is subtracted from the second row, 0/2 times of the row of the pivot value is subtracted from the third row, and 23/2 times of the row of the pivot value is added to the row of the objective. In the above calculations (i.e. 1/2, 0/2, and 23/2) the numerator comes from the column of the pivot value, and the denominator is always the pivot value. The goal of choosing these factors is that by adding the pivot row to others, each element in the pivot column other than the pivot element is eliminated to zero. After the transformation we get the following tableau.

B	x_B	a_1	a_2	a_3	a_4	a_5	u_1	u_2	u_3
a_2	12	1/2	1	1/2	3/2	0	1/2	0	0
u_2	31	−1/2	0	1/2	7/2	1	−1/2	1	0
u_3	18	1	0	0	2	**2**	0	0	1
z	276	−15/2	0	−7/2	−15/2	−33	23/2	0	0

Now let us choose the column of a_5 to enter the basis. According to the minimum rule u_3 is the leaving vector. After the transformation we get:

B	x_B	a_1	a_2	a_3	a_4	a_5	u_1	u_2	u_3
a_2	12	1/2	1	**1/2**	3/2	0	1/2	0	0
u_2	22	−1	0	1/2	5/2	0	−1/2	1	−1/2
a_5	9	1/2	0	0	1	1	0	0	1/2
z	573	9	0	−7/2	51/2	0	23/2	0	33/2

Only one negative entry remained in the last row, we must choose this column to enter the basis. According to the minimum rule a_2 leaves the basis.

As we could realize, the rule of the change of the objective function is the following:

- If we choose a column to enter the basis where the value is negative in the row of the objective: the consequence is that the objective value is growing.
- If we choose a column where the value in the bottom is positive: the objective will decrease (but we do not want this as our goal is to maximize the objective)

- If we choose a column where the value in the bottom line is zero: there will be no change in the objective value.

Note that this rule about the choice of the entering column and the change of the objective is only true if the basis solution is non-degenerate, which means that if for example the current basis is $B(a_2, u_2, a_5)$, then all out of x_2, s_2, x_5 are positive. In fact, if the value of the variable in the basis solution in the row of the pivot value would be zero then there will be no change in the column of the basis solution, so there is no change in the objective.

Let us see what happened after the transformation:

B	x_B	a_1	a_2	a_3	a_4	a_5	u_1	u_2	u_3
a_3	24	1	2	1	3	0	1	0	0
u_2	10	$-3/2$	-1	0	1	0	-1	1	$-1/2$
a_5	9	$1/2$	0	0	1	1	0	0	$1/2$
z	657	$25/2$	7	0	36	0	15	0	$33/2$

We can see that there is no more negative entry in the last row which means (according to Theorem 1) that the present solution is optimal. This optimal solution is as follows: $x_B = (0, 0, 24, 0, 9\,|0, 10, 0\,)$. Here in the vector we put a vertical line to separate the first five variables from the last three variables (i.e. the slack variables), as their meanings are different. Now the solution means that in the optimal solution (if we get maximal amount of profit, so that all constraints are still satisfied) we need to produce 24 units and 9 units from the third and fifth products, respectively. Moreover, 10 units form the second resource remains, the whole amounts of the other two resources are completely consumed.

We can realize that we have performed three transformations, but we could also have reached the final tableau by only two transformations (as only a_3 and a_5 needed to be changed in the original basis). It is not easy to see in advance what kinds of transformations will lead to the final state in the fewest steps. But there are several tricks that can help. For example (for such a small problem) it can be beneficial if we choose such a vector to leave the basis, for which the increment of the objective is as much as possible.

There are other issues that can make the case difficult, for example very rarely "cycling" can happen, this means that after several transformations we return to some basis that we had before. This case is very rare in the practice, it almost never happens.

Another issue is that in many cases before we start to solve a problem, we can make some preprocessings, e.g. some constraints can be found that are redundant (which means that we loose nothing if we delete the constraint, the optimal solution of the remaining system will be the same).

What happens if the factory introduces a new (in fact a sixth) product, so that the data of its resource need is given by a_6? We can realize easily that there is no need to repeat the whole computation, we compute only the column of the new product (the transformed form of the a_6 vector can be written as $B^{-1}a_6$, where B is the present basis, and B^{-1} is its inverse), and we insert this column vector into the tableau. If the coefficient of this new vector in the last row (calculated as $c_B B^{-1} a_6 - c_6$) is nonnegative, it means that producing this new product is not advantageous. Otherwise this value is negative, and only this one is the negative value in the last row. Then we continue the transformations as we made it before. For an example, let us suppose that the column of the new product is $a_6(1, 2, 1)^T$, here the components mean the resource consumption from the three resources for producing one unit from this new product (say product P_6). Furthermore, let $c_6 = 35$ which means the profit gained from producing one unit from this product. Note that currently the basis is

$$B(a_3, u_2, a_5) = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 2 \end{bmatrix} \text{ and } B^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & -1/2 \\ 0 & 0 & 1/2 \end{bmatrix}.$$

We can find the inverse also in the last three columns of the simplex tableau. Thus $B^{-1}a_6 = \left(1, \frac{1}{2}, \frac{1}{2}\right)^*$, and $c_B B^{-1}a_6 - c_6 = (15,0,33)\cdot(1,1/2,1/2) - 35 = -7/2$, which means that the production of the new product is advantageous, and the procedure will continue, the vector representing the new product will enter the basis.

Finally we mention an article which treats certain different pivoting rules [34].

8.2 The Two-phase (Primal) Simplex Method

Here we introduce very briefly the two-phase method. What has been shown before, this is in fact the second phase of the method. So we need to show the first phase.

The purpose of the first phase is finding a feasible basis solution, while in the second phase, starting from the feasible (basis) solution, step by step we get an optimal solution. When do we need the first phase? In such a case when we are not able to begin with a suitable feasible solution. We show the first phase through an example. Let us suppose that our LP is as follows:

$$\begin{array}{rrrrrl} x_1 & +2x_2 & +x_3 & +3x_4 & & \geq 24 \\ & x_2 & +x_3 & +5x_4 & +x_5 & \leq 43 \\ x_1 & & & +2x_4 & +2x_5 & = 18 \\ & & & & x_i & \geq 0, 1 \leq i \leq 5 \\ 19x_1 & +23x_2 & +15x_3 & +42x_4 & +33x_5 & = z \to \max \end{array}$$

We modified our original example a little bit. It can be interpreted that from the first resource we need to use *at least* 24 units, we can use *at most* 43 units from the second resource (as before), and finally, we need to consume *all amounts* of the third resource.

First we convert the inequalities to equations.

$$\begin{array}{rrrrrrrl} x_1 & +2x_2 & +x_3 & +3x_4 & & -s_1 & & = 24 \\ & x_2 & +x_3 & +5x_4 & +x_5 & & +s_2 & = 43 \\ x_1 & & & +2x_4 & +2x_5 & & & = 18 \\ & & & & & & & \mathbf{x} \geq \mathbf{0}, \mathbf{s} \geq \mathbf{0} \\ 19x_1 & +23x_2 & +15x_3 & +42x_4 & +33x_5 & & & = z \to \max \end{array}$$

Now we can realize that we do not have enough unit vectors (that would serve us as an initial basis). We note that here s_2 is called slack variable and s_1 is called surplus variable. Let us create the missing unit vectors that we will need for a basis:

$$\begin{array}{rrrrrrrrrl} x_1 & +2x_2 & +x_3 & +3x_4 & & -s_1 & & +t_1 & & = 24 \\ & x_2 & +x_3 & +5x_4 & +x_5 & & +s_2 & & & = 43 \\ x_1 & & & +2x_4 & +2x_5 & & & & +t_2 & = 18 \\ & & & & & & & & & \mathbf{x} \geq \mathbf{0}, \mathbf{s} \geq \mathbf{0}, \mathbf{t} \geq \mathbf{0} \\ 19x_1 & +23x_2 & +15x_3 & +42x_4 & +33x_5 & & & & & = z \to \max \end{array}$$

We call t_1 and t_2 as artificial variables, as we will use them but we „are not allowed” to use them. It is true that the original equation system has a feasible solution (which means that all equations are satisfied with the bounded $\mathbf{x}$ and $\mathbf{s}$ variables) if and only if we can eliminate the artificial variables from the system. So our goal is to eliminate them, this will be the task of the first phase. Our technique is to introduce an artificial (or secondary) objective function as follows, let $\hat{z} = -t_1 - t_2$. Instead of the original objective function z, we will maximize $\hat{z}$ in the first phase (in the same way as we have shown how we maximize z). If the maximum value of $\hat{z}$ is zero, it means that we already eliminated the artificial variables, and continue with the second phase, where we return to the original objective function. Otherwise, if the maximum of $\hat{z}$ is negative, it means that the original LP cannot be solved without the artificial variables, i.e. does not have a feasible solution.

8.3 The Dual Simplex Method

There are several versions of the simplex method, besides the Primal Simplex Method, the Dual Simplex Method is the other main version. We introduce it briefly. Let us consider the following LP:

$$\begin{array}{rrrl}
x_1 & & +x_3 & \geq 1 \\
2x_1 & +x_2 & +3x_3 & \geq 3 \\
x_1 & +2x_2 & & \geq 5 \\
4x_1 & +2x_2 & +x_3 & \geq 7 \\
 & & \mathbf{x} & \geq \mathbf{0} \\
10x_1 & +12x_2 & +15x_3 & = z \to \min
\end{array}$$

After multiplying all inequalities and the objective function by -1, and adding slack variables we get the following system:

$$\begin{array}{rrrrrrrl}
-x_1 & & -x_3 & +s_1 & & & & = -1 \\
-2x_1 & -x_2 & -3x_3 & & +s_2 & & & = -3 \\
-x_1 & -2x_2 & & & & +s_3 & & = -5 \\
-4x_1 & -2x_2 & -x_3 & & & & +s_4 & = -7 \\
 & & & & & & \mathbf{x} & \geq \mathbf{0} \\
-10x_1 & -12x_2 & -15x_3 & & & & & = -z \to \max
\end{array}$$

And here is the first simplex tableau below:

B	x_B	a_1	a_2	a_3	u_4	u_5	u_3	u_4
u_1	-1	$\mathbf{-1}$	0	-1	1	0	0	0
u_2	-3	-2	-1	-3	0	1	0	0
u_3	-5	-1	-2	0	0	0	1	0
u_4	-7	-4	-2	-1	0	0	0	1
$-z$	0	10	12	15	0	0	0	0

For the first sight we can realize that now the simplex tableau is different from the previous form. In the last row there are only nonnegative values. We will call this property that the basis solution is *dual-feasible*. But now the basis solution is not primal feasible, i.e. there are negative values in the basis solution. In fact, all values are negative or zero since now this is the basis solution: $x_B = (0, 0, 0, -1, -3, -5, -7)$. If we can reach that the tableau is both dual and primal feasible after several transformations, this means that we will get the optimal tableau. Note that in the Primal Simplex Method (in the second phase of it) we go through primal feasible solutions, while in the Dual Simplex Method we go through dual feasible solutions.

Here is the rule of the transformation:

- We choose a row where the coordinate of the basis solution is negative (this vector of the basis will leave the basis).
- From this row we will choose a negative pivot value.
- We apply an appropriate version of the minimum rule for choosing the entering column.

Let us suppose that the leaving vector is u_1, so we choose the pivot value from the first row. We can choose the entering column from a_1, a_2 and a_3 as all other vectors are in the basis. The appropriate values in this row and in these columns are -1, 0 and again -1. As we told, we would choose a negative value, so we cannot choose the zero. Regarding the two other values we make the following calculation $\min\{10/1, 15/1\}$. Here the numerators come from the row of the objective, and the denominators are the negatives of the previously listed two -1 values. Since $10/1$ is smaller, the appropriate value is chosen as pivot value, we denoted it by bold letter in the tableau. After the transformation (where the rule is the same as in the transformation for the primal simplex method), we get the following tableau:

B	x_B	a_1	a_2	a_3	u_4	u_5	u_3	u_4
a_1	1	1	0	1	-1	0	0	0
u_2	-1	0	-1	-1	-2	1	0	0
u_3	-4	0	$\mathbf{-2}$	1	-1	0	1	0
u_4	-3	0	-2	3	-4	0	0	1
$-z$	-10	0	12	5	10	0	0	0

Now let us choose u_3 as the leaving vector. There are only two negative values in its row that can be considered as pivot value. The calculation is the following: $\min\{12/2, 10/1\}$. Since the former fraction is the smaller, -2 is chosen as the pivot value. After the transformation the next tableau is the following:

B	x_B	a_1	a_2	a_3	u_4	u_5	u_3	u_4
a_1	1	1	0	1	-1	0	0	0
u_2	1	0	0	$-3/2$	$-3/2$	1	$-1/2$	0
a_2	2	0	1	$-1/2$	$1/2$	0	$-1/2$	0
u_4	1	0	0	2	-3	0	-1	1
$-z$	-34	0	0	11	4	0	6	0

Since we got a tableau with a solution that is both primal and dual feasible, this is an optimal solution. Here $x_B = (1, 2, 0\,|, 0, 1, 0, 1\,)$ and the (optimal) value of the objective is $z = 34$.

8.4 The Gomory Cut

The Gomory cut is a main tool for getting integer solutions for an LP. If we require that the variables of an LP are in fact integer numbers, we call the model Integer Linear Program (ILP). In many cases some of the variables are required as integers but the other are allowed to be real values, in this case the program is called Mixed-Integer Linear Program (MILP).

Solving an integer (or mixed-integer) program is (usually) much harder. There are several tools to handle the integrality of the variables, we show only one such method, which is commonly used. We will show the method through an example. The example comes from a book of András Prékopa, as it is mentioned in the manuscript of Tamás Szántai [35], page 28 (in Hungarian), and it is also shown in the short (Hungarian) draft of Mihály Hujter [36]. We note that we perform the calculations in a little bit different way, as we tried to make our calculations uniform (for the Primal Simplex and Dual Simplex tableau).

Let us consider the next ILP:

$$\begin{aligned}
2x_1 \; +x_2 &\geq 1\\
2x_1 \; +5x_2 &\geq 4\\
-x_1 \; +x_2 &\geq 0\\
-x_1 \; -x_2 &\geq -5\\
-x_1 \; -2x_2 &\geq -4\\
\mathbf{x} &\geq \mathbf{0},\ x_1, x_2 \text{ are integers}\\
5x_1 \; +4x_2 &= z \rightarrow \min
\end{aligned}$$

Let us transform the inequalities to „$\leq$” type, then form them to equations by introducing slack variables. Also, let us transform the objective to „max” form. We get the following.

$$\begin{array}{rrrrrrrl}
-2x_1 & -x_2 & +s_1 & & & & & = -1 \\
-2x_1 & -5x_2 & & +s_2 & & & & = -4 \\
x_1 & -x_2 & & & +s_3 & & & = 0 \\
x_1 & +x_2 & & & & +s_4 & & = 5 \\
x_1 & +2x_2 & & & & & +s_5 & = 4 \\
& & & & & & & \mathbf{x} \geq \mathbf{0} \\
-5x_1 & -4x_2 & & & & & & = -z \to \max
\end{array}$$

Now let us create the usual simplex tableau:

B	x_B	a_1	a_2	u_1	u_2	u_3	u_4	u_5
u_1	-1	-2	-1	1	0	0	0	0
u_2	-4	-2	-5	0	1	0	0	0
u_3	0	1	-1	0	0	1	0	0
u_4	5	1	1	0	0	0	1	0
u_5	4	1	2	0	0	0	0	1
$-z$	0	5	4	0	0	0	0	0

The tableau is dual feasible, but primal infeasible. Let us make basis transformations with the Dual Simplex Method to reach the optimal solution. In fact, two steps will be enough. First u_1 leaves the basis and a_1 enters (according to the minimum rule), then u_2 leaves the basis and a_2 enters. The resulting tableau is as follows:

B	x_B	a_1	a_2	u_1	u_2	u_3	u_4	u_5
a_1	$1/8$	1	0	$-5/8$	$1/8$	0	0	0
a_2	$3/4$	0	1	$1/4$	$-1/4$	0	0	0
u_3	$5/8$	0	0	$7/8$	$-3/8$	1	0	0
u_4	$33/8$	0	0	$3/8$	$1/8$	0	1	0
u_5	$19/8$	0	0	$1/8$	$3/8$	0	0	1
$-z$	$-29/8$	0	0	$17/8$	$3/8$	0	0	0

The given tableau is optimal. The optimal (fractional) solution is the following:

$$x_B = (1/8, 3/4 \,|\, 0, 0, 5/8, 33/8, 19/8)$$

This is the optimal solution of the *relaxed* model, where $\mathbf{x} \geq \mathbf{0}$ is still required, but the integrality of the variables is not required. The optimum value of the relaxed problem is $z = 29/8$.

At this point we will perform the famous *Gomory Cut*!

Let us choose a fractional variable, e.g. $x_1 = 1/8$. Let us consider the row of this variable in the tableau, as well. Here the meaning of the data (provided by the tableau) is the following:

$$1/8 = x_1 - 5/8u_1 + 1/8u_2$$

We round up all coefficients in this equation. The values (to round up $1/8$, 1, $-5/8$ and $1/8$) are the following: $7/8$, 0, $5/8$ and $7/8$, respectively. With these values (using them one by one and omitting the zero) we create the next (new) constraint:

$$7/8 \leq 5/8u_1 + 7/8u_2$$

We can realize that this condition does not hold, since $u_1 = u_2 = 0$ in the current optimal (fractional) basis solution. It means that adding this last constraint to the previously used constraints, the current basis solution becomes infeasible. Let us multiply the new constraint by -1 and introduce a new slack variable denoted by s_6, we get:

$$-5/8u_1 - 7/8u_2 + s_6 = -7/8$$

We add this new equation to the system, and write it into a new line into the simplex tableau as follows:

B	x_B	a_1	a_2	u_1	u_2	u_3	u_4	u_5	u_6
a_1	$1/8$	1	0	$-5/8$	$1/8$	0	0	0	0
a_2	$3/4$	0	1	$1/4$	$-1/4$	0	0	0	0
u_3	$5/8$	0	0	$7/8$	$-3/8$	1	0	0	0
u_4	$33/8$	0	0	$3/8$	$1/8$	0	1	0	0
u_5	$19/8$	0	0	$1/8$	$3/8$	0	0	1	0
u_6	$-7/8$	0	0	$-5/8$	$\mathbf{-7/8}$	0	0	0	1
$-z$	$-29/8$	0	0	$17/8$	$3/8$	0	0	0	0

Now the tableau is still dual feasible but not primal feasible. We choose the row of u_6 for the leaving vector. According to the minimum rule the pivot value is $-7/8$ (denoted by bold), since $3/7 < 17/5$. After the transformation the next tableau is as follows:

B	x_B	a_1	a_2	u_1	u_2	u_3	u_4	u_5	u_6
a_1	0	1	0	$-5/7$	0	0	0	0	$1/7$
a_2	1	0	1	$3/7$	0	0	0	0	$-2/7$
u_3	1	0	0	$8/7$	0	1	0	0	$-3/7$
u_4	4	0	0	$2/7$	0	0	1	0	$1/7$
u_5	2	0	0	$-1/7$	0	0	0	1	$3/7$
u_2	1	0	0	$5/7$	1	0	0	0	$-8/7$
$-z$	-4	0	0	$13/7$	0	0	0	0	$3/7$

We got an optimal tableau again (primal feasible and dual feasible, so it is optimal). But here the optimal solutions are as follows: $x_B = (0, 1\,|0, 1, 1, 4, 2)$. Since $x_1 = 0$ and $x_2 = 1$ are integer values, this means that we got the optimal solution of the ILP, as well.

Chapter 9

Summary

Capabilities of the GNU MathProg modeling language was presented, on various problems. Linear equation system solution, the production and the diet problem and their common extensions, the transportation problem, various cost functions in general, and integer programming techniques through common optimization problems were involved.

Mathematical programming in general offers a simple practical solution for the class of problems it is adequate for. MILP models can be used for a much wider range of cases than their pure LP counterparts. Nevertheless both problem classes are useful on their own.

Using our expertise in GNU MathProg gives us a unique tool for hard optimization problems. The solution speed is not necessarily the best available for a particular problem, but ease of implementation, maintenance of code and adaptation to changes in the problem definition makes the methodology valuable for both industrial and scientific purposes.

Usually, a single model file is developed, which addresses all problem instances implemented in their own data files. The `glpsol` solver is capable of parsing the language and solving the model at the same time, and user-defined output can also be obtained.

We also gave an insight into how LP models can be solved by the Simplex Method, and one technique for integer programming problems, the Gomory cut. These are only an introduction to how LP/MILP solver software work in the background. We are allowed to treat solvers as black boxes, but basic knowledge about them can be useful for developing mathematical programming models, improving them and better understanding their results.

Note that there are many approaches other than the main line shown in this Tutorial. We mention a few as an ending.

- The language GNU MathProg and a parser/solver like `glpsol` can be used alone to develop and solve models, but note that the GLPK software kit offers other features, notably a callable library. Accessing linear programming tools from a programming language can be better in the long term than using standalone GNU MathProg model files. For example, in GNU MathProg, we cannot „change" the value of a parameter, which limits input data processing possibilities.

- Different configurations of the solver `glpsol` are only barely touched in this Tutorial. Application of the appropriate heuristics or alternative solution methods can speed up the search.

- The solver `glpsol` is an easy-to-use tool, but probably not the fastest LP/MILP solver. Other free solvers may be superior if performance is an issue, for example CBC [7] or lpsolve [8]. Commercial MILP solvers can be even much better. We can use alternative solvers with GNU MathProg models, as `glpsol` supports exporting a model into well-known formats.

- GNU MathProg is not the only language for linear programming. There are dozens of other languages, each having an own class of models, input/output formats and solvers to support.

- Not LP and MILP are the only mathematical programming problem classes having general purpose solvers. If specific nonlinear objectives and constraints are needed to model a situation, then we might try developing a Nonlinear or Mixed-Integer Nonlinear Programming (NLP or MINLP) model in some adequate environment. Remember, more general tools can be much more costly in terms of running time.
- Mathematical programming is a powerful tool, but some problems do have much more effective algorithmic solutions. Sometimes developing an algorithm can yield better results, although often for the cost of more coding.
- Throughout this Tutorial, we only solved models in a whole, resulting in a final answer. In general, mathematical programming tools can be used as part of some algorithmic framework. For example, a large problem can be decomposed into several different models that are solved separately or one after the other. Or, an LP/MILP model can serve as a relaxation for a more complex optimization problem, and as such, provide a bound for its objective.

We hope the reader will find this Tutorial helpful and motivating in solving real-life optimization problems in the future.

Bibliography

[1] GNU MathProg (GMPL), reference manual, gusek.sourceforge.net/gmpl.pdf

[2] GNU Linear Programming Kit (GLPK), gnu.org/software/glpk

[3] GLPK Wikibooks, en.wikibooks.org/wiki/GLPK

[4] IMOLS by Máté Hegyháti, hegyhati.github.io/IMOLS

[5] Tankönyvtár homepage, dtk.tankonyvtar.hu/

[6] General Public License 3.0, gnu.org/licenses/gpl-3.0.en.html

[7] Coin-OR Branch and Cut (CBC), LP/MILP solver, projects.coin-or.org/Cbc

[8] lpsolve, LP/MILP solver, lpsolve.sourceforge.net/5.5

[9] GUSEK, Windows IDE for GLPK, gusek.sourceforge.net/gusek.html

[10] Gaussian elimination, en.wikipedia.org/wiki/Gaussian_elimination

[11] Solver tutorial, Product Mix, solver.com/solver-tutorial-solver-model-using-product-mix-example

[12] LINGO tutorial, Product Mix, lindo.com/downloads/LINGO_text/Chapter6.pdf

[13] George J. Stigler, The Cost of Subsistence, Journal of Farm Economics (1945) 27(2), 303-314, doi.org/10.2307/1231810

[14] Stigler Diet problem, en.wikipedia.org/wiki/Stigler_diet

[15] L. R. Ford, Jr. and D. R. Fulkerson, Solving the Transportation Problem, Management Science (1956) 3(1) 24-32, jstor.org/stable/2627172

[16] Short tutorial on the transportation problem, mathcs.emory.edu/~cheung/Courses/323/Syllabus/Transportation/intro.html

[17] Goal programming, en.wikipedia.org/wiki/Goal_programming

[18] Knapsack problem, en.wikipedia.org/wiki/Knapsack_problem

[19] Richard E. Korf, Multi-Way Number Partitioning Problem, Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI-09), ISBN 978-1-57735-426-0, 2009, ijcai.org/Proceedings/09/Papers/096.pdf

[20] Bin packing problem, en.wikipedia.org/wiki/Bin_packing_problem

[21] Assignment problem, en.wikipedia.org/wiki/Assignment_problem

[22] Harold W. Kuhn, The Hungarian Method for the assignment problem, Naval Research Logistics Quarterly (1955) 2, 83–97, doi.org/10.1002/nav.3800020109

[23] John Adrian Bondy, Uppaluri Siva Ramachandra Murty, Graph Theory with Applications, Macmillan, 1976, ISBN 0333177916

[24] Shortest path problem, en.wikipedia.org/wiki/Shortest_path_problem

[25] Minimum weight spanning tree problem, en.wikipedia.org/wiki/Minimum_spanning_tree

[26] Dijkstra's algorithm explained,
math.mit.edu/~rothvoss/18.304.3PM/Presentations/1-Melissa.pdf

[27] Kruskal's algorithm explained, tutorialspoint.com/data_structures_algorithms/kruskals_spanning_tree_algorithm.htm

[28] Travelling salesman problem, en.wikipedia.org/wiki/Travelling_salesman_problem

[29] Scalable Vector Graphics (SVG) 2, w3.org/TR/SVG2

[30] Manual SVG output by GLPK, en.wikibooks.org/wiki/GLPK/Scalable_Vector_Graphics

[31] Wayne L. Winston, Operations Research, Applications and algorithms, Duxbury Press, Belmont, California, 1991 (second Edition), ISBN 0-534-92495-6

[32] István Maros, Computational Techniques of the Simplex Method, Series Title: International Series in Operations Research & Management Science, Series Volume: 61, 325 pages, Hardcover ISBN 978-1-4020-7332-8, 2003

[33] G. B. Dantzig, Maximization of linear functions of variables subject to linear inequalities, in: Activity Analysis of Production and Allocation, ed. T. C. Koopmans, John Wiley and Sons, Inc., New York, 1951.

[34] Terlaky, T. & Zhang, Pivot rules for linear programming: A survey on recent theoretical developments, Ann Oper Res (1993) 46(1), 203-233, doi.org/10.1007/BF02096264

[35] OR tutorial (Hungarian), math.bme.hu/~hujter/mmopsztl.pdf

[36] Gomory cuts example (Hungarian), math.bme.hu/~hujter/gomory.pdf