Practical Mathematical Models of Optimization Problems

Using Google OR-Tools

Serge Kruk

January 5, 2016

Contents

Contents **i**

1 Introduction **1**
- 1.1 What is this book about? . . . 2
- 1.2 Features of the text . . . 3
- 1.3 Getting our feet wet: Amphibian coexistence . . . 4

2 Linear continuous models **11**
- 2.1 Mixing . . . 13
- 2.2 Blending . . . 19
- 2.3 Project management . . . 24
- 2.4 Multi-stage models . . . 28

3 Hidden linear continuous models **35**
- 3.1 Piecewise linear . . . 37
- 3.2 Curve fitting . . . 44

4 Linear pseudo discrete models **49**
- 4.1 Maximum flow . . . 51
- 4.2 Minimum cost flow . . . 55
- 4.3 Transshipment . . . 59
- 4.4 Shortest paths . . . 63

5 Linear pure discrete models **71**
- 5.1 Minimum set cover . . . 73
- 5.2 Set packing . . . 77
- 5.3 Bin packing . . . 79
- 5.4 TSP . . . 86

6 Mixed models **93**
- 6.1 Facility location . . . 95
- 6.2 Multi-commodity flow . . . 99
- 6.3 Staffing level . . . 104
- 6.4 Cutting stock . . . 108
- 6.5 Non-convex trickery . . . 116

6.6 Staff scheduling 127
6.7 Sports timetabling 133
6.8 Puzzles 140
Quick reference for OR-Tools MPSolver in Python 151

Chapter 1

Introduction

1.1 What is this book about?

This book is an introduction to the art and science of implementing *mathematical models* of *optimization problems*.

An optimization problem is almost any problem that is, or can be, formulated as a question starting with "What is the best ..."? For instance:

- What is the best route to get from home to work?
- What is the best way to produce cars to maximize profit?
- What is the best way to carry groceries home, paper or plastic?
- Which is the best school for my kid?
- Which is the best fuel to use in rocket boosters?
- What is the best placement of transistors on a chip?
- What is the best NBA schedule?

These questions are rather vague and can be interpreted in a multitude of ways. Consider the first: By best do we mean fastest, shortest, most pleasant to ride, least bumpy or least fuel-guzzling? Besides, the question is incomplete. Are we walking, riding, driving or snowboarding? Are we alone or accompanied by a screaming two year-old?

To help us formulate solutions to optimization problems, optimizers[1] have established a frame into which we mold the questions; we call that a model. The most crucial aspect of a model is that it has an objective and it has constraints. Roughly, the objective is what we want; the constraints, what obstacles are on our way. If we can reformulate the question to clearly identify both the objective and the constraints, we are closer to a model.

Let us consider in more detail the "best route" problem but with an eye to clarify objective and constraints. We could formulate it as:

> Given a map of the city, my home address and the address of the daycare of my two-year-old son, what is the best route to take on my bike to bring him to daycare as fast as possible?

The goal is to find, among all the solutions that satisfy the requirements, that is, they are paths following either streets or bike lanes (the constraints), one path that minimizes the time it takes to get there (the objective).

Objectives are always quantities we want to maximize or minimize (time, distance, money, surface area, ...). Though we will see examples where we want to maximize something and minimize something else; this is easily accommodated. Sometimes there are no objectives. We say that the problem is one of feasibility, i.e. we are looking for any solution satisfying the requirements. From the point of view of the modeler, the difference is minimal. Especially since, in most practical cases, a feasibility model is usually a first step. After noticing a solution one usually wants to optimize something and the model is modified to include an objective function.

[1] I will use the term "optimizers" to name the mathematicians, theoreticians and practioners, who, since the nineteen-fiftees, have worked in the fields of Linear Programming (LP) and Integer Programming (IP). There are other who could make valid claims to the moniker, chefly among them researchers in Constraint Programmming, but my focus will be mostly in LP and IP models, hence my restricted definition.

1.2 Features of the text

As this text is an introduction, I do not expect the reader to be already well versed in the art of modeling. I will start at the beginning, assuming only that the reader understands what is a variable (both in the mathematical sense and in the programming sense), an equation, an inequality and a function. I will also assume that the reader knows some programming language, preferably Python, though knowing any other imperative language is enough to be able to read the Python code displayed in the text.

Note that **the code in this book is an essential component**. To get the full value, the reader must, slowly and attentively, read the code. This book is not a text of recipes described from a birds-eye view, using mathematical notation, with all the nitty-gritty details “left as an exercise for the reader”. This is implemented, functional, tested, optimization code that the reader can use and is encouraged to modify to fully understand. The mathematics in the book has been reviewed by mathematicians, like any mathematical paper. But the code has been subjected to a much more stringent set of reviewers with names Intel, AMD, Motorola, IBM. [2]

The book is the fruit of decades of consulting and of years teaching both an introductory modeling class (MOR242 Intro to Operation Research Models) and a graduate class (APM568 Mathematical Modeling in Industry) at Oakland University. I start at the undergraduate level and proceed up to the graduate level in terms of modeling itself, without delving much into the attendant theory.

- Every model is expressed in Python using Google or-tools[3] and can be executed as stated. In fact, the code presented in the book is automatically extracted, executed and the output inserted into the text without manual intervention; even the graphs are produced automatically (thanks to Emacs [4] and org-mode [5]).
- My intention is to help the reader become a proficient modeler, not a theoretician, therefore little of the fascinating mathematical theory related to optimization is covered; it is nevertheless used, and used profitably, to create simple yet efficient models.
- The associated web site provides all the code presented in the book along with a random generator for many of the problems and variations. The author uses this as a personalized homework generator. It can also be used as a self-guided learning tool.

http://www.practicalopt.com/

1.2.1 A note on notation

Throughout the book, I will describe algebraic models. These models can be represented in a number of ways. I will use two. I will sketch each model using common mathematical notation typeset with TEX in math mode. I will then express the complete, detailed model in executable Python code. The reader should have no problem seeing the equivalence between the formulations. Table 1.1 illustrates some of the equivalencies.

[2] My doctoral advisor used to say “There are error-free mathematical papers”. But we only have found an existence proof of that theorem. I will not claim that the code is error-free, but I am certain that it has fewer errors than any mathematical paper I ever wrote.

[3] https://developers.google.com/optimization/library

[4] The one and only editor: http://www.gnu.org/software/emacs/

[5] http://orgmode.org/

Table 1.1: Equivalence of expression in Math and Python modes.

Object	Math mode	Python mode
Scalar Variable	x	`x`
Vector	v_i	`v[i]`
Matrix	M_{ij}	`M[i][j]`
Inequality	$x + y \leq 10$	`x+y <= 10`
Summation	$\sum_{i=0}^{9} x_i$	`sum(x[i] for i in range(10))`
Set definition	$\{i^2 \mid i \in [0, 1, \ldots, 9]\}$	`[i**2 for i in range(10)]`

1.3 Getting our feet wet: Amphibian coexistence

The simplest problems are similar to those first encountered in high-school (the dreaded word problems). They are algebraic in nature; that is, can be formulated and sometimes solved using the simple tools of elementary linear algebra. We consider here one such problem to illustrate our approach to modeling and define some fundamental concepts.

A zoo biologist will place three species of amphibians (a toad, a salamander and a caecilian) in an aquarium where they will feed on three different small preys, worms, crickets and flies. Each day 1500 worms, 3000 crickets and 5500 flies will be placed in the aquarium. Each amphibian consumes a certain number of preys per day. The Table 1.2 summarizes the relevant data.

Table 1.2: Number of preys consumed by each specie of amphibian.

Food	Toad	Salamander	Caecilian	Available
Worms	2	1	1	1500
Crickets	1	3	2	3000
Flies	1	2	3	5000

The biologist wants to know how many amphibian, up to 1000 of each specie, can coexist in the aquarium assuming that food is the only relevant constraint.

How to we model this problem? All optimization and feasibility problems in this book are modeled using a three step approach. We will expand on this approach as we encounter problems on increasing complexity but the fundamental three steps remain the cornerstone of a good model.

1. **Identify the question to answer.** This identification should take the form of a precise sentence involving either counting or valuating one or more objects. In this case: "How many amphibians each specie can coexist in the aquarium?". Notice that "How many amphibians?" would not not be precise enough because we are not interested in the total count, but rather in the count of each specie. Formulating a precise question is often the hardest part.

   Once we have this precise question, we assign a variable to each of the objects to count. We will use x_0, x_1 and x_2. These are traditionally known as *decision variables*. The expression is a misnomer in our first example but reflects the origins of optimization problems in logistics where the decision variables were indeed representative of quantities under the control of the modeler and mapped to planning decisions.

2. **Identify all requirement and translate them into constraints.** The constraints, as we will see throughout the book, can take on a multitude of forms. In this simple

problem, they are algebraic, linear inequalities. It is often best to write down each requirement in a precise sentence before translating it into a constraint. For the coexistence case, the requirements, in words, are:

- All amphibians combined consume 1500 worms.
- All amphibians combined consume 3000 crickets.
- All amphibians combined consume 5000 flies.

Note that a statement starting with "The amount of ..." may not be precise enough. In our simple case, there are no specified units but there could be. For instance, the amount consumed could be stated in grams while the availability is in kilograms. This happens often and is the cause of many a model going awry.

Yet, even with our seemingly precise statements, there is an ambiguity left to consider. It is one of the main contribution of a good modeler to highlight ambiguity and clarify problem statements. Here, do we mean that the amphibians will consume exactly the amounts stated, or that they will consume at most the amounts stated[6]? We will assume that "at most" is the proper form of the requirement, both because it is more interesting and, in a sense, subsumes the "equal" question. We will then translate these requirements into algebraic constraints based on our decision variables.

Let us consider worms. The toads eat two per day. The salamanders and caecilians each eat one. Since we decided on x_0 toads, x_1 salamanders and x_2 caecilians, the total number of worms consumed will be bounded by the following inequality:

$$2x_0 + x_1 + x_2 \leq 1500. \tag{1.1}$$

Had we decided that "equal to" was the proper constraint, we would replace the inequality by an equality.

Consider now crickets. Toads consume one per day while salamanders consume three and caeciliens consumes two . They will collectively consume $x_0 + 3x_1 + 2x_2$ and we obtain the constraint

$$x_0 + 3x_1 + 2x_2 \leq 3000. \tag{1.2}$$

The constraint on flies is obtained similarly to produce

$$x_0 + 2x_1 + 3x_2 \leq 5000. \tag{1.3}$$

3. **Identify the objective to optimize.** The objective is, in the case of an optimization problem, what we want to maximize (or minimize). In the case of a feasibility problem, there is no objective but in practice, most feasibility problems are really optimization problems that have been incompletely formulated.

   Since the problem is stated as "How many amphibians of each specie can coexist?", a possible, even likely reading is that we want the maximum number of amphibians. (The minimum number is zero and is an example of the uninteresting *trivial* solution.) In terms of our decision variables, we want to maximize the sum and obtain

$$\max x_0 + x_1 + x_2. \tag{1.4}$$

[6]This seemingly trivial change, from "exactly equal" to "at most" represents more than 2000 years of mathematical development in solution techniques. We have known how to solve the "equal" form since ancient Babylonians (though it is known today as "Gaussian elimination") and we teach it in high school, but we only discovered how to solve the "at most" form in the twentieth century.

At this point we have a model! Not *the* model, but *a* model; a simple, clear and precise algebraic model that has a solution, one which answers our original question.

Since we are not mere theoreticians uninterested in practical applications, our next step is to solve the model. As we will do for every model in this book, we need to translate the mathematical expressions above ((1.1)-(1.4)) into a form digestible by one of the many solvers available.

Over the years, optimizers have developed a number of specialized modeling languages and solvers. Here is a short list of the better known ones:

- Modeling languages
  - AMPL (`www.ampl.com`)
  - GAMS (`gams.com`)
  - GMPL (`http://en.wikibooks.org/wiki/GLPK/GMPL_(MathProg)`)
  - Minizinc (`http://www.minizinc.org/`)
  - OPL (`http://www-01.ibm.com/software/info/ilog/`)
  - ZIMPL (`http://zimpl.zib.de/`)
- Solvers
  - CBC (`http://www.coin-or.org/`)
  - CLP (`http://www.coin-or.org/Clp/`)
  - CPLEX (`http://www-01.ibm.com/software/info/ilog/`)
  - ECLiPSe (`http://eclipseclp.org/`)
  - Gecode (`http://www.gecode.org/`)
  - GLOP (`https://developers.google.com/optimization/lp/glop`)
  - GLPK (`http://www.gnu.org/software/glpk/`)
  - Gurobi (`http://www.gurobi.com/`)
  - SCIP (`http://scip.zib.de/`)

We should maintain a distinction between *modeling languages*, formal constructions with specific vocabulary and grammars; and *solvers*, software packages that can read in models expressed in certain languages and write out the solutions. Though in some cases, this distinction is blurry.

As a modeler, one creates a model (in language X) which is then fed to a solver (solver Y). This can happen because solver Y knows how to parse language X or because there is a translator between language X and another language, say Z, which the solver understands. This, over the years, has been the cause of much irritation ("What? You mean that I have to rewrite my model to use your solver?").

To make matters worse, these languages and solvers are not equivalent. Each has its strengths and weaknesses, its areas of specialization. After years of writing models in all the languages above, and then some, my preference today is to eschew specialized languages and to use a general purpose programming language, for instance Python, along with a library interfacing with multiple solvers. Throughout this book I will use Google's Operations Research Tools (or-tools), a very well structured and easy to use library.

The or-tools library is comprehensive. It offers the best interface I have ever used to access multiple linear and integer solvers (MPSolver). It also has special-purpose code for network flow problems as well as a very effective constraint programming library. In this text, I will display only a very small fraction of this cornucopia of optimization tools.

One of the many advantages of using a general purpose language like Python is that we can not only do the modeling part, but also the insertion of the models into a larger application, maybe a web or a phone app. We can also easily present the solutions in a clear format. We have all the power of a complete language at our disposal. True, the specialized modeling languages sometimes allow more concise model expression. But, in my experience they all, at one point or another, hit a wall forcing the modeler to write kludgy glue to connect a model to the rest of the application. Moreover, writing or-tools models in Python can be such a joy[7]. The whole coexistence model is shown at Code 1.1.

Code 1.1: Amphibian coexistence model.

```
1  from linear_solver import pywraplp
2  def solve_coexistence():
3    t = 'Amphibian␣coexistence'
4    s = pywraplp.Solver(t,pywraplp.Solver.GLOP_LINEAR_PROGRAMMING)
5    x = [s.NumVar(0, 1000,'x[%i]' % i) for i in range(3)]
6    pop = s.NumVar(0,3000,'pop')
7    s.Add(2*x[0] + x[1] + x[2] <= 1500)
8    s.Add(x[0] + 3*x[1] + 2*x[2] <= 3000)
9    s.Add(x[0] + 2*x[1] + 3*x[2] <= 4000)
10   s.Add(pop == x[0] + x[1] + x[2])
11   s.Maximize(pop)
12   s.Solve()
13   return pop.SolutionValue(),[e.SolutionValue() for e in x]
```

Let us deconstruct the code. Line 1 loads the Python wrapper of the Linear Programming subset of or-tools. Every model we write will start this way. Line 4 names and creates a linear programming solver (hereafter named `s`), using Google's own[8] GLOP The or-tools library has interfaces to a number of solvers. Switching to a different solver, say GNU's[9] GLPK or Coin-or[10] CLP is a simple matter or modifying this line.

On line 5, we create a one-dimensional array `x` of three decision variables that can take on values between 0 and 1000. The lower bound is a physical constraint since cannot have a negative number of amphibians. The upper bound is part of the problem statement as the biologist will not put more than 1000 of each specie in the test tube. It is possible to state ranges as any contiguous subsets of $(-\infty, +\infty)$, but, as a general rule of thumb, restricting the range as much as possible during variable declaration tends to help solvers run efficiently. The third parameter of the call to `NumVar` is used as the name to print if and when this variable is displayed, for instance, in debugging a model. We will have little use for this feature as we prefer to write bug-free models.

The constraints on lines 7 to 9 are direct translations of the mathematical expressions (1.1)-(1.3). The order of the terms is irrelevant. In contrast to some restrictive modeling languages, we could have written line 7 as

```
1500>=x[0]+x[2]+x[1]
```

[7] Writing in Common Lisp would be even better. Alas, there is no Lisp binding for or-tools yet.

[8] https://developers.google.com/optimization/lp/glop

[9] https://www.gnu.org/software/glpk/

[10] https://projects.coin-or.org/Clp

or

```
x[0]+x[1]+x[2]-1500<=0
```

or any other equivalent algebraic expression.

At line, 6, we declare an auxiliary variable, `pop`. Though there is no such distinction in the modeling language, this is not a decision variable but rather a helpful device to model the problem. We use this auxiliary on line 10 where we add an equation that does not constrain the model in any way. It is simply defining the auxiliary variable `pop` to be the sum of our decision variables. This allows us to express the objective easily and, possibly, to help display the solution.

We have the objective function on line 11, a translation of (1.4). The function choices are, unsurprisingly, either `s.Maximize` or `s.Minimize` with, for parameter, a linear expression in terms of the variables declared previously. We used

```
s.Maximize(pop)
```

We could have written

```
s.Maximize(x[0]+x[1]+x[2])
```

We then call on the solver at line 12 to do its job. This is where all the computational work gets done; work that we will not describe. The interested reader can search for "Simplex method" and "Interior-point methods" to learn about the fascinating theory[11] behind the solution methods of linear optimization models. To understand the simplex method, one needs only high-school algebra. To understand interior-point methods requires somewhat more mathematical background.

For some models, solvers may complete their work in a fraction of a second; for others, it may take hours. Moreover, not all solvers will have the same runtime behavior. Model A may run faster than model B on solver X while it may be exactly the reverse on solver Y. One more advantage or using the or-tools library is that we can try out another solver by changing one line.

We should, if this code were meant for production and the problem non-trivial, check the return value to ensure that the solver found an optimal solution. It may have aborted because of a model error, or because it ran out of time or memory, or for some other reason. But for our simple first example, we will forgo good engineering practice in the name of simplicity of exposition.

We return, on line 13, both the optimal objective function value held in variable `pop`, as well as the optimal values of the decision variables (not all the associated object attributes carried by those variables).

On more complex models, we may post-process the decision variables to return something simpler and more meaningful to the caller. We will see a good example of this when we solve the shortest path problem in Section 4.4. The general approach we encourage is to create models that can be used without any knowledge of the internals of or-tools. The modeler is responsible for the creation of the model, but once the model is created and validated, it should leave the hands of its creator for those of the domain expert who originally formulated the problem.

When the diligent reader executes Code 1.2, he will observe a result similar to Table 1.3.

[11] See, for example, Theory of Linear and Integer Programming, by Alexander Schrijver, isbn 978-0471982326.

Code 1.2: How to execute the coexistence model.

```
from coexistence import solve_coexistence
pop,x=solve_coexistence()
T=[['Specie', 'Count']]
for i in range(3):
  T.append([['Toads','Salamanders','Caecilians'][i], round(x[i])])
T.append(['Total', pop])
for e in T:
  print e[0],e[1]
```

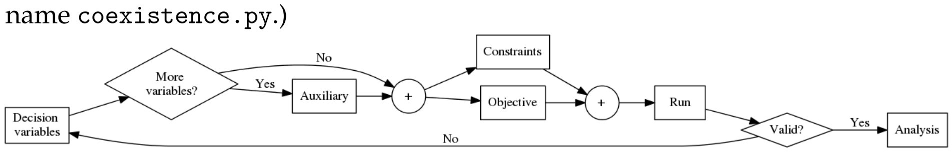
Table 1.3: Solution to the coexistence problem.

Specie	Count
Toads	100.0
Salamanders	300.0
Caecilians	1000.0
Total	1400.0

Notice that we can look at the solution of Table 1.3 and see that it does indeed satisfy the constraints. By substituting the solution into (1.1)-(1.3), we obtain

$$\begin{aligned} 2(100.0) + 300.0 + 1000.0 &= 1500 & \leq 1500, \\ 100.0 + 3(300.0) + 2(1000.0) &= 3000 & \leq 3000, \\ 100.0 + 2(300.0) + 3(1000.0) &= 3700 & < 5000. \end{aligned}$$

Notice that the first two inequalities are satisfied with equality. In the jargon of optimization, such inequalities are *tight* or *active*. The last one is said to be *slack* or *inactive*. In a certain sense, we could delete it from the problem and nothing would change. (The reader can try this and other modifications. The code is available in the additional material under the name `coexistence.py`.)



In summary, the steps to construct and run a model are:

- Formulate the question precisely.
- Define the decision variables by identifying what is required to answer the question.
- Possibly define auxiliary variables to help simplify the statements of constraints or of the objective function. They can also help in the analysis and the presentation of the solution.
- Translate each constraint into an algebraic equality or inequality involving directly the decision variables or indirectly, through the auxiliary variables.
- Construct the objective function as some quantity that should be minimized or maximized.

- Run the model using an appropriate solver.
- Display the solution in an appropriate manner.
- Validate the results: Does the solution correctly satisfy the constraints? Is the solution meaningful and implementable? If so, declare that we are done; if not, consider the necessary modifications to the model.

The rest of the text will construct models of increasing complexity, illustrating and expanding the points above.

Chapter 2

Linear continuous models

At the dawn of optimization (the nineteen fifties), the state-of-the-art was defined by linear optimization models and the simplex method, the only reasonably efficient algorithm known at the time to solve such models. When I started studying this subject, one repeatedly heard from multiple sources that over 70% of the CPU cycles in the world were devoted to running various simplex codes. Surely an exaggeration, but it is indicative of the power of linear models. The world is not linear, but sometimes, a linear approximation is **good enough**.

More precisely, we discuss here *Linear Continuous* models (though the usage is to call these models LPs for Linear Programs, implying the continuity properties). Linear continuous models are the simplest to write down and the simplest to solve. They have been the workhorse of optimizers since George Dantzig invented the simplex method to solve them. What characterizes them is three elements:

- All variables are continuous.
- All constraints are linear.
- The objective function is linear.

In detail, the decision variables (say $x_0, \dots, x_n$) can take on integral and fractional values. This is appropriate when the solution is measuring amounts (for example pounds of flour or tons of concrete). It is not appropriate when the solution is counting objects, (as in people or politicians) unless one is looking only for an approximation.

The objective function is (or can be) parameterized by a constant array c and expressed as

$$c_1 x_1 + c_2 x_2 + ... + c_n x_n.$$

This limitation precludes objective functions with terms of the form

$$x_1^2, x_4^3, \sin(x), e^{x_3}, x_1 \cdot x_2, \text{ or } |x|$$

among infinitely many others. Though we will see later how to handle some of these nonlinearities by model transformation.

Finally, the constraints are parameterized by a matrix a_{ij}, an array b, and can be stated as a set of relations, for $i \in \{1, \dots, m\}$,

$$a_{i1} x_{i1} + a_{i2} x_{i2} + ... + a_{in} x_{in} \geq b_i, \quad \text{or } \leq b_i, \quad \text{or } = b_i,$$

or some equivalent algebraic form.

In this chapter we consider problems where the natural formulation is such a linear continuous model.

2.1 Mixing

The canonical linear programing example is the *diet problem*, one of the first optimization problems to be studied in the thirties and forties (twentieth century, not twenty-first[1]). The likely apocryphal origin of the problem is the US military's desire to meet the nutritional requirements of the field GIs while minimizing the cost of the food. One of the early researchers to study this problem was George Stigler. He made an educated guess of the optimal solution to the linear program using a heuristic method. In the fall of 1947, Jack Laderman of the Mathematical Tables Project of the National Bureau of Standards (NBS, today NIST) undertook solving Stigler's model with the new simplex method. The linear model consisted of nine equation in 77 unknowns, a huge problem for the time. Some models in this book are orders of magnitude larger and will be solved in a minuscule fraction of the time it took the NBS people to solve the diet problem in 1947. The increase in efficiency is partly due to the hardware, but mostly due to the software.

A generic version of the problem is:

> Given a list of food with some nutritional content, each with a cost, find the combination of food that will minimize cost and yet provide all the necessary nutrients.

Here is one simple version of this problem. We will let the foods be F0, F1, F2, F3, etc. (Imagine them to be pizza, ramen noodles, cupcakes or, if you are of a more health-conscious bent, tofu, green peas, quinoa, etc.) The nutrients will be represented by N0, N1, N2, N3, etc. (Imagine them to be calories, protein, calcium, vitamin A, etc.). Each has a cost per serving. In addition, to avoid eating one food all week long, let us restrict the number of servings per week.

A randomly generated instance is given in the Table 2.1.[2] Each row represents a food, with the nutritional content per serving, followed by the acceptable range of servings of the food and its cost per serving. Ignore the last row and column for now. We will return to them after the model is constructed and solved. The two rows before last represent the allowed range of each nutrient.

Table 2.1: Example of data and solution for the diet problem.

	N0	N1	N2	N3	Min	Max	Cost	Solution
F0	606	563	665	23	7	17	9.06	17.0
F1	68	821	83	72	6	27	8.42	7.47
F2	28	70	916	56	1	36	9.47	6.11
F3	121	429	143	38	14	26	6.97	14.0
F4	60	179	818	46	9	35	4.77	35.0
Min	5764	28406	48157	1642				
Max	15446	76946	82057	6280				
Sol	14775	28406	48157	3413			539.37	

[1] I add this temporal precision on the odd chance that this text is still being read long after my body has maximized its entropy.

[2] To encourage the reader to experiment, every model in this book is available in the additional material, along a random instance generator.

2.1.1 Constructing a model

What would a solution be but a list of servings of each food. Therefore the decision variables must be one per food, representing the number of servings. Let us name these variables $f_0, \ldots, f_n$. We will assume that it is acceptable to have fractional answers. (i.e. one half serving is acceptable.)

The objective is to minimize cost. We have one cost per food ($c_0, \ldots, c_n$. These are not variables, they are data.) Therefore, what we want is to minimize the sum of all the products $c_i \times f_i$. This leads to the objective function

$$\min \sum_i c_i f_i.$$

Let us tackle the constraints. We have two sets: one indicating the range of acceptable servings of each food (assume that the minimum of food i is l_i and maximum is u_i) and one indicating the required nutrients range (mininimum of nutrient j is a_j and maximum is b_j). The simpler constraint is related to the food. Since our decision variables indicate the number of servings of each food, we need only to box each serving count,

$$l_i \leq f_i \leq u_i. \tag{2.1}$$

The constraint on nutrients is a bit more involved. Consider nutrient j. How much of it will be included in the diet? Each food i may have some of it, as indicated in Table 2.1. Let us call this amount N_{ji} (corresponding to the entry at row of food i and the column of nutrient j). To get the total of this nutrient, we therefore need to sum over all foods the product of the food serving and the nutrient content. For each nutrient j,

$$a_j \leq \sum_i N_{ji} f_i \leq b_j.$$

We are done with the theory. Let us translate this into an executable model general enough to solve all problems of this type. We will assume that the data is given in a 2-dimensional array called `N`. It has the structure of Table 2.1 without the last column and row. Each row represents a food, except that the last two rows represent the minimum and maximum requirement of each nutrient, represented by the columns, with the last three representing the minimum, maximum and the cost of each food serving.

Code 2.1: Model for minimal cost diet.

```
def solve_diet(N):
  s = newSolver('Diet')
  nbF,nbN = len(N)-2, len(N[0])-3
  FMin,FMax,FCost,NMin,NMax = nbN,nbN+1,nbN+2,nbF,nbF+1
  f = [s.NumVar(N[i][FMin], N[i][FMax],'f[%i]'%i) for i in range(nbF)]
  for j in range(nbN):
    s.Add(N[NMin][j] <= s.Sum([f[i]*N[i][j] for i in range(nbF)]))
    s.Add(s.Sum([f[i]*N[i][j] for i in range(nbF)]) <= N[NMax][j])
  s.Minimize(s.Sum([f[i]*N[i][FCost] for i in range(nbF)]))
  rc = s.Solve()
  return rc,ObjVal(s),SolVal(f)
```

The model uses the `newSolver` function to simplify the expression of the code[3] as the reader can see at Code 2.2. These, and other simplification, can be found in `my_or_tools.py`.

Code 2.2: Utility function to create an appropriate solver instance.

```
from linear_solver import pywraplp
def newSolver(name,integer=False):
  return pywraplp.Solver(name,\
                   pywraplp.Solver.CBC_MIXED_INTEGER_PROGRAMMING \
                   if integer else \
                   pywraplp.Solver.GLOP_LINEAR_PROGRAMMING)
```

To help the expression of the model, lines 3 - 4 give meaningful names to the row and column indices that we will use. In line 5 we define the decision variables, one per food, each taking values in the range $[l_i, u_i]$ as in (2.1). It would be correct to give a range of $[0, +\infty)$ and then add constraints to enforce the bounds. The solver would still find the same solution; but it is simpler and good practice to limit as much as possible the range of decision variables. In complex models, it often dramatically improves the solution time.

The two-line loop starting on line 6 establishes the range on each nutrient as in (2.1.1). Lines 9 and following create the objective function, solve the problem and return three numbers: the status of the solver (it should be zero), the optimal value and optimal solution. The dual role of the functions `SolVal` and `ObjVal` (seen at Code 2.3) is to simplify the results returned to the caller and the code to read.

Code 2.3: Utility functions to extract values from the or-tools objects.

```
def SolVal(x):
  if type(x) is not list:
    return 0 if x is None \
      else x if isinstance(x,(int,float)) \
          else x.SolutionValue() if x.Integer() is False \
              else int(x.SolutionValue())
  elif type(x) is list:
    return [SolVal(e) for e in x]

def ObjVal(x):
  return x.Objective().Value()
```

The results from executing this model are shown in the last row and column of Table 2.1. The column indicates the number of servings of each food and the row indicates the amount of each nutrient that will be in the diet. The reader should notice that many of the food items and nutrient counts are at their minimum required values. This is expected of such a model since we are trying to minimize a linear cost function; the optimal solution should push towards the boundary of the constraints as much as possible.

The reader can experiment with this model. It is included in the additional material as `diet_problem.py`, along with a generator of random diet problems and a routine to display the solution in a table format similar to Table 2.1.

[3]Mostly to make the code fit a page, but also to hide some of the verbosity of the or-tools library. The authors having chosen, rightly in my opinion, meaningful, but rather long names for their functions.

2.1.2 Variations

- There are a number of simple variations of this problem. For instance, instead of minimizing cost, we could be given a profit to maximize. We could also not have either the minima or the maxima in either the foods or nutrients.

- It becomes more complex, consequently interesting, when we have, in addition, requirements of the form: "If food 2 is used, then we must have at least as much food 3 in the diet". Or "Nutrient 3 must be included in at least twice the amount as nutrient 4".

  Let us consider some of these in detail. First "If food 2 is used, then food 3 must also be included in at least as many servings". The following inequality ensures the required result:
  $$f_3 \geq f_2.$$
  Notice that food 3 could still be included when food 2 is not, but that does not violated the requirement. And if food 2 is included, then we will have at least as many servings of food 3. It should be clear that the requirement could have been stated in reverse "No more food 2 than food 3." The constraint is the same.

- A requirement on the nutrients: "Nutrient 3 must be included in at least twice the amount as nutrient 4." has a similar flavor but we note that the amount of any given nutrient is spread among all the food items. It may be fruitful to introduce auxiliary variables that will tally the nutrients, say n_j. We then add to the model one equality per nutrient,
  $$n_j = \sum_i N_{ij} f_i.$$
  Note that these equations do not constrain the problem; their insertion is simply a helpful device to implement the requirement. We can now easily relate the nutrient content according to the new requirement as
  $$n_3 \geq 2n_4.$$
  which we could have stated, had we not defined the variables n_i, as
  $$\sum_i N_{i3} f_i \geq 2 \sum_i N_{i4} f_i.$$
  Defining the auxiliary variables n_j seems clearer. Moreover, displaying the total of each nutrient at the end might help with the analysis or the presentation of the solution.

- A similar requirement may occur to the reader, namely "If food (nutrient) 3 is used then food (nutrient) 4 must not be (and vice versa). This may look like a simple variation to the above but it is decidedly **not** simple. If fact, it forces the modeler to use a different modeling technique. We will see how to implement such requirements in later chapters (see, for instance, Section 6.5). There are two valid approaches to model such requirements properly: Integer Programming and Constraint Programming. The reader is encouraged to spend some time trying to model such constraints to develop intuition into the difficulties. The key, and the reason that this is a beast of an entirely different ilk is that the change is not uniquely quantitative (as much as, or twice the amount of) but is additionally qualitative: we transition between having an element and not having that element.

Table 2.2: Abstract structure of product mix problems.

		Components			Availabilities		Cost
		C_1	...	C_n	Min	Max	
Products	P_1	99	...	99	99	99	99
	...	...	...	...	...	...	...
	P_m	99	...	99	99	99	99
Demands	Min	99	...	99	99	99	99
	Max	99	...	99	99	99	99
Price		99	...	99	99	99	99

2.1.3 Structure of the problems under consideration

Problems with the structure of the diet problem are generally known as *product mix* problems. They can be presented in various ways but if they can be fitted into the abstract Table 2.2 they can all be handled in the manner described in this section. Of course, it may be that some of the columns or rows are missing (no cost, or no price, or no maximum demand, etc...) That only simplifies the model.

The decision variable indicates the amount of product needed and the constraints indicate availabilities of the raw material or, equivalently, capacities of the processing units as well as demand bounds. The objectives are often profits to maximize or costs to minimize or, simply, quantity to produce.

Here are a few instances to help the reader recognize the underlying structure. The reader is encouraged to marshall the problems into the format of Table 2.2 by inventing numbers. [4]

- A factory is producing cement of various types. Each product is composed of the same elements, but in various quantities and we have on hand a limited supply of each of these elements, each with a cost. To each final product is associated a profit. What is the best mix of product to produce to maximize profit?

- A Florida based fruit company produces orange drinks, juices and concentrates for various markets. The raw materials for all products are oranges, sugar, water and time, in various quantities, some positive and some negative (producing orange drinks requires water; producing concentrates generates water). Given certain availabilities, how much can they produce to maximize profit?

- A toy manufacturer produces a number of different toys. Each is composed of a number of basic materials and, in addition, requires special processing (assembling, painting, boxing). The processing is done on specialized machines and has a duration. Since the manufacturer has limited supplies of materials and machines, which can only operate a certain number of hours per day, how many toys can be produced?

- A fertilizer company, Bush, Rove and Company, (BR & Co.) has two products, a high-phosphate blend and a low-phosphate blend. These are produced by mixing different raw materials in various quantities.

  They can procure, from their own subsidiaries, at most some amount, of each raw material per day at a fixed internal cost. This cost includes labor, power, depreciation,

[4] The web site `www.practicalopt.com` will generate some of these variations on demand.

delivery, bribes, etc. In addition, the mixing process incurs a certain cost per ton for each product.

Both products are sold to a wholesaler, Fox inc., at a fixed price. Moreover, The wholesaler has agreed to buy all the production BR & Co. can produce. How much of each fertilizer should they produce?

- Queequeg sells half-kilo bags of coffee in three blends, House, Special and Gourmet, which sell at different prices per bag. Each of the three blends are made up of Colombian, Cuban and Kenyan coffee beans in various proportions. Queequeg has on hand some Colombian, Cuban and Kenyan. How much of each blend should they bag to maximize revenues?

2.2 Blending

A second type of problems that readily admits a linear model is the *blending problem*. The classical example involves blending so-called raw or crude gasolines to achieve various refined products with specified octane value. For instance, let us assume that we are given crude gasolines `R0`, `R1`, . . . , `Rn` each with a certain octane rating, maximum availabilities in barrels and a cost in dollars per barrels as in Table 2.3.

Table 2.3: Example of raw gasolines for the blending problem.

Gas	Octane	Availability	Cost
R0	99	782	55.34
R1	94	894	54.12
R2	84	631	53.68
R3	92	648	57.03
R4	87	956	54.81
R5	97	647	56.25
R6	81	689	57.55
R7	96	609	58.21

We are also given demands for multiple types of refined gasolines, (think Bronze, Silver and Gold, for instance) with each their own octane ratings. The demands are stated in minimal and maximal number of barrels along with their selling prices as in Table 2.4.

Table 2.4: Example of refined gasolines for the blending problem.

Gas	Octane	Min demand.	Max demand.	Price
F0	88	415	11707	61.97
F1	94	199	7761	62.04
F2	90	479	12596	61.99

We create the three types by mixing the appropriate raw gasolines together, assuming that the octane rating of a mix is a linear function of the volumes mixed. This is a crucial assumption: if we mix half and half of octane ratings 80 and 90, we get an octane rating of 85 because

$$\frac{1}{2}80 + \frac{1}{2}90 = 40 + 45 = 85.$$

If we mix 40% of octane 80 and 60% of octane 90 we get

$$\frac{40}{100}80 + \frac{60}{100}90 = 32 + 54 = 86.$$

This assumption is the key to the blending model.

Notice that there might be a number of ways to mix the raw gasolines together to get the required ratings. Our task is to construct a model that will tell us exactly how to mix the raw products to satisfy the demands and maximize the profits (understood as the difference between the total selling price of the finished products and the total cost of the raw gasolines).

2.2.1 Constructing a model

What is the question to answer? Let us ask this question a number of times, with increasing precision. A first stab is: "How much of each type of refined gas to produce?" This is correct but is incomplete, since we need to know the composition of each refined gas; how much of each crude goes into each mix. "How to mix the crude gas to produce the refined gas?" This is the right question, but is not yet in the proper form for an algebraic model. Imagine you are the manager of the refinery. On one side you have all these tanks filled with crude gas, and on the other side, all empty tanks that will contain the refined gas. In between: miles of pipes with valves that you control. What you really want to know is which valves to open and by how much to have exactly the right mix. So the proper question is "How much of each crude gas goes into each of the refined gas?".

The key difference between the *mixing* problems of the preceding section and this *blending* problem is that previously, we were told the exact composition of the products in terms of the material (in each food, the amount of each nutrient, for instance) while, in the problem considered here, the composition of each product is one of the answers sought.

Since we need to know "How much of raw i goes into refined j?", we are led to a two-dimensional decision variable, say G_{ij} where i is the index of the crude gas and j is the index of the refined gas. For example $G_{51} = 250$ will mean that there are 250 barrels of crude 5 going into the mix of refined 1. We understand here that the units will be barrels; it seems natural as the prices are per barrels. We probably should also introduce auxiliary variables to help us model and present the solution: the total of each crude gas (the sum of a row of G), say R_i, and the total of each refined gas (the sum of a column of G) say F_j. So we will have these non-constraining equations in the model:

$$R_i = \sum_j G_{ij} \quad \forall i \tag{2.2}$$

$$F_j = \sum_i G_{ij} \quad \forall j. \tag{2.3}$$

Note that, by construction, $\sum R_i = \sum_j F_j$. We need not enforce this though we need it. We can think of this as a "continuity" equation: it reflects that the refining process does not lose product along the way. This idea of continuity is a useful modeling idea. It will re-appear in various guises throughout the models we develop.

Armed with these variables, we can now easily model the objective function. We are asked to maximize profits, hence the difference between total sales (given price p_j for refined gas j) and costs (given cost c_i for crude gas i),

$$\max \sum_j F_j p_j - \sum_i R_i c_i.$$

The constraints come in multiple form. The easy ones are, as in the mixing problems, constraints on the availability of each raw material. With our auxiliary variables, these are simple to express and can be included in the range of the defined variables or in a constraint

$$0 \leq R_i \leq u_i \quad \forall i.$$

The constraints on the demand of refined gas (minimum and/or maximum) are just as simple:

$$a_j \leq F_j \leq b_j \quad \forall j.$$

Notice how our auxiliary variables help write down constraints. Having only our decision variables, the constraints would have to be written with respect to column and row sums.

The only real complication of this problem refers to the octane rating. The key here is the assumption of linearity. To see how to model the octane requirement, let us imagine a simple case: say we mix 800 barrels of crude 1 at octane rating of 98 with 200 barrels of crude 2 at octane rating of 90. What is the resulting octane rating? Since we have a total of 1000 barrels of refined,

$$\frac{800 \times 98 + 200 \times 90}{1000} = 96.4$$

So, in general, we need the fraction of each crude that goes into a mix, times its octane rating. Assuming O_i as the octane rating of crude i and o_j the octane rating of refined j, this leads us to:

$$\sum_i O_i G_{ij} = F_j o_j \quad \forall j \tag{2.4}$$

We now have an algebraic linear model. Let us translate it into executable code. We will assume that the data are entered in two-dimensional arrays, exactly as Tables 2.3 and 2.4, except for the first columns, added for reference only.

Code 2.4: Gasoline blending model.

```
1  def solve_gas(C, D):
2    s = newSolver('Gas␣blending␣problem')
3    nR,nF = len(C),len(D)
4    Roc,Rmax,Rcost = 0,1,2
5    Foc,Fmin,Fmax,Fprice = 0,1,2,3
6    G = [[s.NumVar(0.0,10000,'G[%i][%i]' % (i,j))
7          for j in range(nF)] for i in range(nR)]
8    R = [s.NumVar(0,C[i][Rmax],'R[%i]' %i) for i in range(nR)]
9    F = [s.NumVar(D[j][Fmin],D[j][Fmax],'F[%i]' %j) for j in range(nF)]
10   for i in range(nR):
11     s.Add(R[i] == sum(G[i][j] for j in range(nF)))
12   for j in range(nF):
13     s.Add(F[j] == sum(G[i][j] for i in range(nR)))
14   for j in range(nF):
15     s.Add(F[j]*D[j][Foc] ==
16           s.Sum([G[i][j]*C[i][Roc] for i in range(nR)]))
17   Cost = s.Sum(R[i]*C[i][Rcost] for i in range(nR))
18   Price = s.Sum(F[j]*D[j][Fprice] for j in range(nF))
19   s.Maximize(Price - Cost)
20   rc = s.Solve()
21   return rc,ObjVal(s),SolVal(G)
```

At lines 3-5 we declare some constants to access the appropriate rows and columns of the data. The constraints on the range of each variables are entered, not as constraints, but rather as range on the corresponding variables. The equations (2.2)-(2.3) are seen on the four lines starting at 10.

The blending equations are created on the loop of line 14. Note that since the goal is to achieve a certain octane level, we might replace the equality with an inequality, indicating that the refined product has *at least* the required octane level. This relaxes the problem a little and allows optimization over a larger space. This might be required if, for example, we did not have sufficient low octane crude gasolines available.

The objective function (three lines starting at 17) maximizes the difference between the selling price of the refined product and the cost of the crude gas used.

Executing this model with the data above produces the Table 2.5 where the bottom right number is the profit: the difference between the sum of the row *Price* and the column *Cost*.

Table 2.5: Complete solution to blending problem.

	F0	F1	F2	Barrels	Cost
R0	542.5		239.5	782.0	43275.88
R1		894.0		894.0	48383.28
R2	631.0			631.0	33872.08
R3		648.0		648.0	36955.44
R4	704.41	251.59		956.0	52398.36
R5		647.0		647.0	36393.75
R6	449.5		239.5	689.0	39651.95
R7	50.93	558.07		609.0	35449.89
Barrels	2378.33	2998.67	479.0		
Price	147385.32	186037.28	29693.21		36735.18

2.2.2 Variations

While blending problems can be presented in various ways, they can all be handled in the manner above. The decision variables should be two-dimensional; the sum in one dimension and in the other indicating total input material used and total output material produced. Finally, in addition to the capacity and demand constraints, there should be, at least, one blending constraint satisfying a linearity assumption.

One interesting variation is that we might be asked to achieve more than one characteristic. For instance, in addition to an octane level, we might also be given a certain concentration of sulphur in each of the crude and asked to keep the refined gas below certain sulphur threshold. In this case, the octane equation (2.4) will almost certainly need to be replaced by an inequality, ensuring a minimum octane level, and another similar inequality will ensure a maximum sulphur lever. Assuming S_i as the sulphur level of crude i and s_j as the sulphur level of refined j, we get

$$\sum_i O_i G_{ij} \leq F_j o_j \quad \forall j$$
$$\sum_i S_i G_{ij} \geq F_j s_j \quad \forall j.$$

The reason for the inequalities is that it is unlikely for the problem to have any feasible solution with exactly the specified octane and sulphur levels. The reader might try to modify Code 2.4 to verify this.

To help the reader recognize the underlying structure of blending problems, here is an instance we will revisit soon, with additional complexities.

A very popular ingredient in junk food is manufactured by refining and blending various oils together. The oils come in five flavors (O1 to O5) and measures of ‘hardness’ as given in the following table where cost is in dollars per tons and the hardness is measured in the appropriate unit.

	O1	O2	O3	O4	O5
Cost	110	120	130	110	115
Hardness	8.8	6.1	2.0	4.2	5.0

The oils O1 and O2 can be refined at production facility A which has a capacity of 200 tons per month while O3, O4 and O5 can be refined at production facility B which has a capacity of 250 tons per month. There is no loss of weight during the refining process and you can ignore the cost of the process.

The final product is obtained by mixing various amounts of the five oils. It has a hardness restriction. Measured in the same unit as given in the table, it must lie between 3 and 6 units. It is assumed that hardness blends linearly. That is, if we mix 10 tons of O1 with 20 tons of oil O2 the blend will have a hardness rating of

$$(10 \times 8.8 + 20 \times 6.1)/(10 + 20).$$

The final product sells for 150 dollars per ton. How should the oils be refined and blended to maximize profit?

2.3 Project management

Project Management, as is usually understood in the context of optimization, refers to a set T of tasks, each with with two properties,

- a duration,
- a subset of T (possibly empty) of preceding tasks.

The classical example is house construction: tasks include 'Finding location', 'Drawing plans', 'Getting permits', 'Breaking ground', 'Laying foundations', 'Building walls', 'Installing plumbing', 'Bribing inspector', etc. Crucially, some tasks must be done before others: you cannot 'Build the roof' until you 'Raise the walls'. The main question under consideration: "When should each task start to minimize the total project completion time?" That is, when do we start each task to have the house entirely built in the shortest time possible? Incidentally, if one task falls behind schedule, what is the impact on all the ulterior tasks and how do we re-schedule them.

Table 2.6 is an instance of such a project we will use to illustrate a solution technique.

Table 2.6: Example of project management tasks.

Task	Duration	Preceding tasks
0	3	{ }
1	6	{ 0 }
2	3	{ }
3	2	{ 2 }
4	2	{ 1 2 3 }
5	7	{ }
6	7	{ 0 1 }
7	5	{ 6 }
8	2	{ 1 3 7 }
9	7	{ 1 7 }
10	4	{ 7 }
11	5	{ 0 }

2.3.1 Constructing a model

What we need to decide in this instance is how early to start each task, respecting precedence, to minimize the total completion time. This suggests, as a decision variable, the starting time of each task in the same units as the given durations. Let us assume a set T of tasks (corresponding to the first column of Table 2.6) to declare our decision variables as

$$0 \leq t_i \quad \forall i \in T.$$

To ensure that precedence requirements are met, let us assume that we have, in addition to duration D_i (corresponding to the second column of Table 2.6), subsets $T_i \subset T$ of preceding tasks for each task i (corresponding to the third column of Table 2.6). Then we need to lower bound the starting times by

$$t_j + D_j \leq t_i \quad \forall j \in T_i; \forall i \in T.$$

The objective is to minimize the project completion time. This time would be the starting time of the last task plus its duration if the tasks were all done sequentially. But they are likely not; we might be doing as many tasks in parallel as possible. Then how do we find the completion time if we do not know the last task, or if there is no single 'last' task?

Let us introduce another variable, t. We will constrain this t to be larger than, for each task, its starting time plus its duration. It will therefore be larger than the completion time. And if we add the objective $\min t$, to the set of constraint :

$$t_i + D_i \leq t \quad \forall i \in T.$$

then t will, at optimality, be the completion time. A condition that will hold, no matter how many tasks we do in parallel.

We now translate this into an executable model in Code 2.5 where we assume that the data is given to us in a table `D` with the same structure as Table 2.6: each row has a task identifier, a duration and a set, possibly empty of preceding tasks.

Code 2.5: Project management model.

```
1  def solve_model(D):
2    s = newSolver('Project management')
3    n = len(D)
4    max = sum(D[i][1] for i in range(n))
5    t = [s.NumVar(0,max,'t[%i]' % i) for i in range(n)]
6    Total = s.NumVar(0,max,'Total')
7    for i in range(n):
8      s.Add(t[i]+D[i][1] <= Total)
9      for j in D[i][2]:
10       s.Add(t[j]+D[j][1] <= t[i])
11   s.Minimize(Total)
12   rc = s.Solve()
13   return rc, SolVal(Total),SolVal(t)
```

Line 4 computes a valid upper bound on the times by adding all the durations. This is clearly an overestimate but is fine to use in the declaration of the decision variables at line 5. We declare the total completion time variable at line 6, which we use as an upper bound on all starting time plus duration at line 8. Finally we add the precedence bounds at line 10. The results appear at Table 2.7 and, graphically, at Figure 2.1. Note that the last ending time is the total project completion time.

Table 2.7: One optimal solution to the project management problem.

Task	0	1	2	3	4	5	6	7	8	9	10	11
Start	0	3	0	3	9	0	9	16	26	21	24	23
End	3	9	3	5	11	7	16	21	28	28	28	28

Note that all tasks could have started at any time after their required tasks have ended. And in fact, depending on the solver used, the solution might look rather different. We can see an example of an alternate solution at Table 2.8. This situation of multiple optimal solutions offers us, as modelers, an opportunity to improve the model. In this particular case, it might be useful to start all tasks as early as possible. This will not affect the total

Table 2.8: An alternate optimal solution to the project management problem.

Task	0	1	2	3	4	5	6	7	8	9	10	11
Start	0	3	0	3	9	0	9	16	21	21	21	3
End	3	9	3	5	11	7	16	21	23	28	25	8

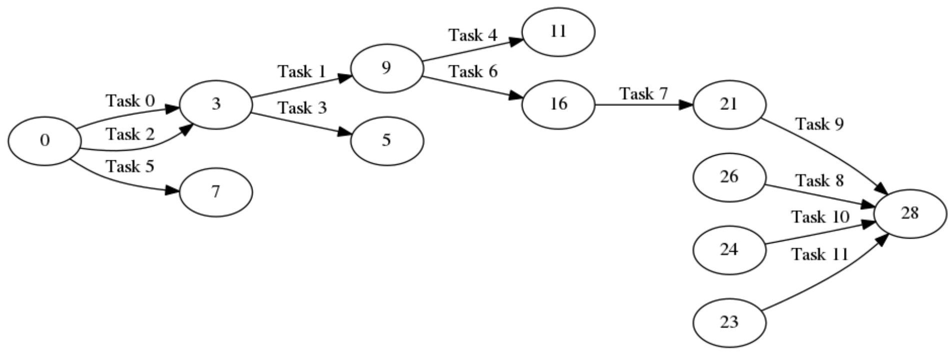


Figure 2.1: Graphical representation of example solution. (Nodes are times.)

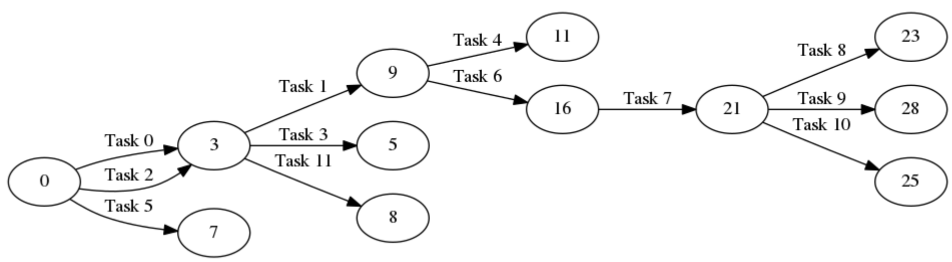


Figure 2.2: Graphical representation of alternate solution.

completion time but might make the project more practical and less prone to delays if some tasks' duration were poorly estimated.

Note finally that, by looking at the graphical representation, it is clear that the subset of tasks `0,2,1,6,7,9` is critical in the sense that, if any of those is delayed, the project completion time is delayed. On small projects, such a graphical representation is sufficient to identify the critical tasks. On larger projects it might be profitable to identify these tasks programmatically. We will see one way to compute critical paths in Section 4.4.3 when we discuss longuest paths.

2.3.2 Variations

We displayed two possible solutions to our problem. The alternate might be preferable for practical reason. How can we ensure that, among all solutions that minimize total completion time, we choose a solution that starts all tasks as early as possible? One way is to minimize the sum of starting times. That is, replace the objective function by

```
s.Minimize(sum(t[i] for i in range(n)))
```

In cases like this, optimizers talk of *multiple objectives*. In general, these might be independent, or worse, contradictory. But in our project management situation the objectives (minimizing completion time and starting all tasks as early as possible) are coherent. Note that the new optimal value of the model is neither interesting nor useful. We need to inspect the `Total` variable to give us the completion time.

2.3.2.1 Minimax problems

The technique we used for the Project Management can be used more generally whenever we face a *minimax* problem. This is a problem where we want to minimize the maximum of some set of functions. For example, let us assume we want to find the optimal x for

$$\min_x \max_{i \in T} \sum_j a_{i,j} x_j.$$

This is handled by introducing a new variable, say t, along with the objective

$$\min t$$

and the constraints

$$\sum_j a_{i,j} x_j \leq t \quad \forall i \in T.$$

The corresponding *maximim* problem is handled the similarly. Note that the related *maximax* and *minimim* are considerably more difficult to handle. We will revisit those in a later section (See 6.5.4.)

2.3.2.2 Absolute value problems

Essentially the same approach can also be used for some non-linear functions, for instance, those involving absolute values. Say we seek the optimal x for

$$\min_x |\sum_j c_j x_j|.$$

Since the absolute value function is defined as

$$|z| = \begin{cases} z & z \geq 0 \\ -z & z < 0, \end{cases}$$

we can use the same $\min t$ objective along with constraints

$$\begin{aligned} \sum_j c_j x_j &\leq t && \forall i \in T \\ -(\sum_j c_j x_j) &\leq t && \forall i \in T. \end{aligned}$$

We will illustrate some non-trivial applications of this technique in Section 3.2.

2.4 Multi-stage models

In life, decisions at one stage often influence decisions at a later stage. The same holds for more pedestrian situations. For instance, consider a warehouse: what it contains at the end of a month surely should influence what is ordered at the beginning of the following month.

In a certain sense there is little that is new in these multi-stage models except that we have to be careful to properly setup the continuity from one stage to the next.

To illustrate, we will revisit the blending problem. To multiple targets, prices and costs, we will add a planning horizon of many months. These will exemplify the stages. This problem will require all the tricks and techniques we have seen so far (and then some). It forms is a comprehensive review of the current chapter.

2.4.1 Problem instance

Soap is manufactured by refining and blending various oils together. The oils come in various flavors (Apricot, Avocado, Canola, Coconut, etc.) and each oil contains multiple fatty acids (Lauric, Linoleic, Oleic, etc.) in various proportions. For example, see Table 2.9.

Table 2.9: Example of oils (Oi) with their acid content (Aj).

	A0	A1	A2	A3	A4	A5	A6
O0	36	20	33	6	4		1
O1		68	13			8	11
O2		6		66	16	5	7
O3		32				14	54
O4			49	3	39	7	2
O5	45		40		15		
O6						28	72
O7	36	55					9
O8	12	48	34		4	2	

According to the properties of the soap one is creating (Cleaning power, Lather production, Dryness of the skin, etc...) one targets the final proportions of the fatty acids to be in certain ranges by blending the oils appropriately. For instance, we will target our soap to have acid contents in the ranges of Table 2.10.

Table 2.10: Fatty acid content targets.

	A0	A1	A2	A3	A4	A5	A6
Min	$13.25	$23.22	$17.79	$3.73	$4.56	$8.80	$23.65
Max	$14.65	$25.66	$19.67	$4.13	$5.04	$9.72	$26.13

Here is an additional twist, relative to periods. We will be planning for a certain number of months. Each oil may be purchased for immediate delivery or bought on the futures market for delivery in a later month. The prices of each oil, in each of the month, are given in Table 2.11 in dollars per ton.

It is possible to store up to 1000 tons of oils for later use (any combination of oils) but there is a holding cost of 5 dollar per ton per month. Finally, we must satisfy a demand of 5000 tons of soap per month. This demand drives the model.

Table 2.11: Cost of oils in dollars per ton over the planning horizon.

	Month 0	Month 1	Month 2	Month 3	Month 4
O0	118	128	182	182	192
O1	161	152	149	156	174
O2	129	191	118	198	147
O3	103	110	167	191	108
O4	102	133	179	119	140
O5	127	100	110	135	163
O6	171	166	191	159	164
O7	171	131	200	113	191
O8	147	123	135	156	116

At the beginning of the planning horizon, we have some oils in inventory as illustrated in Table 2.12. How should the oils be refined and blended every month to minimize cost?

Table 2.12: Initial inventory (in tons).

Oil	Held at beginning(in tons)
O0	15
O1	52
O2	193
O3	152
O4	70
O5	141
O6	43
O7	25
O8	89

2.4.2 Constructing a model

2.4.2.1 Decision variables

The question to answer is “How should the various oils be blended every month?” This means we need to identify how much of each oil goes into the final blend during each month. This is a good start but is is clearly not enough. For instance, we can blend from oil we buy and from oil we have in inventory. So we need to distinguish these two quantities. Moreover, we may decide to buy for storage (because the prices are about to go up) so we also need to know how much we store. This suggests at least three decision variables for each oil ($O = \{0, 1, 2, \ldots, n_o\}$ will be the set of oils), and for each month ($M = \{0, 1, 2, \ldots, n_m\}$ is the set of months)

$$
\begin{aligned}
x_{i,j} &\geq 0 \quad \forall i \in O, \forall j \in M, && \text{buy,}\\
y_{i,j} &\geq 0 \quad \forall i \in O, \forall j \in M, && \text{blend,}\\
z_{i,j} &\geq 0 \quad \forall i \in O, \forall j \in M, && \text{hold.}
\end{aligned}
$$

The interpretation is $x_{i,j}$ will be the number of tons of oil i bought during month j; $y_{i,j}$ will be the number of tons blended into our soap; and $z_{i,j}$ is the number of tons held at the beginning of the month. Note that we have a choice here to have the variable represent the amount at the beginning or at the end of the period. Either is acceptable but it must be clear in the model which one is choosen because it affects the constraints. A typical mistake in a multi-period model is to have some constraints assume that a variable represents a quantity at the start of the period while some other constraints assume the end. The model may run, but the solution will be nonsensical. Since we are given quantities in storage at the beginning of the planning period, having a variable represent the quantity held at beginning means that we can easily initialise it with the given data.

We probably will need to know how much soap we are producing each month. This is not, strictly speaking, essential to the problem as formulated, but it may make the presentation of the solution and maybe the formulation of some constraints much simpler. As usual, it helps to introduce auxiliary variables to clear up some statements. To tally the total production per month,

$$t_j \quad \forall j \in M.$$

2.4.2.2 Constraints

Let us tackle the continuity constraints. We need to specify, for each oil and for each month (but the last), how the inventory fluctuates,

$$z_{i,j} + x_{i,j} - y_{i,j} = z_{i,j+1} \quad \forall i \in O, \forall j \in M \setminus \{n_m\}. \tag{2.5}$$

In words, what is held at the beginning of the month plus what we buy minus what we blend forms the new inventory.

We have a minimum and a maximum storage capacity, at each month, of the total amount of oil, or

$$C_{min} \leq \sum_i z_{i,j} \leq C_{max} \quad \forall j \in M.$$

Now comes the blending constraint or rather constraints since we need to target a number of fatty acids. To help the formulation, let us extract the total production,

$$t_j = \sum_i y_{i,j} \quad \forall j \in M.$$

Let us assume that for each acid $k \in A$ we have a target range $[l_k, u_k]$ and that each oil $i \in O$, a percentage $p_{i,k}$ of the required acid (Table 2.10). Since the final product, for each acid, must fall in a certain range, we should have two constraints, one for the low end and one for the high end of the interval. That is,

$$\sum_i y_{i,j} p_{i,k} \geq l_k t_j \qquad \forall k \in A, \forall j \in M, \tag{2.6}$$

$$\sum_i y_{i,j} p_{i,k} \leq u_k t_j \qquad \forall k \in A, \forall j \in M. \tag{2.7}$$

These constraints could be written without the production variables t_j but would be more cumbersome and difficult to read.

Finally we need to satisfy demand. This is simple, assuming a demand of D_j at each month j,

$$t_j \geq D_j \quad \forall j \in M.$$

2.4.2.3 Objective function

We are told that the objective is to minimize costs, comprised of the varying oil costs at each month plus the fixed storage cost of the oils we keep in inventory. Therefore,

$$\sum_i \sum_j x_{i,j} P_{i,j} + \sum_i \sum_j z_{i,j} p.$$

This type of objective (fixed plus variable cost) appears regularly in business type problems. We will see this again when considering facility location to service customer demands. The decision to build incurs a fixed cost. The servicing of the various customers is a variable cost.

2.4.2.4 Executable model

Let us now translate this into executable Code 2.6. There is a fair amount of data to pass in. Let us assume arrays `Part` as in Table 2.9, `Target` as in Table 2.10, `Cost` as in 2.11 and `Inventory` as in Table 2.12 in addition to three parameters, `D` in tons for the demand, `SC` in dollars per ton for the storage cost and `SL` in tons for the mininum and maximum to hold in inventory.

From line 5 to line 11 we declare variables but only the first three are true decision variables. All the others are artificially introduced either to help us state the constraints (for 8 and 11) or to help us display some details of the resulting solutions. They will not affect the running time of the solver in any appreciable manner but will make our life easier.

At line 12 we set the `Hold` variable to contain what is known to be in the inventory at the start of the planning period.

The large loop starting at line 14 will set all the constraints since they have the identical structure for each month and we have declared our variables to be arrays indexed by the month.

The line 15 sets the artificial variable `Prod` to be the sum of the blended oils. This is not really a constraint, but rather a simplifying trick. If we repeat some calculations in a model as here,

```
sum(Blnd[i][j] for i in range(nO)
```

we should consider introducing an artificial variable. Assuming a decent solver, it will cost nothing and is likely to help. One of the principles of programming (and modeling) is "Do not repeat yourself".

We use this `Prod` variable immediately after, at line 16 to ensure that we satisfy the demand. If this demand is a scalar, we set it identically for each month, but it could be an array indexed by the month.

The code starting with the `if` on line 17 implements the continuity requirement we described at Equation (2.5). We ensure that what we buy and what we have on hand at the beginning of the month equals what we blend and what we store for the next month. The conditional is to avoid setting a constraint on a month past the planning horizon.

The two lines 20 and 21 ensure the bounds on the oils we keep in inventory.

The loop starting at line 22 first defines our auxiliary `Acid` variable to ease the formulation of the blending constraints stated on the following two lines which correspond to Equations (2.6) and (2.7). `Acid`, indexed by the ordinal of the fatty acid `k` and of the month j under consideration, is summed over all oils of the quantity blended with the oil's percentage of acid k. This quantity, divided by the total blended, will be the precentage that must fall within the required range.

Finally the four lines starting at 26 set the artificial variables that will hold the costs of purchasing and holding at each period and then sum them to construct the objective function which we will minimize.

Code 2.6: Multi-period blending model.

```
1  def solve_model(Part,Target,Cost,Inventory,D,SC,SL):
2    s = newSolver('Multi-period soap blending problem')
3    Oils= range(len(Part))
4    Periods, Acids = range(len(Cost[0])), range(len(Part[0]))
5    Buy = [[s.NumVar(0,D,'') for _ in Periods] for _ in Oils]
6    Blnd = [[s.NumVar(0,D,'') for _ in Periods] for _ in Oils]
7    Hold = [[s.NumVar(0,D,'') for _ in Periods] for _ in Oils]
8    Prod = [s.NumVar(0,D,'') for _ in Periods]
9    CostP= [s.NumVar(0,D*1000,'') for _ in Periods]
10   CostS= [s.NumVar(0,D*1000,'') for _ in Periods]
11   Acid = [[s.NumVar(0,D*D,'') for _ in Periods] for _ in Acids]
12   for i in Oils:
13     s.Add(Hold[i][0] == Inventory[i][0])
14   for j in Periods:
15     s.Add(Prod[j] == sum(Blnd[i][j] for i in Oils))
16     s.Add(Prod[j] >= D)
17     if j < Periods[-1]:
18       for i in Oils:
19         s.Add(Hold[i][j] + Buy[i][j] - Blnd[i][j] == Hold[i][j+1])
20     s.Add(sum(Hold[i][j] for i in Oils) >= SL[0])
21     s.Add(sum(Hold[i][j] for i in Oils) <= SL[1])
22     for k in Acids:
23       s.Add(Acid[k][j] == sum(Blnd[i][j]*Part[i][k] for i in Oils))
24       s.Add(Acid[k][j] >= Target[0][k] * Prod[j])
25       s.Add(Acid[k][j] <= Target[1][k] * Prod[j])
26     s.Add(CostP[j] == sum(Buy[i][j] * Cost[i][j] for i in Oils))
27     s.Add(CostS[j] == sum(Hold[i][j] * SC for i in Oils))
28   Cost_product = s.Sum(CostP[j] for j in Periods)
29   Cost_storage = s.Sum(CostS[j] for j in Periods)
30   s.Minimize(Cost_product+Cost_storage)
31   rc = s.Solve()
32   B,L,H,A = SolVal(Buy),SolVal(Blnd),SolVal(Hold),SolVal(Acid)
33   CP,CS,P = SolVal(CostP),SolVal(CostS),SolVal(Prod)
34   return rc,ObjVal(s),B,L,H,P,A,CP,CS
```

Since this model is of a certain complexity, the caller should examine the return code of the solver. It needs to be zero for the solution to be optimal. The most frequent non-zero return status will be for infeasibility. This may occur for a number of reasons, the most likely of which is that there is no combination of oil that will achieve our target fatty acid content.

The results of a run with all the above data is displayed in Table 2.13. It displays everything we need to know. The first set of lines, to be sent to Purchasing, specify how much of each oil to buy per month. The next set of lines, to be sent to Manufacturing, describe the exact recipe of the blending to do each month. Notice that the soap is created from different oils at each month to achieve the minimal cost. The next set of lines, to be sent to the Bean Counters, describes the inventory, the product costs and the storage costs at each month. And finally, we can send to Quality Control the last set of lines, indicating the actual percentages of fatty acids achieved by the blending recipe.

The main point of this model is to present some of real models' complexity along with some tricks on managing this complexity at the model level. A second point is to highlight some of the advantages of modeling in Python instead of in specialized modeling languages.

2.4.3 Variations

There is an infinite number of variations of such a complex model.

- The demand could vary at each month.
- Instead of satisfying some demand, we may be asked maximize profit. In this case we need to know the price of the final product, which, of course, may change at each month.
- The inventory levels may be stated in terms of each oil instead of agregate quantities.
- There may be uncertainty in the oils' fatty acid content.

Table 2.13: Multi-period blending results.

Buy qty	Month 0	Month 1	Month 2	Month 3	Month 4
O0	1935.7	0.0	0.0	0.0	0.0
O1	480.7	0.0	274.6	0.0	0.0
O2	192.4	0.0	545.9	0.0	0.0
O3	2835.0	1553.3	0.0	0.0	0.0
O4	293.7	0.0	0.0	136.8	0.0
O5	0.0	966.7	1611.3	0.0	0.0
O6	482.6	1011.5	275.1	1517.9	0.0
O7	0.0	0.0	0.0	1247.9	0.0
O8	0.0	1468.5	2293.1	597.4	0.0
Blend qty	Month 0	Month 1	Month 2	Month 3	Month 4
O0	1683.6	117.7	149.4	0.0	2034.4
O1	532.7	0.0	274.6	0.0	919.5
O2	113.3	272.1	269.3	276.6	105.6
O3	1551.3	1465.1	1524.0	0.0	382.6
O4	363.7	0.0	0.0	136.8	392.7
O5	141.0	966.7	1051.8	559.5	0.0
O6	525.6	684.9	601.7	1517.9	1165.2
O7	0.0	25.0	0.0	747.9	0.0
O8	89.0	1468.5	1129.2	1761.3	0.0
Hold qty	Month 0	Month 1	Month 2	Month 3	Month 4
O0	15.0	267.2	149.4	0.0	0.0
O1	52.0	0.0	0.0	0.0	0.0
O2	193.0	272.1	0.0	276.6	0.0
O3	152.0	1435.7	1524.0	0.0	0.0
O4	70.0	0.0	0.0	0.0	0.0
O5	141.0	0.0	0.0	559.5	0.0
O6	43.0	0.0	326.6	0.0	0.0
O7	25.0	25.0	0.0	0.0	500.0
O8	89.0	0.0	0.0	1163.9	0.0
Prod qty	5000.0	5000.0	5000.0	5000.0	5000.0
P. Cost	$735098.96	$616064.04	$644688.93	$491829.66	$0.00
S. Cost	$3900.00	$10000.00	$10000.00	$10000.00	$2500.00
Acid %	Month 0	Month 1	Month 2	Month 3	Month 4
A0	$13.60	$13.25	$13.25	$14.65	$14.65
A1	$24.90	$24.55	$25.25	$25.47	$23.22
A2	$17.79	$18.50	$17.79	$17.79	$19.67
A3	$3.73	$3.73	$3.73	$3.73	$4.07
A4	$5.04	$5.04	$5.04	$5.04	$5.03
A5	$8.80	$8.80	$8.80	$9.67	$9.72
A6	$26.13	$26.13	$26.13	$23.65	$23.65
Total	$100.00	$100.00	$100.00	$100.00	$100.00

Chapter 3

Hidden linear continuous models

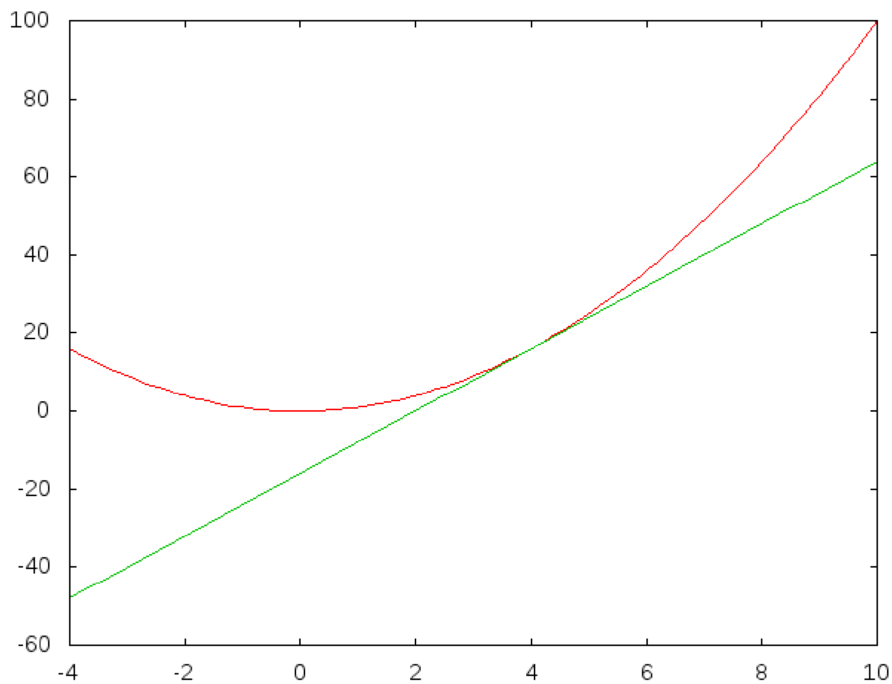
In this chapter we do violence to some problems to reveal their inner structure. The focus is on problems which, at first glance, may not seem to be of the continuous linear variety yet can be marshalled into that form with a handful of creative alterations. The key is to ensure a one-to-one correspondence between the original and the altered problems so that we can retrieve a solution to the original from a solution to the alteration.

The main reason for massaging problems in this way is that continuous linear solvers have become so fast that they can handle models with hundreds of thousands of variables and constraints. Therefore, if a problem can be modeled in that manner, there is little practical limit on the instance size that can be solved. As we will see later, this is not the case with more complex models. In fact we can write models with a few dozen variables that no current solver can solve in a reasonable time.

The main obstacles encountered in this chapter are non-linearities, of one kind or another, but with the advantageous restriction that the functions considered be convex. A convex function[1] is one that sits "above" any valid linear approximation to it. In one dimension, algebraically, f is convex at a point x_0 if

$$f(x_0 + h) \geq f(x_0) + f'(x_0)h.$$

Geometrically, it looks like this, with a first-order approximation of $f(x) = x^2$ at $x_0 = 4$:



Convexity will be the Trojan horse used to beat the non-linearity into submission.

[1]All research mathematicians agree on the labels "convex" and its opposite "concave", but textbook authors for high schools in the US, ignoring thousands of papers, journals and research monographs, insists on "concave up" and "concave down".

3.1 Piecewise linear

We consider here broken-up linear functions. In the traditional parlance, they are *piecewise linear*. As such, the linear programming solvers we have used up to now (GLPK, GLOP, CLP) cannot handle them directly, but a little coding on our part will morph them into a standard form that all solvers can handle. This is one of the good reasons to code models in Python instead of the specialized modeling languages.

A first example, to illustrate the technique without any side issue that might hide the essence, let us consider a piecewise function defined as

$$f(x) = \begin{cases} c_1 x & 0 \leq x \leq B_1, \\ c_1 B_1 + c_2(x - B_1) & B_1 \leq x \leq B_2, \\ c_1 B_1 + c_2(B_2 - B_1) + c_3(x - B2) & B_2 \leq x \leq B_3, \\ \dots \end{cases} \tag{3.1}$$

We can think of this function as a shipping cost function with additional penalties for weights; that is, the more product we ship, the more expensive is each unit. Table 3.1 is an instance of this simple function where the first two columns bracket the quantities $(B_i, B_{i+1}]$ for which the third column is the unit cost, c_i.

We will illustrate the approach by minimizing this function subject to a simple bound on the quantity.

Table 3.1: Example of piecewise function.

(From	To]	Unit cost	(Total cost	Total cost]
0	148	24	0	3552
148	310	28	3552	8088
310	501	32	8088	14200
501	617	34	14200	18144
617	762	36	18144	23364
762	959	40	23364	31244

3.1.1 Constructing a model

What we need to decide in this problem is simply the quantity to produce. We can define a decision variable with bounds from `0` to the last quantity in the table,

$$x \in [0, B_n].$$

But the quantity we decide on will affect the objective function, therefore we need to know in which bracket we end up, and where on that bracket. Here is the key trick : we introduce additional variables, one for each breakpoint in the function. Assuming that we have n brackets, we introduce

$$\delta_i \in [0, 1] \quad \forall i \in \{0, \dots, n\}.$$

Conceptually, we consider these variables as weights on the bracket boundaries, telling us where we are in the bracket. We want at most two consecutive variables to be non-zero, with their sum to be one. This will tell us where x lies and, consequently, what is the objective function value.

For example, if $\delta_2 = \frac{1}{4}$ and $\delta_3 = \frac{3}{4}$ we know that we are in the third bracket, one quarter of the way and that $x = \delta_2 B_2 + \delta_3 B_3$.

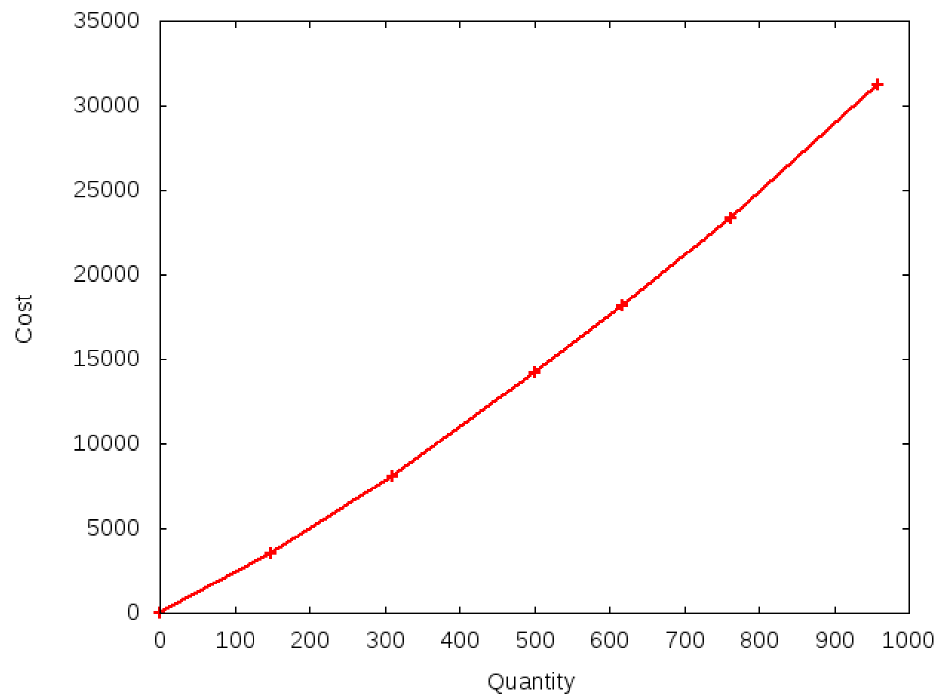


Figure 3.1: Piecewise convex cost function.

3.1.1.1 Constraints

We will enforce that the δ sum to one, and, by the convex structure of the problem, at most two, adjacent δ, will be non-zero. This will tell us which bracket and where in the bracket. To do this we add the constraint

$$\sum_i \delta_i = 1.$$

We deduce the value of the decision variable by

$$x = \sum_i \delta_i B_i. \tag{3.2}$$

Note that this x variable and its associated constraint play no role in the optimization model. For the solver, the δ are the real decision variable and the x is simply a translation into the language of the original problem. This is the key.

3.1.1.2 Objective

The objective function is linear within a bracket therefore we sum over all brackets,

$$\min \sum_{i=1}^{n} \delta_i \sum_{j=1}^{i} (B_j - B_{j-1}) \times C_{j-1}.$$

We must stress here that the transformation trick only works because of the structure of this objective function. It is convex. Had it been concave, the problem would not have been solvable by a Linear Programming solver. We will see in a later chapter (6.5) how to use an Integer Programming solver to handle this more difficult case.

3.1.1.3 Executable model

Let us translate this into an executable model. First assume that the objective function is described by an array D of tuples $(x, f(x))$. This allows us to consider any continuous piecewise linear function. Assume that we are also given a lower bound b for the quantity to produce. This problem is so simple that we know what the solution will be, namely, the lower bound b. But this is meant to illustrate the technique used to solve piecewise linear function using a linear solver. In the next section we will use this technique on a more realistic problem.

Code 3.1: Piecewise model (simplest example).

```
1  def minimize_piecewise_linear_convex(Points,B):
2      s,n = newSolver('Piecewise'),len(Points)
3      x = s.NumVar(Points[0][0],Points[n-1][0],'x')
4      l = [s.NumVar(0.0,1,'l[%i]' % (i,)) for i in range(n)]
5      s.Add(1 == sum(l[i] for i in range(n)))
6      s.Add(x == sum(l[i]*Points[i][0] for i in range(n)))
7      s.Add(x >= B)
8      Cost = s.Sum(l[i]*Points[i][1] for i in range(n))
9      s.Minimize(Cost)
10     s.Solve()
11     R = [l[i].SolutionValue() for i in range(n)]
12     return R
```

Line 4 defines our additional variables, one for each breakpoint of the piecewise function. We force that the sum of those be one at line 5. The definition of x at line 6 and its simple bound at 7 will allow us to consider various interesting scenarios.

The objective function is handle in a similar fashion to x at line 8. We solve and return the solution in a table with all the appropriate information to understand what the solver produced.

We will run this code with various bounds to illustrate the types of solution produced. First, at Table 3.7 we see a typical run with a solution within a bracket. We set a bound of $x \geq 250$, which is exactly the value obtained. Note that only two δ are non-zero and that

Table 3.2: Optimal solution to convex piecewise objective with $x \geq 250$.

Interval	0	1	2	3	4	5	6	Solution
δ_i	0.0	0.3704	0.6296	0.0	0.0	0.0	0.0	$\sum \delta$=1.0
x_i	0	148	310	501	617	762	959	x=250.0
$f(x_i)$	0	3552	8088	14200	18144	23364	31244	Cost=6408

$$\delta_1 \times B_1 + \delta_2 \times B_2 = 0.37 \times 148 + 0.63 \times 310 = 250,$$

while the cost function is

$$0.37 \times 3552 + 0.63 \times 8088 = 6408.$$

Table 3.3: Optimal solution to convex piecewise objective with $x \geq 310$.

Interval	0	1	2	3	4	5	6	Solution
δ_i	0.0	0.0	1.0	0.0	0.0	0.0	0.0	$\sum \delta$=1.0
x_i	0	148	310	501	617	762	959	x=310.0
$f(x_i)$	0	3552	8088	14200	18144	23364	31244	Cost=8088

To illustrate a boundary case, let us set a bound of $x \geq 310$, the beginning of a bracket, and observe the result in Table 3.3. Notice that only one δ is non-zero in this case and is set at the maximum weight of one.

As a final example of a boundary case, let us force $x \geq 1$. The result is in Table 3.4.

Table 3.4: Optimal solution to convex piecewise objective with $x \geq 1$.

Interval	0	1	2	3	4	5	6	Solution
δ_i	0.9932	0.0068	0.0	0.0	0.0	0.0	0.0	$\sum \delta$=1.0
x_i	0	148	310	501	617	762	959	x=1.0
$f(x_i)$	0	3552	8088	14200	18144	23364	31244	Cost=24

3.1.2 Variations

The first variation is an application of the piecewise approach to non-linear optimization.

3.1.2.1 Non-linear function minimization via linear approximations

Since we can solve optimization problems with piecewise linear functions, we can use this approach to approximate convex non-linear functions with piecewise linear functions of increasing accuracy. Here is an example. Say we need to minimize, on the interval $[2, 8]$, the nonlinear function

$$f(x) = \sin(x)e^x$$

We can easily decompose this function into segments on which we interpolate linearly between function values, as in Figure 3.2

We then minimize this piecewise linear approximation. If the solution is accurate enough for our needs, we are done. If not, we zoom in around the solution, and approximate the function again using smaller segments. The executable code is shown at Code 3.2 and is one more instance where using a general-purpose programming language like Python clearly wins over special-purpose modeling languages.

The function `minimize_non_linear` accepts as parameters any Python function, along with an interval of values over which to minimize and a desired precision. At line 4 we compute the length of each sub-interval and we construct a piecewise description of the given function at line 5 which we use as a parameter to our previously described solver (Code 3.1).

The lines 9 and 11 zoom in of the appropriate sub-interval, which becomes the new interval to be sub-divided. The process stops when the interval is smaller than the required precision. In ten very simple lines of codes we leverage the power of a linear solver to minimize non-linear convex functions.

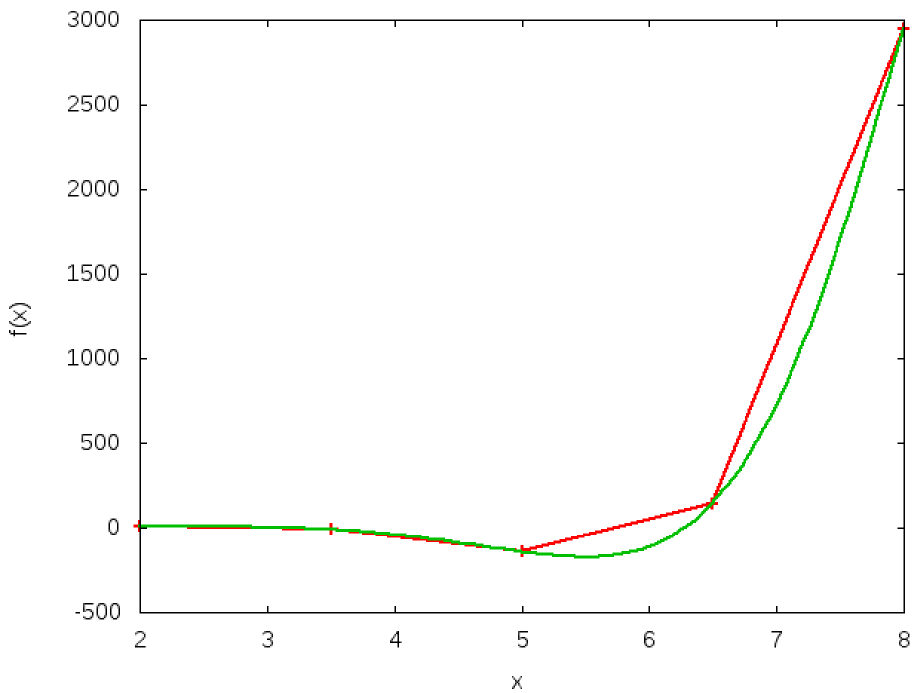


Figure 3.2: Piecewise approximation of a non-linear function.

Code 3.2: Minimizing non-linear functions via piecewise linear approximations.

```
def minimize_non_linear(my_function,left,right,precision):
  n = 5
  while right-left > precision:
    dta = (right - left)/(n-1.0)
    points = [(left+dta*i, my_function(left+dta*i)) for i in range(n)]
    G = minimize_piecewise_linear_convex(points,left)
    x = sum([G[i]*points[i][0] for i in range(n)])
    left = points[max(0,[i-1 for i in range(n) \
                     if G[i]>0][0])][0]
    right = points[min(n-1,[i+1 for i in range(n-1,0,-1) \
                       if G[i]>0][0])][0]
  return x.SolutionValue()
```

We can see the increasing accuracy of the solution in Table 3.5. Each set of three consecutive rows represents the breakpoints in `x`, the value of the function at those points, and the interval parameter `delta`, indicating the optimal bracket. The rightmost two columns are the corresponding optimal `x` and `f(x)`. We note that `x` jumps alternatively under and above the final solution, which can be important if one requires an under or over-estimate.

Table 3.5: Optimal solution to non-linear minimization.

Interval	0	1	2	3	4	x^*	$f(x^*)$
x_i	2.0	3.5	5.0	6.5	8.0		
$f(x_i)$	6.7	-11.6	-142.3	143.1	2949.2		
δ_i	0.0	0.0	1.0	0.0	0.0	5.0	-142.3
x_i	3.5	4.2	5.0	5.8	6.5		
$f(x_i)$	-11.6	-62.7	-142.3	-159.7	143.1		
δ_i	0.0	0.0	0.0	1.0	0.0	5.8	-159.7
x_i	5.0	5.4	5.8	6.1	6.5		
$f(x_i)$	-142.3	-170.2	-159.7	-72.0	143.1		
δ_i	0.0	1.0	0.0	0.0	0.0	5.4	-170.2
x_i	5.0	5.2	5.4	5.6	5.8		
$f(x_i)$	-142.3	-159.2	-170.2	-171.9	-159.7		
δ_i	0.0	0.0	0.0	1.0	0.0	5.6	-171.9
x_i	5.4	5.5	5.6	5.7	5.8		
$f(x_i)$	-170.2	-172.5	-171.9	-167.8	-159.7		
δ_i	0.0	1.0	0.0	0.0	0.0	5.5	-172.5
x_i	5.4	5.4	5.5	5.5	5.6		
$f(x_i)$	-170.2	-171.7	-172.5	-172.6	-171.9		
δ_i	0.0	0.0	0.0	1.0	0.0	5.5	-172.6
x_i	5.4	5.5	5.5	5.5	5.6		
$f(x_i)$	-171.7	-172.4	-172.6	-172.5	-171.9		
δ_i	0.0	0.0	1.0	0.0	0.0	5.5	-172.6
x_i	5.5	5.5	5.5	5.5	5.5		
$f(x_i)$	-172.4	-172.5	-172.6	-172.6	-172.5		
δ_i	0.0	0.0	1.0	0.0	0.0	5.5	-172.6

3.1.2.2 Non-convex piecewise linear

The most vexing situation occurs when the function to minimize is non-convex. For example, if the unit cost went decreasing as in Table 3.6 instead of increasing, then the technique presented in this section will fail, as we can see in Table 3.7.

Table 3.6: Example of non-convex piecewise function.

(From	To]	Unit cost	(Total cost	Total cost]
0	194	18	0	3492
194	376	16	3492	6404
376	524	14	6404	8476
524	678	13	8476	10478
678	820	11	10478	12040
820	924	6	12040	12664

Notice that the sum of the δ is one and the value of the decision variable is correct, but the total cost is nonsensical. It is obtained by a combination of the first and last δ, non-consecutive points. What is happening is that the solver is considering the straight line between $f(0)$ and $f(924)$; this line is below $f(x)$ hence produces a lower cost value for all intermediate values of x.

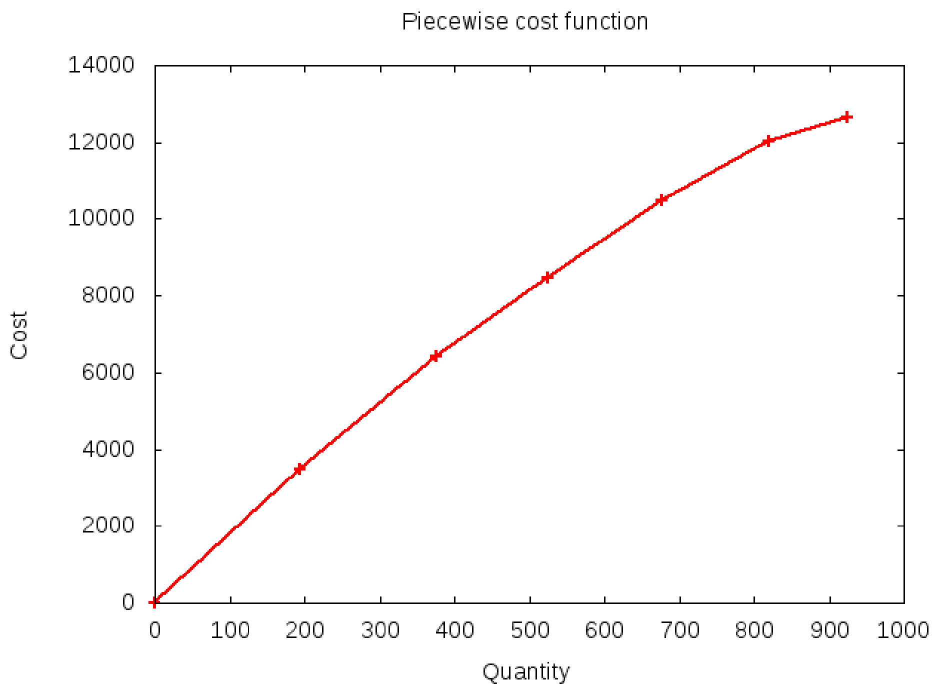


Figure 3.3: Piecewise non-convex cost function.

Table 3.7: Incorrect solution to non-convex objective with $x \geq 250$.

Interval	0	1	2	3	4	5	6	Solution
δ_i	0.7294	0.0	0.0	0.0	0.0	0.0	0.2706	$\sum \delta$=1.0
x_i	0	194	376	524	678	820	924	x=250.0
$f(x_i)$	0	3492	6404	8476	10478	12040	12664	Cost=3426

As we will see in a later section (6.5), all is not lost. The approach taken here can be amended to use an Integer Solver, adding a few more constraints.

3.2 Curve fitting

A very common problem is to move from a set of data points to an analytic representation of the same data. Statisticians call this *Regression*; applied mathematicians utter *Parameter estimation*; engineers usually speak of *Curve fitting*. I prefer the latter expression.[2]

Its most famous and simplest example is illustrated by the following: Imagine that we know (or, conjecture, as Galileo was the first to do) that a falling body follows a curve of the form

$$f(t) = a_2 t^2 + a_1 t + a_0$$

where t represent time, but we do not know exactly what are the appropriate values for a_0, a_1, a_2. We run an experiment where we collect the data in Table 3.8.

Table 3.8: Example of data to fit on a quadratic $f(t) = a_2 t^2 + a_1 t + a_0$.

t_i	f_i
0.1584	0.0946
0.8454	0.2689
2.1017	5.8285
3.1966	14.8898
4.056	25.6134
4.9931	38.3952
5.8574	43.5065
7.1474	91.3715
8.1859	119.075
9.0349	115.7737

Since we need to identify the coefficients of our function (the $a_0, a_1, \ldots$), we need, somehow to minimize a distance from each possible curve to our data points. Statisticians are fond of using the euclidean distance (or, equivalently, its square)

$$\min \sum_n (\bar{f}_i - f(\bar{t}_i))^2$$

This Least-Squares approach dates from Gauss who developed it to predict planetary motion. It often makes sense and is very easy to obtain by solving one system of linear equations, the so-called Normal Equations.

Notwithstanding its popularity, the Euclidean distance is not the only valid distance to minimize. Another one is to use the absolute values of the deviations,

$$\min \sum_n |\bar{f}_i - f(\bar{t}_i)|$$

Or even the largest of the absolute values of the deviations,

$$\min \max_n |\bar{f}_i - f(\bar{t}_i)|$$

This latter approach is the most appropriate one when, for instance, we are dealing with tolerances, that is, where all errors must be within some maximal value. We will develop code which can choose, at runtime, between the latter two objective functions.

[2]The expression "regression" comes from Francis Galton's original paper about "Regression towards mediocrity" and shadows rather than highlights the technique. As for "parameter", what, pray tell, is not a parameter?

3.2.1 Constructing a model

3.2.1.1 Objective function

Let us assume, with some generality, that we are asked to identify a polynomial of degree k in the variable t. The coefficients $a_0, a_1, \ldots, a_k$ are to be determined which minimize either either the sum of deviations or the largest deviation between the data points and the polynomial.

The first abstraction here is to think of all of these deviations as some functions, say $e_1, e_2, \ldots, e_n$ which we will determine later.

In the case of the sum of deviations, the objective is simply

$$\min \sum_i e_i$$

But for the second case, we need an objective that minimizes the maximum deviation:

$$\min \max_n e_i$$

This is clearly not a form that fits our framework of linear programs; we can have a min or a max but not both and we must have one objective function, not a set of them.

To handle this, We introduce a set of inequalities with a new variable, say e, which will represent the maximum deviation:

$$e_1 \leq e \quad \forall i \in [1, n]$$

Then the objective becomes $\min e$. Since e is an upper bound on all the deviations and we minimize it, we minimize the maximum deviation. Note that, at optimality, at least one of the inequalities will be binding (or else we clearly are not optimal) but most will likely be slack since their deviation will be smaller than the maximum deviation.

3.2.1.2 Constraints

We need to express these deviations. We will be given a set of couples $(\bar{t}_i, \bar{f}_i)$ representing the measurement at time $\bar{t}_i$ of the putative function f. The deviation for a particular couple is

$$e_i = |a_0 + a_1\bar{t}_i + a_2\bar{t}_i^2 + \ldots + a_k\bar{t}_i^k - \bar{f}_i|$$

That is, the absolute value of the difference between the experimental $\bar{f}_i$ and the theoretical displacement $f(\bar{t}_i)$ which is obtained by evaluating the function at the time $\bar{t}_i$. Why the absolute value? Because the deviation could be positive or negative and we care only about its magnitude.

In terms of the inequalities we intend to write we would want a constraint of the form $|f(\bar{t}_i) - \bar{f}_i| <= e$ But this is not linear. There are at least two different ways to handle this situation. Which one to use depends on what information we want to extract from the model's solution.

1. Double the inequalities and bound the deviations.

   Consider the definition of absolute value. $|a| = a$ if a is positive and $-a$ if it is negative. This suggests therefore replacing the inequality $|e_i| <= e$ by two inequalities:

   $$|f(t_i) - \bar{f}_i| <= e$$
   $$| - f(\bar{t}_i) + \bar{f}_i| <= e$$

   This is a workable approach.

2. Double the variables and find each deviation.

   Note that the above model will not find the deviation at each point. We simply have a bound of all deviations, a bound which we minimize. What if we want to know each deviation; for instance, to minimize their sum?

   One way to find the value of each deviation is to introduce two non-negative variables for each point $(\bar{t}_i, \bar{f}_i)$. Let us call them u_i and v_i and introduce the following equality:

   $$f(t_i) - u_i + v_i = \bar{f}_i \tag{3.3}$$

   Notice that since the new variables are non-negative only one of them will be non-zero per equality. That one will equal the deviation, i.e. the difference between the experimental point and the theoretical point.

   If we want to minimize the sum of deviations, we minimize the sum of all u_i and v_i. If we want to minimize the maximum deviation we add the following inequalities:

   $$u_i \leq e$$
   $$v_i \leq e$$

   and minimize e. This approach increases the number of variables instead of the number of constraints.

3.2.1.3 Executable model

Let us translate this into an executable model seen in Code 3.3. Assuming that we obtain the data in an array of tuples $(\bar{t}_i, \bar{f}_i)$ named `D`, along with the degree of the polynomial required and an indicator of the distance to minimize (0 for the sum and 1 for the maximum).

Code 3.3: Polynomial Curve fitting model

```
def solve_model(D,deg=1,objective=0):
  s,n = newSolver('Polynomial␣fitting'),len(D)
  b = s.infinity()
  a = [s.NumVar(-b,b,'a[%i]' % i) for i in range(1+deg)]
  u = [s.NumVar(0,b,'u[%i]' % i) for i in range(n)]
  v = [s.NumVar(0,b,'v[%i]' % i) for i in range(n)]
  e = s.NumVar(0,b,'e')
  for i in range(n):
    s.Add(D[i][1]==u[i]-v[i]+sum(a[j]*D[i][0]**j \
                               for j in range(1+deg)))
  for i in range(n):
    s.Add(u[i] <= e)
    s.Add(v[i] <= e)
  if objective:
    Cost = e
  else:
    Cost = sum(u[i]+v[i] for i in range(n))
  s.Minimize(Cost)
  rc = s.Solve()
  return rc,ObjVal(s),SolVal(a)
```

Line 4 defines the real decision variables, the coefficients of the polynomial. Since we cannot easily set bounds on the coefficients, we use infinity. The next two lines 5 and 6 define the deviations between the data points and the corresponding theoretical values. This is used at line 8 which corresponds to (3.3). Then we bound the deviations at line 11 by our maximum error variable defined at line 7.

The last element is the choice of objective function. The user can select to minimize the maximum deviation at 15 or the sum of deviations at line 17. These are displayed in Table 3.9 under the headings, respectively of e_i^{max} and e_i^{sum}.

Table 3.9: Optimal solution to curve fitting problem

t_i	f_i	$f_{sum}(t_i)$	e_i^{sum}	$f_{max}(t_i)$	e_i^{max}
0.1584	0.0946	-0.4382	0.5328	-12.4063	12.5008
0.8454	0.2689	0.3421	0.0731	-8.3251	8.594
2.1017	5.8285	5.8285	0.0	2.0924	3.7362
3.1966	14.8898	14.8898	0.0	14.285	0.6047
4.056	25.6134	24.7951	0.8184	25.8879	0.2744
4.9931	38.3952	38.3952	0.0	40.5766	2.1814
5.8574	43.5065	53.5269	10.0204	56.0073	12.5008
7.1474	91.3715	80.7311	10.6403	82.3995	8.972
8.1859	119.075	106.6547	12.4203	106.5742	12.5008
9.0349	115.7737	130.5102	14.7365	128.2745	12.5008

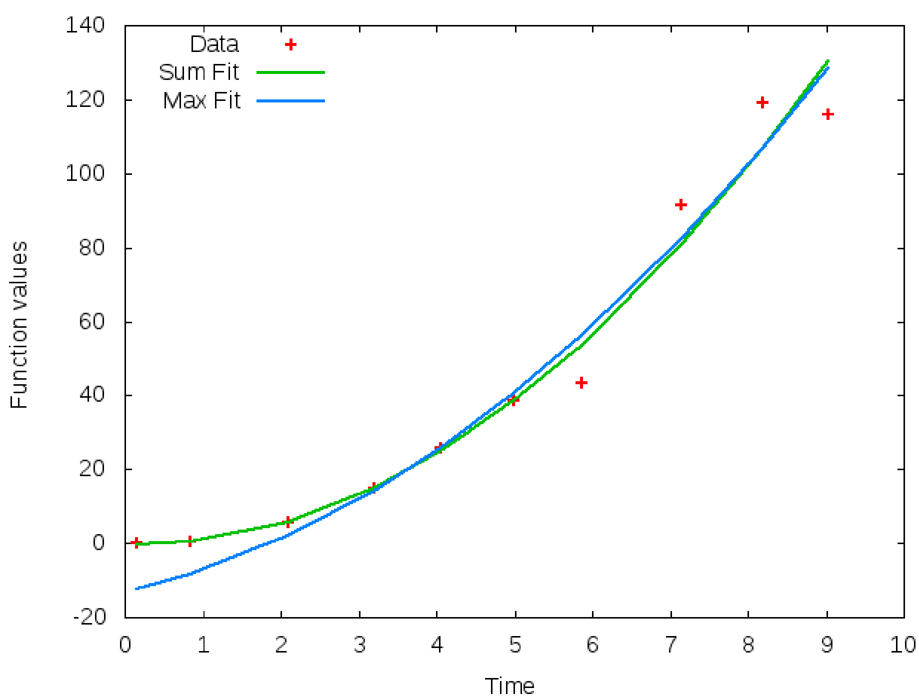


Figure 3.4: Data and fitted curves

The data points, along with both solutions (one for minimizing the maximum deviation and one for minimizing the sum of deviation) are displayed at Figure 3.4.

3.2.2 Variations

The example above is a special case of a technique very useful in practice, the so-called handling of *soft constraints*. Often we would like an equation to be satisfied but we know that it is unlikely to be. Examples abound. Here is one: constructing a model of a scheduling system to produce student schedules at school. In the intelligent manner, we take all the student course choices and their days of availability (I have to work Fridays, so no class then. Or, I work nights; I need day classes only). From all these we wish to construct schedules working for all students.

Unfortunately, it is unlikely that a schedule accommodating all requests is feasible. The best one can hope for is to satisfy as many students' requests as possible. These become soft constraints and the technique is similar to the one above: we introduce new variables (like the u_i and v_i above) to measure the distance to the ideal (the number of students with unsatisfied requests) and minimize the sum of these.

So, if we aim to satisfy, say

$$a_1x_1 + a_2x_2 + ... + a_nx_n = b$$

But we know that this is unlikely, we change to

$$a_1x_i + a_ix_2 + ... + a_nx_n + u - v = b$$

where u and v are non-negative. Then add $(u + v)$ to the objective function (assuming this is a minimization problem).

Note that we may need only one of u and v if we already know that the left hand side will always be either too high or too low with respect to the right hand side. We need both only when it can deviate in both directions, as for our curve fitting example.

Chapter 4

Linear pseudo discrete models

The next class of problems we encounter is linear but not continuous; the solution must be integral. For instance, imagine a complex problem involving the flight of a space shuttle. The constraints involve weight, amount of fuel, amount of oxygen, work to do while in orbit, etc. Thousands, maybe hundred of thousands of variables and constraints. If such a problem is asking "How many astronauts to carry in the shuttle?" It is unlikely that NASA (or SpaceX) would accept "Two and a half astronauts" as the optimal answer.

Before going any further, let us dispense with the tempting and wrong work-around. **It is rarely the case that rounding helps.** If we round a fractional solution many, maybe even all constraints might be violated. Round up the solution of our space shuttle instance, and the weight constraint might prevent lift-off; round down, and the astronauts may fail to accomplish all the required tasks. To be fair, there are problems where rounding is acceptable, but those are either boring, or the solution is obvious or both.

Onward to the easy yet famous cases.

4.1 Maximum flow

Here is a first class of problems where the integrality of the solution is guaranteed "for free". We have nothing to do, except to recognize that the problem falls in that special category and be merry. The goal of this section is to recognize and model problems with this special structure. The prototypical, overt example, is the *network maximum flow* problem where some substance *flows* from some source(s) to some destinations(s) on capacitated channels and we try to maximize the amount flowing.

The substance *flowing* does not have to be material, as water, oil or even electricity. It could be data packets flowing through a network of fiber optics cables. Imagine, for instance, that you are trying to establish how many concurrent video streams you can send from your servers to your viewers. This fits nicely in the context of a maximum flow problem.

To consider the simplest problem abstractly, let us assume a network as described by Figure 4.1 where each arc has the noted capacity and we are trying to send as much as possible from the nodes marked as sources (-S), to the nodes marked as sinks (-T).

4.1.1 Constructing a model

What we need to decide in this problem is the amount to deliver from the source to the sink and through which arcs of the network.

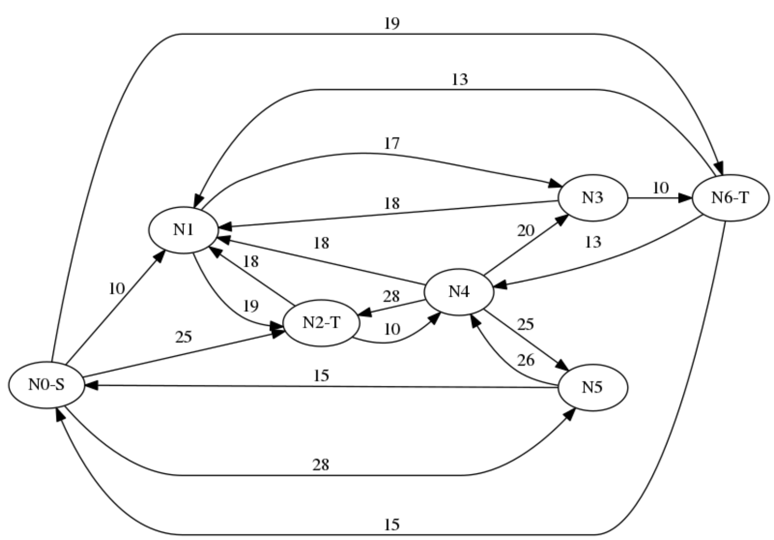


Figure 4.1: Visual representation of network flow problem instance.

4.1.1.1 Decision variables

The simplest, most natural way to answer the question is is to introduce a two-dimensional variable. The first dimension indicates the origin and the second dimension indicates the destination; the value of the variable will be the amount of substance flowing on the arc,

$$x_{i,j} \quad \forall i \in N, \forall j \in N.$$

For example if $x_{2,3} = 35$ it will mean that we should send 35 units from node 2 to node 3.

4.1.1.2 Objective

The objective is to maximize the mount flowing from the sources to the sinks. So the sum of either what is coming out of the sources (say set S) or the sum of what goes into the sinks (say sets T) will work. Since we allow multiple sources and nothing prevents a sink to send to another sink for further transport, we must be careful to maximize the 'net' flow out of the sources,

$$\max \sum_{i \in S} \left(\sum_{j \in N} x_{i,j} - \sum_{j \in N} x_{j,i} \right). \tag{4.1}$$

There may be applications where the flow out of the source is to be maximized; the change to the objective is obvious.

4.1.1.3 Constraints

The first constraint type is known as *conservation of flow*: For every node that is neither a source nor a sink, whatever flow goes in must come out,

$$\sum_{j \in N} x_{i,j} = \sum_{j \in N} x_{j,i} \quad \forall i \in N \setminus \{S \cup T\}. \tag{4.2}$$

In many maximum flow problems where there are only one source, one sink, and no arcs going into the source or out of the sink, the conservation of flow constraints specify the problem perfectly. But, in our slightly more general case, we have two minor annoyances to consider: chained sources and cycles.

The first problematic case is illustrated by Figure 4.2. It is clear that since the capacity of the arc going into the sink is 2, exactly two units will flow. But these two units could come from source `N1-S` or one could come from `N0-S` and a second from `N1-S`.

Figure 4.2: Problematic chaining of sources.

The second problematic case occurs when there is a cycle including a source and a sink. Then, given any flow f around that cycle, if that flow is not as large as the capacity on that cycle, then there is another flow $f + 1$ with exactly the same objective value. Consider Figure 4.3 as an example. An optimal solution could send 10 units from the source to the sink through the intermediary or up to 20 units, with up to 10 flowing back into the source.

Figure 4.3: Problematic cycles.

There is little chance that, for any application, this multitude of optimal flows represents a feature. It will likely be considered a nuisance, especially if two different solvers or two different runs of the same solver return two different flows! How can we ensure that the solver consistently returns the same flow?

One can see this as a *dual objective*: maximize the net flow on the sources, and minimize the inflow into the sources. This idea of a dual objective, or more generally of multiple objectives often occurs in practice and often for the same reason as here: determine, among multiple 'optimal' solution the most preferable one.

Since we need to maximize one object and minimize another, all at one go, we need another trick, the reversal:

$$\max f(x) \iff \min -f(x).$$

We can always replace a minimization problem by a maximization problem, and vice versa. And now we can add the two objectives, maximizing the net flow as in Equation (4.3) and minimizing the inflow, or maximizing $-\sum_{i\in S}\sum_{j\in N} x_{j,i}$. After simplification,

$$\max \sum_{i\in S}\left(\sum_{j\in N} x_{i,j} - 2 * \sum_{j\in N} x_{j,i}\right). \tag{4.3}$$

In the case of our chaining example we would maximize $x_{0,1} + x_{1,2} - 2 * x_{0,1}$, or $x_{1,2} - x_{0,1}$ which would force a solution where all the flow is issued from `N1-S`. In our cyclic example, this would yield $x_{0,1} - 2x_{2,0}$ which guarantees that no flow comes back into the source and it emits 10 units.

4.1.1.4 Executable model

Let us translate this into an executable model. To make the model general enough to solve all problems of this type, we will assume that the data is given in a 2-dimensional array of capacities called `C` The two-dimensional array `C` is indexed by nodes and contains the capacity of the arc between two nodes. We also require an array of sources `S` and of sinks, `T`.

Code 4.1: Maximum flow model.

```
1  def solve_model(C,S,T,net=True):
2    s,n = newSolver('Maximum␣flow␣problem'),len(C)
3    x = [[s.NumVar(0,C[i][j],'') for j in range(n)] for i in range(n)]
4    B = sum(C[i][j] for i in range(n) for j in range(n))
5    Flowout,Flowin = s.NumVar(0,B,''),s.NumVar(0,B,'')
6    for i in range(n):
7      if i not in S and i not in T:
8        s.Add(sum(x[i][j] for j in range(n)) == sum(x[j][i] for j in range(n))
           )
9    s.Add(Flowout == s.Sum(x[i][j] for i in S for j in range(n)))
10   s.Add(Flowin == s.Sum(x[j][i] for i in S for j in range(n)))
11   s.Maximize(Flowout - 2*Flowin if net else Flowout)
12   rc = s.Solve()
13   return rc,SolVal(Flowout),SolVal(Flowin),SolVal(x)
```

Line 3 defines our two-dimensional variable where the first index specifies the origin and the second, the destination.

Line 8 ensures that we conserve flow across nodes that are neither sources nor sinks. The objective function at line 11 computes the total flow and indicates that we should maximize that quantity.

We return the total flow out of the sources and the total flow into the sources. On simple problems the latter will always be zero. On more complicated problems it may be non-zero.

The output of the model is displayed in Table 4.1 where we maximize the flow out of the sources, and Table 4.2 where we maximize the net flow out of the sources.

Table 4.1: Optimal solution to the maximum flow problem (maximize flow out).

82-15	N0-S	N1	N2-T	N3	N4	N5	N6-T
N0-S		10.0	25.0			28.0	19.0
N1			19.0	10.0			
N2-T							
N3							10.0
N4		18.0				13.0	
N5	15.0				26.0		
N6-T		1.0			5.0		

Table 4.2: Optimal solution to the maximum flow problem (maximize net flow).

80-0	N0-S	N1	N2-T	N3	N4	N5	N6-T
N0-S		10.0	25.0			26.0	19.0
N1			19.0	10.0			
N2-T							
N3							10.0
N4		18.0	8.0				
N5					26.0		
N6-T		1.0					

Here is the interesting phenomenon: all solutions are integers, yet we did not impose an integrality constraint. This is a consequence of two elements: First, the structure of the problem, which guarantee that if there is an optimal solution, there is an integral solution[1]. Second the solution technique of all solvers, which will either only consider integer solutions (simplex solvers) or will move from a fractional solution to an integer one (interior-point solvers) before returning to the caller. The reader is encouraged to tweak the numbers to verify that, if the problem is feasible, the solver will find an integral solution.

4.1.2 Variations and applications

- One useful application is to model assignment problems as maximum flow problems. Consider that we have a certain number of workers (they could be people or machines) and a certain number of jobs to accomplish. We construct a network where the workers are sources, the jobs are sinks and there are arcs of capacity one between every worker and every job that the worker is capable of executing. Maximizing the flow will assign workers to jobs optimally.

[1] The theoretically-minded reader will research total unimodularity.

4.2 Minimum cost flow

There is a second class of problems where the integrality of the solution is guaranteed "for free", the *minimum cost flow* problems, *mincost* for short.

Here is the prototypical example: Solar-1138 inc. has a set of clean power plants supplying the needs of multiple cities. Each power plant has a maximum capacity, hence can supply a limited number of kilowatt-hour (kW-h). Each city has a peak demand and all cities peak at roughly the same time. Therefore the sum of peak demands is the quantity that the power plants needs to accommodate. The cost of sending one kW-h from a plant to a city varies according to the plant, the city, the delivery infrastructure and distance between plant and city. This cost has been arrived at by considering production of the power and maintenance of the plant as well as the power lines.

Table 4.3 has the cost of delivery between plant and city, whenever possible, in dollars per kW-h. The maximum supply of each plant as well as the peak demand of each city are in kW-h.

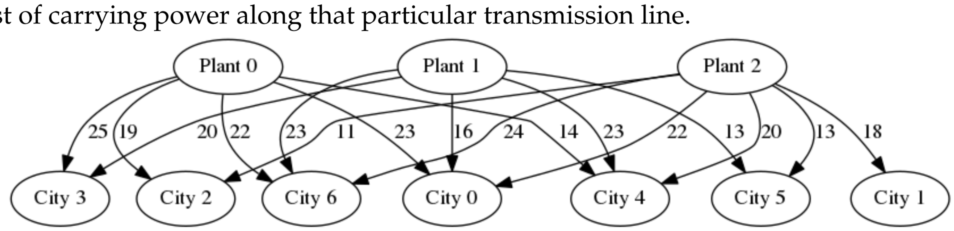
Table 4.3: Example of electrical distribution cost.

From/To	City 0	City 1	City 2	City 3	City 4	City 5	City 6	Supply
Plant 0	23		19	25	14		22	551
Plant 1	16			20	23	13	23	689
Plant 2	22	18	11		20	13	24	634
Demand	288	234	236	231	247	262	281	

The question to answer is "How much power should be sent from each plant to each city, satisfying peak demand while minimizing cost?"

4.2.1 Constructing a model

What we need to decide in this problem is the amount of power to deliver from each plant to each customer. It may be that a plant sends power to none, some or all cities and we need a way to indicate this. As a visual aid, consider the following bipartite[2] graph where plants are the top nodes, cities the bottom and the arcs are power lines annotated by the cost of carrying power along that particular transmission line.



4.2.1.1 Decision variables

With this image in mind, a solution to the problem would be, for each pair of plant-city, the amount of power delivered by plant to city. The simplest, most natural way to do this is to introduce a two-dimensional variable. The first dimension indicates the origin (which

[2]Bipartite means that there will never be any arcs between the top nodes or between the bottom nodes. We will see in the next section a more general problem.

plant of the set of plants P) and the second dimension indicates the destination (which city out of the set of cities C), or

$$x_{i,j} \quad \forall i \in P, \forall j \in C.$$

For example if $x_{2,3} = 35$, it will mean that we should send 35 kW-h from plant 2 to city 3.

These multi-dimensional variables, and we will see many of them, are simply a short-hand notation for one variable per each combination Plant-City. So if we have 3 plants and 4 cities, we really are introducing $3 \cdot 4 = 12$ decision variables. This is somewhat wasteful of memory as there are not always paths between each plant and each city, but we will see how to avoid the waste if it becomes an issue.

4.2.1.2 Objective

The objective is to minimize cost in dollars of delivery. For this we will need a parameter for the cost. Let us assume $C_{i,j}$ indexed exactly as the decision variable. The objective will therefore be

$$\min \sum_i \sum_j C_{i,j} x_{i,j}. \tag{4.4}$$

4.2.1.3 Constraints

The constraints are of two types, closely related, supply and demand. To refer to these we introduce the following parameters $S_i, i \in P$ and $D_j, j \in C$ indicating the supply possible from plant i and the demand required by city j.

Each plant has a maximum production capacity. We need to respect this maximum. So for each plant, we must cap the sum of the power delivered from that plant by an availability constraint, as in

$$\sum_j x_{i,j} \leq S_i \quad \forall i \in P.$$

Notice the inequality; we are not forcing the amount delivered to be at capacity, but only to be at most the supply capacity.

The city demands are similar except that they must be met. Therefore,

$$\sum_i x_{i,j} = D_j \quad \forall j \in C.$$

Here we have an equality constraint. If we make a modeling error and put an inequality, say $\leq$ the optimal solution will be all zero. On the other hand, we could have a $\geq$. In this case, it would change nothing to the solution since we are minimizing the total cost.

4.2.1.4 Executable model

Let us translate this into an executable model. To make the model general enough to solve all problems of this type, we will assume that the costs, demands and supply capacities are given in a 2-dimensional array called `D` structured as Table 4.3, where a zero cost indicates that there are no power lines between that plant and city combination.

In more general terms than "plants" and "cities", we can view each row as representing a producer and each column a consumer, except that the last row represents the demand and the last column the supply. The "product" exchanged between producers and consumers can be anything, not only divisible quantities like kw-h or liters of water, but trucks, flowers, data packets or people. The optimal solution will never contain fractions of people.

Code 4.2: Power distribution model.

```
1  def solve_model(D):
2    s = newSolver('Mincost flow problem')
3    m,n = len(D)-1,len(D[0])-1
4    B = sum([D[-1][j] for j in range(n)])
5    G = [[s.NumVar(0,B if D[i][j] else 0,'') for j in range(n)] \
6        for i in range(m)]
7    for i in range(m):
8      s.Add(D[i][-1] >= sum(G[i][j] for j in range(n)))
9    for j in range(n):
10     s.Add(D[-1][j] == sum(G[i][j] for i in range(m)))
11   Cost = s.Sum(G[i][j]*D[i][j] for i in range(m) for j in range(n))
12   s.Minimize(Cost)
13   rc = s.Solve()
14   return rc,ObjVal(s),SolVal(G)
```

Line 6 defines our two-dimensional variable where the first index specifies the producer and the second, the consumer. Since we know that if a particular producer-consumer pair has no channel between, the entry is zero, we use this to collapse the range of the variable to zero. A good solver will use this information to eliminate these variables before doing any other work.

Line 8 ensures that we supply no more than each plant can produce while line 10 ensures that the peak demands are satisfied. The objective function at line 11 computes the total cost and indicates that we should minimize that quantity.

The output of the model is displayed in Table 4.4. The reader can verify that the total column is below or at the maximum that each plant can produce, while the total row is exactly the required peak demand of each consumer.

Table 4.4: Optimal solution to power distribution problem.

From/To	City 0	City 1	City 2	City 3	City 4	City 5	City 6	Total
Plant 0					247		281	528
Plant 1	288			231		170		689
Plant 2		234	236			92		562
Total	288	234	236	231	247	262	281	

Here is again the interesting phenomenon: all solutions are integers, yet we did not impose any integrality constraints.

4.2.2 Variations

- The simplest variation is to have capacities on the arcs. Then we need to insure that no flow goes above the capacity. Assuming we have the capacities in a matrix A, this is simply a question of adding a constraint of the form

$$x_{i,j} \leq A_{i,j} \quad \forall i \in P, \forall j \in C.$$

- An interesting variation involves spreading the sources. To minimize risks, for instance, we might not want to satisfy more than some fraction of the demand from one source. Say we decide that no city's demand may be satisfied at more than 60 percent from a single source, we could add a constraint of the form

$$x_{i,j} \leq 0.6D_j \quad \forall i \in P, \forall j \in C.$$

The reader is encouraged to add this constraint and note that the optimal value will not be as low as it is without the constraint. Moreover, the solution might not be integral anymore. This simple additional constraints destroys the property that guarantees integrality. We must declare the decision variable integral (with the consequent increase in complexity and solution time) to guarantee an integral solution.

- Instead of material that flows, the problem sometimes appears as an *Assignment* question: Given a set of workers with specific skills and hourly wages and a set of jobs, which worker do you assign to which job to minimize cost?

  For example consider a consulting firm with three teams, based in different cities, as well as three customer at different sites. Since the cost of travel varies from team to customer site, we want to minimize total travel cost. In this case, the demand and supply are simply one since we want one team per customer site and one customer assigned to each team.

	Customer 0	Customer 1	Customer 2	Supply
Team 0	25	30	20	1
Team 1	20	15	35	1
Team 2	18	19	28	1
Demand	1	1	1	

4.3 Transshipment

A more general type of problems that can be modeled as a network flow problem is the *Transshipment* problem. The characteristics of such a problem is a set of nodes with a cost of transporting between each pair, a subset of the nodes are suppliers and another subset are consumers. The remaining nodes can be used to carry the material but neither produce nor consumes, hence the moniker *transshipment*.

The Table 4.5 has, for example, the cost of delivery between each pair of nodes. A blank indicates that there is no path between two nodes. The last column indicates the amount that a node can produce, if any; the last row is the demand of each node, if any. Note that the sum of demands should, in general, be the same as the sum of supplies or else the problem is infeasible.

Table 4.5: Example of transshipment distribution cost over a network.

From/To	N0	N1	N2	N3	N4	N5	N6	N7	Supply
N0					17	10	19		
N1	23			28		23			
N2	29				30	25	25		680
N3					17	15	19	29	
N4		16							
N5	22				25			18	540
N6	25	29	16			22			
N7			30		10		27		
Demand	241			164	239		152	424	

Transshipment problems are often depicted visually as in Figure 4.4, corresponding to the data in Table 4.5, where the arrows are annotated with the cost of transportation of the material and where the nodes contain a positive number indicating supply value and/or a negative number indicating a demand value.

Note that this is clearly a generalization of the mincost flow problem as there could be arcs between any two nodes. Which means, for instance, that a source node could receive whatever product is flowing through the network, add it to its production and send the result off towards another node, whether a consumer or a transshipment node.

4.3.1 Constructing a model

What we need to decide in this problem is the amount of material to deliver from each node with a positive supply to each node with a positive demand. The simplest, most natural way to model this is to introduce a two-dimensional variable. The first dimension indicates the origin and the second dimension indicates the destination. The variable itself will contain the amount to ship. We will assume that N is the set of nodes to get

$$x_{i,j} \quad \forall i \in N, \forall j \in N.$$

For example if $x_{2,3} = 35$ it will mean that we should send 35 units from node 2 to node 3.

4.3.1.1 Objective

The objective is to minimize cost in dollars of delivery. For this we will need a parameter for the cost. Let us assume $C_{i,j}$ indexed exactly as the decision variable. The objective will

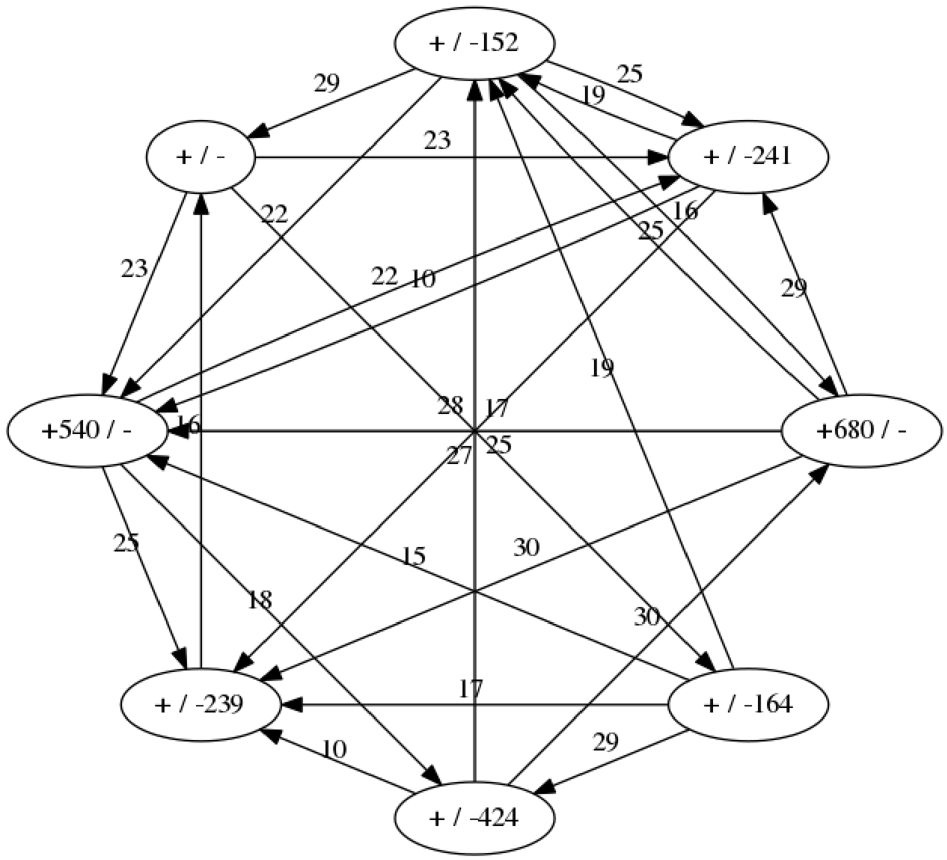


Figure 4.4: Transshipment example viewed graphically.

therefore be

$$\min \sum_i \sum_j C_{i,j} x_{i,j}.$$

4.3.1.2 Constraints

In the previous minimum cost flow problem, we had two types of constraints: one for the producing nodes which stated that the outflow was equal to the supply value; and another one for consuming nodes, stating that the inflow was equal to the demand value. We could use these constraints here, in addition to a third constraint, for intermediate nodes, those without any demand or supply value, stating that the inflow must equal flow.

While it is possible to treat each of types of nodes (producers, consumers and intermediate) differently, it is simpler to notice that one constraint, appropriately general, will hold at each node. Namely, that the flow into the node (f_{in}) minus the flow out of the node (f_{out}) must equal the demand (D) minus the supply (S), or,

$$f_{in} - f_{out} = D - S.$$

We notice that in the case or pure sources and sinks, the equation reduces to the constraints we used in the mincost flow model:

$$-f_{out} = -S \qquad \text{Special case for sources.}$$
$$f_{in} = D \qquad \text{Special case for sinks.}$$

Let us assume that S_i, D_i are respectively the supply and the demand of node i, keeping in mind that, for producing nodes, only supply is non-zero; for consuming nodes, only demand is non-zero; and for intermediate nodes, both are zero. We get

$$\sum_j x_{j,i} - \sum_j x_{i,j} = D_i - S_i \quad \forall i \in N. \tag{4.5}$$

This constraint is known as a general *conservation of flow* constraint. It is the only constraint we need, but it must hold at every node.

4.3.1.3 Executable model

Let us translate this into an executable model. To make the model general enough to solve all problems of this type, we will assume that the costs, demands and supply capacities are given in a 2-dimensional array called `D` structured as Table 4.5. Each entry `{i,j}` represents the cost of transporting from node `i` to node `j`, except that the last row represents the demand and the last column the supply.

Code 4.3: Transshipment distribution model.

```
1  def solve_model(D):
2    s = newSolver('Transshipment␣problem')
3    n = len(D[0])-1
4    B = sum([D[-1][j] for j in range(n)])
5    G = [[s.NumVar(0,B if D[i][j] else 0,'') \
6        for j in range(n)] for i in range(n)]
7    for i in range(n):
8      s.Add(D[i][-1] - D[-1][i] ==
9        sum(G[i][j] for j in range(n)) - sum(G[j][i] for j in range(n)))
10   Cost = s.Sum(G[i][j]*D[i][j] for i in range(n) for j in range(n))
11   s.Minimize(Cost)
12   rc = s.Solve()
13   return rc,ObjVal(s),SolVal(G)
```

Line 6 defines our two-dimensional variable where the first index specifies the producer and the second, the consumer. The range of a variable is from zero to either the total demand or to zero to ensure that we no not use a route that does not exist. In the data `D`, the absence of a cost at entry `i,j` indicates that there are no direct route between `i` and `j`.

The generalized conservation of flow constraint corresponding to (6.5) is implemented on line 7. The objective function at line 10 computes the total cost and indicates that we should minimize that quantity.

The output of the model is displayed in Table 4.6. Note that the solution is, again, entirely integral, even though we did not enforce integrality.

The reader can verify that the difference between the entry in the total column of a given node less the entry in the total row of the same node is equal to the difference of the supply and the demand of that node. This is especially interesting for nodes that receive more than their demands and reroute whatever they do not use. Even for very small problems, those are not solutions can be easily guessed.

Table 4.6: Optimal solution to the transshipment problem.

From/To	N0	N1	N2	N3	N4	N5	N6	N7	Out
N0									
N1				164					164
N2	125				403		152		680
N3									
N4		164							164
N5	116							424	540
N6									
N7									
In	241	164		164	403		152	424	

4.3.2 Variations

- One variation possible is when the supply and demands are not balanced. This could be that the producing nodes have a maximum production capacity that they need not meet; only the demands must be satisfied. In this case, these nodes must be treated separately and, instead of the generalized conservation of flow constraint, we indicate that the flow out less the flow in must be at most the supply,

$$\sum_j x_{i,j} - \sum_j x_{j,i} \le S_i \quad \forall \{i \in N \mid S_i > 0\}.$$

  Nothing else need to change as we are minimizing cost and satisfying demands, we will get the optimal solution.

- The reverse situation is also possible, though unlikely, in which case, we must treat the demand nodes separately and ensure that the flow in less the flow out must be at most the demand value.

- Another simple variation is to have capacities on the arcs, limiting the amount flowing through them. In this case, the additional constraints are, assuming a matrix of capacity C,

$$x_{i,j} \le c_{i,j}.$$

4.4 Shortest paths

We consider now the problem Google faces every time someone asks Google Map to find a path from point A to point B: the shortest path problem (either shortest according to distance or according to time). It may surprise the reader that this too can be modeled and solved very efficiently as a network flow problem.

Here is the abstracted situation: we are given a two-dimensional array of distances between a set of points as exemplified by Table 4.7. This is called the *distance matrix*. These could be distances in thousands of kilometers between cities if we are considering a planetary scale problem or times in minutes between city street intersections if we are looking for a bike path taking into consideration the path elevation. In addition to the array of distances, we could be given a start and an end point but, in their absence, we will assume that the array has been ordered so that we need a path from the first point to the last.

The task is to find a sequence of points between the start and the end that minimizes the corresponding sum of entries in the array. This is called a shortest path, no matter what the units are. Note that we do not say *the* shortest path as there may be many paths with the same shortest total distance. So if we go through the sequence 0,3,2,5, for instance, our total distance will be the sum of $M_{0,3} + M_{3,2} + M_{2,5}$.

Table 4.7: Example of a distance matrix.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
P0		46	17	24	51								
P1	46				31	33		54					
P2		38			34	31			51				
P3	24				33			17	49	31			
P4	51					4			18	39	60		
P5	48				4		4	27		35	57	51	
P6				33	1						59		
P7		54	26		32	27	31			14	42	66	
P8			51	49	18	20	17	43			57	32	
P9					39	35		14			28		
P10					60							58	6
P11									32	61	58		56
P12									59			56	

4.4.1 Constructing a model

What we need to decide in this problem is the sequence of points to choose to go from start to end. This is a subset of the given points (say P) along with an order in which to traverse them. It turns out that the most efficient approach is to picture a graph with the points as nodes and with the distances as weights on the arcs. Choosing a path on the graph will correspond to a path on the original map.

At first glance, this may not seem natural. It may not even be clear how to construct a decision variable to hold the subset of points and the order in which to visit them. Here is the trick: for each arc on the graph we have a decision variable that will take on exactly one of two values, either 0 if we do not take this arc or 1 if we do. Therefore,

$$x_{i,j} \in [0,1] \quad \forall i \in P, \forall j \in P.$$

To pursue our example path 0,3,2,5, we will have decision variables $x_{0,3}, x_{3,2}$ and $x_{2,5}$ at value 1 and all other arc variables at value 0. The reader should see the parallel with the other flow problems by thinking of a network where all arcs have a capacity of one: an integral solution will be a flow of value one on some sequence of adjacent arcs and zero on all others.

The objective function is correspondingly simple. Assuming that the distance matrix is D,

$$\min \sum_i \sum_j D_{i,j} x_{i,j}.$$

How do we ensure that we choose a sequence of adjacent arcs from the start point to the end point? By modeling this as a unit flow through the graph where the start point is a source of value one and the end point is a sink of value one. All we need is the usual flow conservation constraints we have used previously.

The executable model is seen in Code 4.4, where we assume a distance matrix `D` with optional starting and ending points. We could use our existing code for flow problems but, in this case, as a courtesy to the caller, we will write a special-purpose code to help the call and to return a meaningful answer. After all we, as modelers, are thinking of this problem as a flow on a graph, but the caller is thinking of a shortest path! Let us not burden him with our unnatural decision variables. Not to mention that there may be a million points, hence a trillion[3] decision variables and yet the solution, from the caller's perspective, may be only a minuscule fraction of those variables.

Code 4.4: Shortest path model.

```
def solve_model(D,Start=None, End=None):
  s,n = newSolver('Shortest␣path␣problem'),len(D)
  if Start is None:
    Start,End = 0,len(D)-1
  G = [[s.NumVar(0,1 if D[i][j] else 0,'') \
      for j in range(n)] for i in range(n)]
  for i in range(n):
    if i == Start:
      s.Add(1 == sum(G[Start][j] for j in range(n)))
      s.Add(0 == sum(G[j][Start] for j in range(n)))
    elif i == End:
      s.Add(1 == sum(G[j][End] for j in range(n)))
      s.Add(0 == sum(G[End][j] for j in range(n)))
    else:
      s.Add(sum(G[i][j] for j in range(n)) ==
           sum(G[j][i] for j in range(n)))
  s.Minimize(s.Sum(G[i][j]*D[i][j] \
                for i in range(n) for j in range(n)))
  rc = s.Solve()
  Path,Cost,Cumul,node=[Start],[0],[0],Start
  while rc == 0 and node != End and len(Path)<n:
    next = [i for i in range(n) if SolVal(G[node][i]) == 1][0]
    Path.append(next)
    Cost.append(D[node][next])
```

[3]If the reader reads American; a billion if English.

```
25     Cumul.append(Cumul[-1]+Cost[-1])
26     node = next
27   return rc,ObjVal(s),Path,Cost,Cumul
```

On line 3 we set the start and end nodes to be the first and last if the caller did not specify any. Line 6 defines the decision variable. We apply a little trick to the range: We know that if the distance matrix has a zero entry, it means that there is no path between two points. In that case, we give a range of $[0,0]$ which forces this variable to be zero. In the other cases, the range will be $[0,1]$. Notice that this range allows fractions but, again because of the structure of the constraints in a flow problem, no variable will ever have a fractional value. They will all be either 0 or 1.

At lines 9 and 12 we set the supply to be one at the start node and the demand to be one at the end node. At all other nodes (line 16) conservation of flow ensures that whatever goes in comes out. This will produce a solution consisting of a continuous path from start to end.

The objective function at line 18 has the same structure as all the flow problem examples: the product of the cost (here a distance) with the indicator variable of the arc used.

After we solve the problem, we process the solution to return to the caller something smaller, and potentially more meaningful than our decision variable: a sequence of jumps, from point to points, along with the distance of each jump. It is the job of the modeler to hide the tricks required to solve a problem and provide meaningful solutions to the caller. A solution corresponding to the example above is shown in Table 4.8.

Table 4.8: Optimal solution to shortest path problem.

Points	0	3	7	9	10	12
Distance	0	24	17	14	28	6
Cumulative	0	24	41	55	83	89

4.4.2 Alternate algorithms.

If the reader is aware of Dijkstra's algorithm, he may wonder why we create a linear programming model for shortest paths, especially since a fast implementation of Dijstra's algorithm might be faster. The answer is that, in the real life of a modeler, we rarely have to solve pure shortest paths (or a pure anything, really). For the vast majority of situations outside of textbooks, the kernel of the problem might be a shortest path, but there are bound to be multiple additional considerations. And adding these considerations in the form of additional constraints to a basic shortest path linear program is often a simple matter. In contrast, trying to modify an implementation of Dijkstra (assuming that we even have access to the source code) might prove considerably more difficult, if at all possible.

4.4.3 Variations

- It may be that, instead of minimizing the sum of distances, we want to minimize their product. We cannot multiply variables with a linear solver, but we can slightly transform the problem by taking the logarithms of the distances and minimizing the sum of logs.

- Alternatively we might be interested in the longest path between start end end. In theory, this is unlikely to be solved by linear programming in all cases[4] But the pathological cases that hinder the theory are few and may not apply to the problem at hand. Another way to view this is that there is a large class of networks where it is possible to find longest paths.

  The simplest transformation would be to change the minimization to a maximization. We can obtain a maximization by egating the distance matrix. But this will allow repeated nodes; worse, it may lead to an unbounded model (an infinite loop). The problem is that the 'flow' could go around a cycle an infinite number of times. A partial solution is to add constraints to ensure that no more than one unit of flow goes into any node. This is a redundant constraint in the case of a minimization, but not in the case of a maximization. In this manner we get rid of the infinite loop and the repeated nodes. There still remains a problem: subtours. We will explain and handle this problem in Section 5.4.

  In the case of a cycle-free directed graph, then longest paths are easy to find. We have seen before a situation where these paths are of interest: in Section 2.3.1 we discussed *critical paths* of project management. These are the sequence of tasks which, if delayed, will delay the whole project. Note that these paths are rarely unique, so that looking for *a* (or worse *the*) longest path is misguided (See, for a simple example, Tasks 1 and 2 Figure `fig:process-example`). Let us create a small function which will start from the optimal solution of our project management model and use our shortest paths model to extract critical paths.

  Code 4.5: Critical tasks extractor.

```
1  def critical_tasks(D,t):
2    s = set([t[i]+D[i][1] \
3          for i in range(len(t))]+[t[i] for i in range(len(t))])
4    n,ix,start,end,times = len(s),0,min(s),max(s),{}
5    for e in s:
6      times[e]=ix
7      ix += 1
8    M = [[0 for _ in range(n)] for _ in range(n)]
9    for i in range(len(t)):
10     M[times[t[i]]][times[t[i]+D[i][1]]] = -D[i][1]
11   rc, v, Path, Cost, Cumul = solve_model(M,times[start],times[end])
12   T = [i for i in range(len(t)) \
13       for time in Path if times[t[i]+D[i][1]] == time]
14   return rc, T
```

  The first few lines create a set with all the tasks' starting and ending times; those will become our network nodes once we rename them $0, \ldots, n-1$. At line 9 we create our distance matrix by the negative of the duration of each task. There is an entry per task, from its starting to ending time.

  We then call our shortest path model, which, in this case, will find a longest path from earliest time to project completion time. Finally we extract all the tasks that end

[4]Notice that if we can solve the longest path, we can solve the Hamiltonian path. Also, recall that linear programs can be solved in polynomial time. Ergo, if we can solve the longest path problem via LP, we prove P = NP. Then, we collect one million dollars from the Clay Mathematics Institute.

on one of the nodes of the longest path. All these tasks are critical since they will stretch the already longest path if delayed.

Running this code on the example at Table 2.8 produces Table 4.9.

Table 4.9: Critical tasks of project management example.

[0	1	2	6	7	9]

- We might be interested in the shortest paths tree from a start node to every other node in the network. In this case we could run our shortest path model $n-1$ times, but it is simple and interesting to create a separate model, especially since we can return the solution in a more compact form than $n-1$ lists of paths.

  The idea of Code 4.6 is to set the starting node with a supply of $n-1$ (at line 8) and every other node with a demand of one (at line 12). The decision variables at line 5 each have an empty range if there is no corresponding arc or else a range of $[0, n]$. This is contrast to the previous shortest path code where the range was up to one. We need this relaxed range because the flow will not be a unit flow until the very last arc on a given path. That is, the arc incident to a leaf.

  We return a list of arcs in the tree, along with their distances. This is best displayed graphically as in Figure 4.5.

Code 4.6: Shortest paths tree model.

```
1  def solve_tree_model(D,Start=None):
2    s,n = newSolver('Shortest paths tree problem'),len(D)
3    Start = 0 if Start is None else Start
4    G = [[s.NumVar(0,min(n,D[i][j]),'') \
5         for j in range(n)] for i in range(n)]
6    for i in range(n):
7      if i == Start:
8        s.Add(n-1 == sum(G[Start][j] for j in range(n)))
9        s.Add(0 == sum(G[j][Start] for j in range(n)))
10     else:
11       s.Add(sum(G[j][i] for j in range(n)) - \
12             sum(G[i][j] for j in range(n))==1)
13   s.Minimize(s.Sum(G[i][j]*D[i][j] \
14                for i in range(n) for j in range(n)))
15   rc = s.Solve()
16   Tree = [[i,j, D[i][j]] for i in range(n) for j in range(n) \
17           if SolVal(G[i][j])>0]
18   return rc,ObjVal(s),Tree
```

- We might also be interested in the shortest paths between each pair of nodes. Again, if the reader is well-versed in combinatorial algorithms, he might be aware of the Floyd-Warshall algorithm, but for the same reasons we create a shortest path model, we might create an all-pairs shortest paths model or use our current model repeatedly

Table 4.10: Optimal solution to the shortest paths tree problem.

From	To	Distance
0	1	46
0	2	17
0	3	24
0	4	51
2	5	31
2	8	51
3	7	17
5	6	4
5	11	51
7	9	14
9	10	28
10	12	6

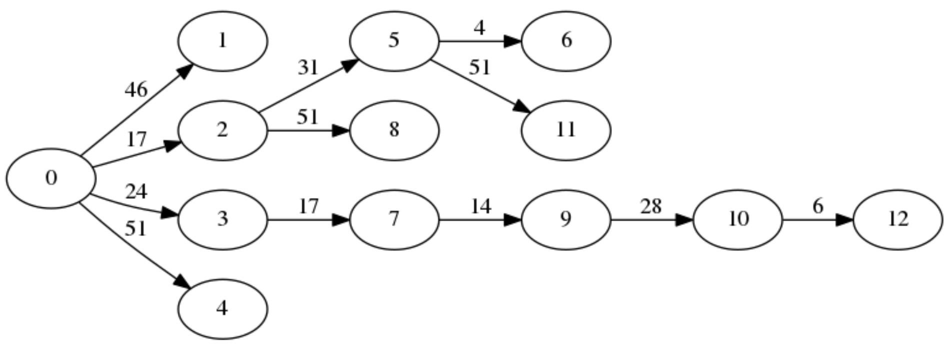


Figure 4.5: Shortest tree solution.

to find all pairs. We will see later a better approach but, since a few lines of code will suffice, let us write an all-pairs function (Code 4.7).

To avoid running n^2 instances we use the *Principle of Optimality*, which states that, if $P = (v_{i+1}, v_{i+2}, v_{i+3}, \ldots, v_{i+k})$ is a shortest path, then so is every sub-path of P. This is used in the loop starting at line 10 to extract all intermediate paths from a given shortest path.

Code 4.7: Simple all pairs shortest path function using our shortest path model.

```
1  def solve_all_pairs(D):
2    n = len(D)
3    Costs =[[None if i != j else 0 for i in range(n)] for j in range(n)]
4    Paths =[[None for i in range(n)] for j in range(n)]
5    for start in range(n):
6      for end in range(n):
7        if start != end and Costs[start][end] is None:
8          rc, Value, Path, Cost, Cumul = solve_model(D,start,end)
9          if rc==0:
10           for k in range(len(Path)-1):
```

```
         for l in range(k+1,len(Path)):
          if Costs[Path[k]][Path[l]] is None:
           Costs[Path[k]][Path[l]] = Cumul[l]-Cumul[k]
           Paths[Path[k]][Path[l]] = Path[k:l+1]
    return Paths, Costs
```

Running Code 4.7 on our example produces the matrix of distances at Table 4.11. The reader should notice that this matrix extends the initial distance matrix 4.7.

Table 4.11: Optimal solution to all-pairs shortest paths problem.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
P0		46	17	24	51	48	52	41	68	55	83	99	89
P1	46		63	70	31	33	37	54	49	68	90	81	96
P2	79	38		68	34	31	35	58	51	66	88	82	94
P3	24	70	41		33	37	41	17	49	31	59	81	65
P4	51	85	57	41		4	8	31	18	39	60	50	66
P5	48	81	53	37	4		4	27	22	35	57	51	63
P6	52	86	58	33	1	5		32	19	40	59	51	65
P7	75	54	26	64	31	27	31		49	14	42	66	48
P8	68	89	51	49	18	20	17	43		55	57	32	63
P9	83	68	40	72	39	35	39	14	57		28	80	34
P10	111	145	116	101	60	64	68	91	65	99		58	6
P11	100	121	83	81	50	52	49	75	32	61	58		56
P12	127	148	110	108	77	79	76	102	59	114	114	56	

Chapter 5

Linear pure discrete models

The problems in this chapters are, for the most part, classical examples of *Integer Programs* (IP). All are very simple to state, if not always simple to model or solve. They are included here to highlight two elements:

- First, many real-life problems have, embedded in them, one or more of these simple, *pure* problems. It is therefore profitable for a modeler to recognize these kernels and model them with ease.
- Second, many of the problems, to be efficiently modeled, require some trickery. Knowing these tricks, and recognizing different situations where they can be applied, is the hallmark of a good modeler.

What makes an integral model is the requirement that some or all of the variables be integral. Keep in mind that, contrary to the pseudo-integral models of the previous chapter, the structure of the problem does not guarantee this integrality and the modeler must choose a solver capable of handling integrality constraints.

There are a few reasons to require integral variables. The first and most obvious case is that we are counting objects, not measuring amounts (people, cars, planets as opposed to water, carbon dioxide or percentages). The second case occurs when the decision variables represent answers to yes/no questions (Should we build this plant? Should we get married?) or, more generally, Boolean conditions (with states either True or False, satisfying the principle of excluded middle). The third case, more technical, applies to auxiliary variables when they are used as “indicator variables”. That is when they indicate the presence or absence of a certain state (y is 1 if and only if the continuous variable x is non-zero). Of course, the boundary between these use case is blurry: a true decision variable could be an indicator variable and an auxiliary variable could be counting people. The three cases are nevertheless good to keep in mind while modeling.

Problems of this chapter have multiple interesting variations. We cannot possibly hope to cover them all, but the reader, after reading some of the variations, is encouraged to imagine others. No matter how creative one is in varying some of the requirements, most of these problems have been studied so extensively that few variations remain untouched and most have found some use[1].

[1]This might be a case, in the words of Wigner, of the unreasonable effectiveness of mathematics in the natural sciences; or more prosaically, because we mostly solve problems that we know how to solve.

5.1 Minimum set cover

The first problem in this chapter is one of the most studied and best understood of the Integer Programs. There are a number of applications of which here is one: General Engine Corp. is considering suppliers for its new line of electric cars. Every supplier can produce some parts of the cars with overlap between suppliers. For instance Dolphin inc. can supply wheel bearings, electrical cables, and low-power light emitting diodes while Schukert GA can supply electrical cables, batteries and battery casings. There are hundreds of suppliers and thousands of parts.

For General Engine, minimizing the numbers of suppliers offers contractual savings. So the goal is to find the smallest number of suppliers that, together, will provide all the required parts. The name *Set Cover* is explained by the goal: covering all the elements of the set, here the parts needed to build the electric cars.

The small example we will use to illustrate the model is given in Table 5.1.

Table 5.1: Example of Set Cover

Supplier	Part numbers	Supplier	Part numbers
S0	{ 3; 4; 5; 8; 24 }	S1	{ 11; 15; 21; 23 }
S2	{ 9; 15; 24 }	S3	{ 9; 13 }
S4	{ 5; 11; 12; 14; 16; 20 }	S5	{ 8; 11; 12; 15; 21 }
S6	{ 1; 4; 18; 20 }	S7	{ 0; 3; 6; 11; 13; 15; 21; 23 }
S8	{ 14; 16; 18; 19; 23 }	S9	{ 2; 7; 16; 22 }
S10	{ 10; 14; 21 }	S11	{ 6; 19 }
S12	{ 4; 10; 24 }	S13	{ 3; 4; 7; 9; 17 }
S14	{ 1; 3; 5; 6; 15; 18; 19; 20; 23 }		

5.1.1 Constructing a model

5.1.1.1 Decision variables

What we need to decide in this problem is which suppliers will get contracts. This is a yes/no decision. We need, for each supplier, a variable that will take on one of two values. The classical approach, in Integer Programming, is to use an integer variable with a range of $[0, 1]$. Being integer, it therefore only has two possible values zero and one and is known as a binary variable. [2]

There are other possible approaches: one using Boolean variables taking on values `True` and `False`, but this is really the same approach, renamed; and one using a dynamic array variable that will include only the chosen suppliers. This later approach may seem natural at first glance, but is not easily implemented using an Integer Solver. It is better suited to a Constraint Solver, which we will not cover here.

Let us, for now, assume a set S of suppliers and declare our first integer variables

$$s_i \in \{0, 1\} \quad \forall i \in S$$

The interpretation is that if, for example s_3, s_5 and s_7 are one while all the others are zero, then General Engine awards a contract to suppliers 3,5 and 7 only. It may occur to the

[2] There are cases where a binary choice -1, 1 would make the model simpler. Alas, no popular integer solver offers that option.

reader that in cases where the number of suppliers is much larger than the final set of chosen suppliers we are wasting resources. We will try to mitigate this waste but in a certain sense, it is unavoidable in Integer Programs.

5.1.1.2 Objective

The objective is to minimize the number of suppliers. Since we have a zero-one variable per supplier, we need to minimize the sum of all these. Therefore

$$\min \sum_{i \in S} s_i$$

5.1.1.3 Constraints

What are the constraints in this problem? From a high-level perspective, there is only one: General Engine must have access to all the required parts. Of course, there may be more than one supplier for some parts, but what we must not have is a part with no supplier (a car without a steering wheel might not sell so well).

How can we insure that we have all parts? Consider a given part, say part 23. Which suppliers provide it? There may be four, say 1, 7, 8 and 14. Which means that we much choose one of these suppliers to get part 23 or, algebraically, that the sum $s_1 + s_7 + s_8 + s_{14}$ must be at least one. (Not equal to one, as there are, in general no solutions without some redundancy.)

This leads us to a constraint per part j in the set P of all parts. We will assume that we have sets P_i of parts supplied by supplier i, just as in Table 5.1.

$$\sum_{i: j \in P_i} s_i \geq 1 \quad j \in P \tag{5.1}$$

The notation $\{i : j \in P_i\}$ is meant to indicate that we choose index i only if index j is in the set P_i. We will see how easily this is accomplished in the executable model.

5.1.1.4 Executable model

In Code 5.1, we see the whole model. Let us look at it carefully, highlighting the two major differences with all the previous models: the solver instantiation and the variable declaration.

Code 5.1: Set Cover model

```
def solve_model(D,C=None):
  t = 'Set Cover'
  s = pywraplp.Solver(t,pywraplp.Solver.CBC_MIXED_INTEGER_PROGRAMMING)
  nbSup = len(D)
  nbParts = max([e for d in D for e in d])+1
  S = [s.IntVar(0,1,'') for i in range(nbSup)]
  for j in range(nbParts):
    s.Add(1 <= sum(S[i] for i in range(nbSup) if j in D[i]))
  s.Minimize(s.Sum(S[i]*(1 if C is None else C[i]) \
                   for i in range(nbSup)))
  rc = s.Solve()
  Suppliers = [i for i in range(nbSup) if SolVal(S[i])>0]
```

```
13   Parts = [[i for i in range(nbSup) if j in D[i] and SolVal(S[i])>0]
14         for j in range(nbParts)]
15   return rc,ObjVal(s),Suppliers,Parts
```

It receives a two-dimensional array `D` containing the part numbers supplied by each supplier, exactly as in Table 5.1. The code also will accept a cost array `C` to be explained later. It is optional and its absence indicates a pure set cover problem: one where we are concerned with minimizing the number of subsets chosen.

The line 3 is different from all our previous models. It chooses a solver, in this case, CBC from IBM[3], that can handle integer variables as well as continuous variables. This very small change on our part, represents an order of magnitude change on the part of the solver. In fact, to solve an integer model, most solvers will internally solve a multitude of continuous models derived from ours. The algorithms are fascinating but beyond the scope of this book. [4]

For this first model, we create the solver instance using the low-level or-tools routine `Solver`. From here onward, we will use our own `newSolver` in this manner:

```
s = newSolver('Name of problem', True)
```

The second parameter, which defaults to `False`, instantiates an integer solver if `True`. Internally we usually use CBC, but there are a number of possible Integer solvers. (See Code 6.32 for details.)

Line 6 defines our binary variable (with the understanding that zero will mean 'ignore supplier' and one will mean 'pick supplier'). Up until now, all variables were defined with `NumVar` which implies a floating point variable modeling a real number. With `IntVar` we are instructing the solver that this variable can only take on integral values. Since we give it a range of zero to one, it forces the variable to have only one of two values. Any range is possible with all solvers.

The reader should experiment with this model by changing the `IntVar` to a `NumVar` and note that the variables will now take on values, zero, one-half and one [5]. What does it mean to have one-half of a supplier? Nothing, hence the integrality requirement.

The loop at line 8 implements the cover constraints. It mimics to the letter the constraint (5.1) forcing the sum of suppliers of a each part to be above one. Notice how easily we can extract subsets based on conditionals in Python.

The cost function is either the number of supplier, as in a traditional set cover, or the total cost of choosing these suppliers if each one incurs a different cost (maybe the cost is based on the number of parts, or on the bargaining strength of the supplier). This is why we added an optional cost array `C`, indexed by supplier.

Finally, after we solve, we construct meaningful return values. It would be painful to the caller to receive the raw `S` variables. Most of them might be zero. In a real problem with thousands, maybe tens of thousands of parts and suppliers, the zeros are not interesting. So we return an array containing only the suppliers who should be offered a contract, along with a cross-reference of parts to suppliers. This way the user knows where to go for each part.

For our example, the solution, absent a cost array, is displayed in Table 5.2. The first line list all retained suppliers, the next indicate who can supply (among those retained) each part. Note that each part is covered.

[3] http://www.coin-or.org/projects/Cbc.xml

[4] The interested reader should search "Branch and bound" to start reading about the solution techniques.

[5] No other fractions are possible in a pure Set Cover problem for fascinating reasons the reader is encouraged to research. Keyword search: 'half-integrality'

Table 5.2: Optimal solution to the Set Cover

Parts	Suppliers	Parts	Suppliers
All	{ 5; 7; 9; 10; 12; 13; 14 }	Part #0	{ 7 }
Part #1	{ 14 }	Part #2	{ 9 }
Part #3	{ 7; 13; 14 }	Part #4	{ 12; 13 }
Part #5	{ 14 }	Part #6	{ 7; 14 }
Part #7	{ 9; 13 }	Part #8	{ 5 }
Part #9	{ 13 }	Part #10	{ 10; 12 }
Part #11	{ 5; 7 }	Part #12	{ 5 }
Part #13	{ 7 }	Part #14	{ 10 }
Part #15	{ 5; 7; 14 }	Part #16	{ 9 }
Part #17	{ 13 }	Part #18	{ 14 }
Part #19	{ 14 }	Part #20	{ 14 }
Part #21	{ 5; 7; 10 }	Part #22	{ 9 }
Part #23	{ 7; 14 }	Part #24	{ 12 }

5.1.2 Variations

The first variation encountered includes a cost per supplier. So that, instead of simply minimizing the number of suppliers, we want to minimizing the total cost. This is trivially accomplished, from the modeler's point of view. Say we have cost C_i for supplier i, we replace the objective function by

$$\min \sum_{i \in S} C_i s_i$$

This is a trivial change, which we have implemented in the code as the caller can optionally provide a cost array `C`. Note that it will usually cause the solver to take more time to solve.

5.1.3 Instances

- A famous instance of the set cover problem is part of the infamous *Crew Scheduling* problem. Imagine that we are an airline and want to make sure that all the so-called 'legs', pairs of cities, are covered during a certain time window. We have rosters of crew members, travelling together, from city A to city B, with stopovers in cities C, D, ..., E. Our task is to cover all legs using the minimum number of rosters.

- Another geeky example involves computer virus detection. Imagine that we have a database of thousands of computer viruses and are trying to build a detector. One way to do this is to try to identify short strings of bytes that are present in these viruses but not in non-virus code. What we want is to minimize the numbers of strings and yet identify all viruses. Then our detector will run by looking for this small set of strings on the data (all programs on the hard drive).

- There are applications in telecommunications. Imagine that we can build cell towers in a city in a number of locations. Considering the cost of each, we want to minimize expenditures and yet cover all the buildings and houses in the city.

- Where to locate fire stations in a city so that, considering the average response time, we minimize the number of stations and yet cover the whole city.

5.2 Set packing

A closely related problem to the Set Cover is the Set Packing . In both cases we are given a universal set and a set of subsets and we need to choose some of them. In the former case, we aim at covering the univeral set with a minimal set of subsets, possibly covering some elements more than once. In the latter case, the objective is to choose as many of the subsets as possible but without ever choosing an elements more than once. Therefore, some elements may not be covered.

An interesting application of this comes from airline crew scheduling. Consider that each plane must have a pilot, a co-pilot, a navigator, etc. These are called rosters. Some pilots may be flying some types of planes, but not others. Pilots also may have preferences for their co-pilots (and vice-versa). Conceptually we can think of a specific combination plane, pilot, co-pilot etc . . . as a subset of our universal set of planes, pilot, co-pilot.

What we want is to maximize the number of subsets we choose, but we must not pick two subsets that share elements (as a pilot cannot be at two places at once.) The Table 5.3 is a small instance of this problem.

Table 5.3: Example of Set Packing

Roster #	Crew IDs	Roster #	Crew IDs
0	{ 17; 18; 28 }	1	{ 18; 21; 38 }
2	{ 7; 13; 31 }	3	{ 14; 39; 39 }
4	{ 5; 6; 36 }	5	{ 16; 23; 34 }
6	{ 5; 9; 26 }	7	{ 20; 29; 39 }
8	{ 14; 14; 24 }	9	{ 10; 31; 36 }
10	{ 20; 31; 37 }	11	{ 6; 28; 35 }
12	{ 2; 8; 25 }	13	{ 4; 13; 30 }
14	{ 8; 21; 38 }		

5.2.1 Constructing a model

5.2.1.1 Decision variables

What we need to decide in this problem is very similar to the decision for set cover: which roster will we pick? A yes-no decision, which suggest an indicator variable. Let us assume a set S of crew rosters and declare our indicator variables

$$s_i \in \{0,1\} \quad \forall i \in S$$

5.2.1.2 Objective

The objective is to maximize the number of rosters chosen, therefore

$$\max \sum_{i \in S} s_i$$

5.2.1.3 Constraints

The constraint, and there is only one, is never to pick two rosters including the same crew member. Since our decision variables are zero-one we can simply force, for each crew, that the sum of roster variables is at most one.

If each roster is in S_i and the universal set of crew is U

$$\sum_{i:j\in S_i} s_i \leq 1 \quad \forall j \in U$$

5.2.1.4 Executable model

The executable model is seen at Code 5.2. Very similar to Code 5.1, it receives a two-dimensional array `D` with a list of crew rosters, exactly as Table 5.3. The function will also accept a cost function to attach to each roster if required.

Code 5.2: Set Packing model

```
def solve_model(D,C=None):
  s = newSolver('Set␣Packing', True)
  nbRosters,nbCrew = len(D),max([e for d in D for e in d])+1
  S = [s.IntVar(0,1,'') for i in range(nbRosters)]
  for j in range(nbCrew):
    s.Add(1 >= sum(S[i] for i in range(nbRosters) if j in D[i]))
  s.Maximize(s.Sum(S[i]*(1 if C==None else C[i]) \
                 for i in range(nbRosters)))
  rc = s.Solve()
  Rosters = [i for i in range(nbRosters) if S[i].SolutionValue()>0]
  return rc,s.Objective().Value(),Rosters
```

A solution for our instance, with no cost array, appears in Table 5.4.

Table 5.4: Optimal solution to the Set Packing

Rosters chosen	{ 1; 5; 6; 7; 8; 9; 11; 12; 13 }

5.2.2 Variations

- The main variation is to have a cost on the rosters selected. We then minimize the total cost. The code given already implements this possibility.
- Another possibility is that we have a combination of Set cover and Set packing: we want to cover entirely the universal set and use each element exactly once. In this case we speak of Set Partitioning.

5.3 Bin packing

The *Bin Packing* problem appears in a number of incarnations. Abstractly, it is the problem of partitioning a set, where each element has a weight so that we minimize the number of groups and yet maintain each group under a prescribed weight limit.

The local warehouse of the international parcel delivery company, VQT inc, has a number of trucks, each with a maximum weight capacity. On a particular morning, they have packages of various weights to transport. A simple instance is described in Table 5.5. The goal is to minimize the number of trucks used to deliver all packages.

Note that having only a weight limit is not entirely realistic. Packages also have a volume and it is likely that we need to be able to pack according to volume. But that problem is considerably more difficult and we will leave it aside. There should also be consideration of distances, which we will tackle in a later section (Section 5.4). To repeat a point worth repeating: few, if any, real life optimization problems are pure, simple, textbook problems. They are always a combination of multiple problems. A good modeler recognizes this and has the toolset to model all.

Table 5.5: Example of Bin Packing

	Truck weight limit	1264
	nb of packages	Unit weight
0	8	219
1	10	293
2	8	267
Total	26	6818

5.3.1 Constructing a model

5.3.1.1 Decision variables

What we need to decide is "'Which package goes into which truck?" We know all the packages. Though there are many instances of identical packages (identical for our purposes, that is with the same weight), we can give them ordinal numbers. But we do not really know the number of trucks. That is one of the questions we are trying to answer. Nevertheless, we can certainly give an upper bound on the number of trucks by some heuristic. At worst, we can say with certainty that we will need at most one truck per package.

A slightly better heuristic is to start adding packages to the first truck until we reach capacity, then move on to the second one, etc .... This greedy approach will never be optimal but is enough to get a reasonable upper bound on the required number of trucks. This is presented at Code 5.4.

So let us assume P packages and at most T trucks. Our decision variable is

$$x_{i,j} \in \{0,1\} \quad \forall i \in P, j \in T$$

where $x_{i,j} = 1$ will mean that package i goes into truck j.

This is a good start but we also need to know which of the trucks we will need. So another decision variable seems indicated, namely

$$y_j \in \{0,1\} \quad \forall j \in T$$

where $y_j = 1$ will indicate that truck j is to be used. This seems to answer all questions we need answered.

5.3.1.2 Constraints

First, we need to establish a relation between our $x_{i,j}$ and y_j variables since we must have a given y_j equal to one (truck j used) if $x_{i,j}$ is one for any i. Another way to view this is that we cannot put packages in a truck we do not use. We have seen this type of constraint before, even on one of the first models, diet, when we discussed "If food 2 is used, then we must have at least as much food 3 in the diet" in section 2.1.2.

The general idea is to ensure that one variable is bounded by another or by a multiple of another. In this case, that trick suggests using

$$x_{i,j} \leq y_j \quad \forall i \in P, \forall j \in T \tag{5.2}$$

This does satisfy our relationship, though it may look rather wasteful to the reader. Indeed we will prune this attempt. Coincidentally, we also need to bound the sum of the package weights in every certain truck. Assuming that package i has weight w_i and truck j has capacity W_j, we need

$$\sum_{i \in P} w_i x_{i,j} \leq W_j \quad \forall j \in T \tag{5.3}$$

Is there a way to 'merge' (5.2) and (5.3)? Indeed there is:

$$\sum_{i \in P} w_i x_{i,j} \leq W_j y_j \quad \forall j \in T \tag{5.4}$$

We can see that Equation (5.4) subsumes both of (5.2) and (5.3). We have reduced the number of constraints from $|P||T| + |T|$ to $|T|$, a non-trivial improvement.

At this point, our model guarantees that

- a truck is used if any package is loaded in it,
- the sum of the package weights in a truck respects its capacity.

What is left? Ensuring that we actually put each package in some truck.

$$\sum_{j \in T} x_{i,j} = 1 \quad \forall i \in P \tag{5.5}$$

5.3.1.3 Objective

The objective is to minimize the number of trucks used, therefore

$$\min \sum_{j \in T} y_j$$

5.3.1.4 Executable model

We will assume that the function receives an array `D` containing a list of packages with their weights and the count of packages of each weight (we will call these weight classes), exactly as in Table 5.5. It also receives a weight capacity for each truck in `W`. The third parameter, optional, will be explained after we solve our small example. Just note that its default value is `False` and in that case, a large set of, as yet unexplained, constraints are skipped (lines 17 to 25).

Code 5.3: Bin Packing model

```
1  def solve_model(D,W,symmetry_break=False):
2    s = newSolver('Bin␣Packing',True),
3    nbC,nbP = len(D),sum([P[0] for P in D])
4    w = [e for sub in [[d[1]]*d[0] for d in D] for e in sub]
5    nbT,nbTmin = bound_trucks(w,W)
6    x = [[[s.IntVar(0,1,'') for _ in range(nbT)] for _ in range(d[0])] \
7         for d in D]
8    y = [s.IntVar(0,1,'') for _ in range(nbT)]
9    for k in range(nbT):
10     sxk = sum(D[i][1]*x[i][j][k] \
11          for i in range(nbC) for j in range(D[i][0]))
12     s.Add(sxk <= W*y[k])
13   for i in range(nbC):
14     for j in range(D[i][0]):
15       s.Add(sum([x[i][j][k] for k in range(nbT)]) == 1)
16   if symmetry_break:
17     for k in range(nbT-1):
18       s.Add(y[k] >= y[k+1])
19     for i in range(nbC):
20       for j in range(D[i][0]):
21         for k in range(nbT):
22           for jj in range(max(0,j-1),j):
23             s.Add(sum(x[i][jj][kk] for kk in range(k+1)) >= x[i][j][k])
24           for jj in range(j+1,min(j+2,D[i][0])):
25             s.Add(sum(x[i][jj][kk] for kk in range(k,nbT))>=x[i][j][k])
26   s.Add(sum(y[k] for k in range(nbT)) >= nbTmin)
27   s.Minimize(sum(y[k] for k in range(nbT)))
28   rc = s.Solve()
29   P2T=[[D[i][1], [k for j in range(D[i][0]) for k in range(nbT)
30                 if SolVal(x[i][j][k])>0]] for i in range(nbC) ]
31   T2P=[[k, [(i,j,D[i][1]) for i in range(nbC) for j in range(D[i][0])
32           if SolVal(x[i][j][k])>0]] for k in range(nbT)]
33   return rc,ObjVal(s),P2T,T2P
```

At line 4 we construct an array of weights, one per package. This implicitly also assigns an ordinal to each package. The function `bound_trucks`, described at Code 5.4 uses the weights of the packages and the capacity of each truck to quickly estimate an upper bound on the number of trucks. This function does not need to be brilliant, but a better bound accelerates the solver.

The two lines starting at 7 define our decisions variables; one to assign packages to trucks, and one to select trucks. The package variable is a three dimensional array. The first dimension indicates the weight class, the second is the ordinal within the class and the third is the truck. So that if, for example, `x[2][3][5]` has value one it will mean that package three of the weight class two is loaded onto truck five.

The constraint at the loop 9 is a transcription of (5.4), our ‘merged’ constraint to both force the truck selection variable and to limit the total package weight carried by a truck, modified to use the three-dimensional decision variables.

The final constraint at line 15 is a transcription of (5.5) to ensure all packages find a truck.

Ignore the lines starting at 16 and guarded by the `symmetry_break` parameter for now.

After solving, we produce two arrays of the solution: one indicating, for each package, the loading truck; a second array indicating, for each truck, the list of packages. The solution for our instance appears in Tables 5.6 and 5.7. The first table lists the trucks and their content indicated by a triple (weight class, package ordinal, weight). The second table list each weight class, in the same order as Table 5.5, with the truck in which each package of the class is loaded.

Table 5.6: Optimal truck loads without symmetry breaking

Trucks 6.0	Packages 26 (6818)
(id weight)	(id weight)*
1 (1191)	[(0, 0, 219), (0, 4, 219), (0, 5, 219), (2, 1, 267), (2, 7, 267)]
2 (1072)	[(0, 1, 219), (1, 3, 293), (1, 7, 293), (2, 6, 267)]
3 (1120)	[(1, 0, 293), (1, 8, 293), (2, 2, 267), (2, 5, 267)]
5 (1243)	[(0, 2, 219), (0, 3, 219), (0, 7, 219), (1, 1, 293), (1, 5, 293)]
6 (1098)	[(0, 6, 219), (1, 2, 293), (1, 6, 293), (1, 9, 293)]
7 (1094)	[(1, 4, 293), (2, 0, 267), (2, 3, 267), (2, 4, 267)]

Table 5.7: Optimal package assignments without symmetry breaking

Weight	Truck id
219	[1, 2, 5, 5, 1, 1, 6, 5]
293	[3, 5, 6, 2, 7, 5, 6, 2, 3, 6]
267	[7, 1, 3, 7, 7, 3, 2, 1]

Even for small instances, this code can take hours to produce a solution. Bin packing is not an easy problem and here is part of the reason why: Notice that some truck numbers are 'skipped'; the solver seemed to have chosen trucks at random. Also, look at how randomly the packages of a given weight class are distributed among the trucks. Indeed, executing the same model on the same instance on a different computer or a different solver might very well produce a different answer (of course, with the same total number of trucks). Because there is a large number of solutions with exactly the same value. Think, for instance, of swapping the entire content of two trucks with the same capacity or swapping two packages of the same weight class within a truck, or between two trucks.

In classical optimization terms, this situation is a form[6] of *degeneracy*. Researchers in Constraint Programming talk[7] of *symmetry*. It almost always affects the solver negatively. The runtime is difficult to predict as it is solver dependent but it is rarely good. There is another reason to want to modify the model to avoid these identical solutions: we could produce 'nicer' solutions.

Adding constraints which favour one optimal solution over another identical one (identical from our point of view), is known as *symmetry breaking*, which explains the parameter `symmetry_break` of the code guarding the additional constraints, which we now proceed to explain. Let us tackle the easiest symmetry first.

[6] For the theoretically-minded, this is a case of dual-degeneracy.

[7] The symmetry stems from visualizing the search tree and noticing that there are multiple branches with exactly the same structure and value.

How can we ensure that the trucks are chosen in order and none are skipped (assuming, of course, that they all have the same capacity)? One way is to bound the truck selection variables pairwise:

$$y_i \geq y_{i+1}$$

To see how this works, consider what happens to the y vector for, say y_5 to be one. It must be that y_4 is one and, transitively, so must be y_3, y_2, y_1 and y_0. On the other hand, it has no effect on y_6 or higher. In terms of code this is done at the loop at line 17.

The second form of symmetry we alluded to is interchangeable packages. The way we stated the problem, there is no difference between two packages in the same weight class. Yet, for the solver swapping two packages within a truck or between two trucks is another potential solution, and any time spent looking in that direction is time wasted.

We consider how to break this symmetry by looking at a small example, say three packages and three trucks. The idea is that since the packages are naturally ordered, let us force that they should be loaded into trucks in their order. For instance, if the second package is loaded in truck one, then the third can only be loaded in truck one or higher. In terms of the decision variables, we want the following implications:

$$\begin{aligned}
&x_{0,2} = 1 \Rightarrow x_{1,2} = 1 \wedge x_{2,2} = 1\\
&x_{0,1} = 1 \Rightarrow x_{1,1} + x_{1,2} = 1 \wedge x_{2,1} + x_{2,2} = 1\\
&x_{0,0} = 1 \Rightarrow x_{1,0} + x_{1,1} + x_{1,2} = 1 \wedge x_{2,0} + x_{2,1} + x_{2,2} = 1\\
&x_{1,2} = 1 \Rightarrow x_{2,2} = 1 \qquad \wedge\ x_{0,0} + x_{0,1} + x_{0,2} = 1\\
&x_{1,1} = 1 \Rightarrow x_{2,1} + x_{2,2} = 1 \qquad \wedge\ x_{0,0} + x_{0,1} = 1\\
&x_{1,0} = 1 \Rightarrow x_{2,0} + x_{2,1} + x_{2,2} = 1 \qquad \wedge\ x_{0,0} = 1\\
&x_{2,2} = 1 \Rightarrow \qquad x_{0,0} + x_{0,1} + x_{0,2} = 1 \wedge x_{1,0} + x_{1,1} + x_{1,2} = 1\\
&x_{2,1} = 1 \Rightarrow \qquad x_{0,0} + x_{0,1} = 1 \wedge x_{1,0} + x_{1,1} = 1\\
&x_{2,0} = 1 \Rightarrow \qquad x_{0,0} = 1 \wedge x_{1,0} = 1
\end{aligned}$$

We will see later (Section 6.5.3 on reification) a general way to implement these implications. For now, let us try to implement them as simply as possible.

The first implication says "If package 0 is loaded into truck 2, then both packages 1 and 2 must be loaded into truck 1". But this is a boundary case as truck 2 is our last truck. The next implication is more interesting: "If package 0 is loaded onto truck 1, then packages 1 and 2 must be loaded onto truck 1 or 2".

In all generality: "If package i is loaded onto truck k, then all packages $ii > i$ must be loaded onto trucks $kk \geq k$." Abstractly, the constraint structure is that if some variable takes on value 1, we must have an equation *with right hand side 1* holding. The unit right hand side is our ticket, as we can use the conditional variable as the right hand side. But beware: consider part of the second implication $x_{0,1} = 1 \Rightarrow x_{1,1} + x_{1,2} = 1$; the following naive approach will fail.

$$x_{1,1} + x_{1,2} = x_{0,1}$$

It will fail because if $x_{0,1}$ is zero (for example if package 0 is loaded onto truck 0 instead of 1), then we are preventing package 1 to be loaded onto trucks 1 or 2, which would be acceptable. So the right constraint is

$$x_{1,1} + x_{1,2} \geq x_{0,1}$$

The right hand side at zero trivializes the constraints as all variables on the left hand side are non-negative; at one, it forces a correct assignment to a higher numbered truck. These are implemented at line 25.

The right column of the implications can be seen to say "If package i is loaded onto truck k, then all packages $ii < i$ must be loaded onto trucks $kk \leq k$." These are mostly redundant yet they can help some solvers under some conditions. The reader is encouraged to enable or disable some symmetry breaking constraints and experiment.

All these additional constraints reduce the search space. With some solvers, on some problems, the approach will drastically reduce the execution time. With these symmetry breaking constraints enabled by calling `solve_model` with the last parameter as `True`, the output, on the same instance is shown at Tables 5.8 and 5.9. We see that all trucks used are consecutive from zero and that the packages are loaded in order. A much nicer solution.

Table 5.8: Optimal truck loads with symmetry breaking

Trucks 6.0	Packages 26 (6818)
(id weight)	(id weight)*
0 (1191)	[(0, 0, 219), (0, 1, 219), (0, 2, 219), (2, 0, 267), (2, 1, 267)]
1 (1095)	[(0, 3, 219), (0, 4, 219), (0, 5, 219), (0, 6, 219), (0, 7, 219)]
2 (1068)	[(2, 2, 267), (2, 3, 267), (2, 4, 267), (2, 5, 267)]
3 (1172)	[(1, 0, 293), (1, 1, 293), (1, 2, 293), (1, 3, 293)]
4 (1120)	[(1, 4, 293), (1, 5, 293), (2, 6, 267), (2, 7, 267)]
5 (1172)	[(1, 6, 293), (1, 7, 293), (1, 8, 293), (1, 9, 293)]

Table 5.9: Optimal package assignments with symmetry breaking

Weight	Truck id
219	[0, 0, 0, 1, 1, 1, 1, 1]
293	[3, 3, 3, 3, 4, 4, 5, 5, 5, 5]
267	[0, 0, 2, 2, 2, 2, 4, 4]

Code 5.4: Simple-minded heuristic to bound the number of trucks

```
def bound_trucks(w,W):
 nb,tot = 1,0
 for i in range(len(w)):
  if tot+w[i] < W:
   tot += w[i]
  else:
   tot = w[i]
   nb = nb+1
 return nb,ceil(sum(w)/W)
```

5.3.2 Variations

- It may be that each truck has a different weight capacity. The capacity constraint is simple to adapt, but care must be taken with the symmetry breaking constraint. Skipping some trucks may be unavoidable. So the symmetry breaking must be done only within subsets of trucks with the same capacity.

- Instead of loading a fixed number of packages in an undetermined number of trucks one may have a fixed number of trucks and an undetermined number of packages to load. In that situation, packages, in addition to a weight, usually also have a value and one is trying to maximize the total value. This situation is usually much simpler to solve. Assuming that package i has value v_i, the objective function is

$$\max \sum_{i \in P} \sum_{j \in T} v_i x_{i,j}$$

  subject to constraint (5.4).

- There is a simpler version of bin packing known as *Knapsack* where packages have value and weight, but there is only one truck with a weight capacity. This problem is so simple that there exists very fast algorithms for it. But, a general-purpose integer solver will, of course, solve it without any difficulty. Even if it is simple it does have some value, not as a problem that occurs naturally, but as a subproblem of a more complex situation. We will see examples of this later.

- A closely related problem is that of *Capital Budgeting*. Consider a multi-period planning horizon T and a set of possible projects, P; each project j requiring an investment of a_{tj} in period t and representing a value c_j. Given a limited budget b_t in period t, which projects should be earmarked for investments? The model is a simplification of bin packing:

$$\begin{aligned} \max & \sum_{j \in P} c_j y_j \\ & \sum_{j \in P} a_{tj} y_i \leq b_t \quad \forall t \in T \\ & y_i \in \{0, 1\} \end{aligned}$$

  Where the y_j represents the decision "Go ahead (or not) with project j."

5.4 TSP

We tackle now the venerable *Travelling Salesman Problem* hereafter TSP , mostly[8] because it never, ever, was an important problem for salesmen; but it is a very important problem in vehicle routing, electronic circuit design and job sequencing, among other applications. Moreover, it will allow us to describe, with a minimum of spurious complexity, a very important modeling technique.

So here is an example situation: At HAL inc, during the process of a new circuit design, power must be routed to each elementary component. These components are set in a two-dimensional lattice, potentially all pairwise connected. The best way to feed power to these components is to establish a path of minimal total length, conceptually starting at the power supply (V_{cc}), going around to each component, then coming back to the power supply (V_{ee} or ground)[9].

Therefore, the problem, viewed abstractly, can be stated as "Given a matrix of pairwise distances in a graph, find a tour of all vertices minimizing total distance." Table 5.10 is the example we will solve to illustrate. In addition to the distances, it includes the Cartesian coordinates of the points. We will not use these coordinates in the model but they are useful for visualizing the problem.

Table 5.10: Example of distance matrix for TSP

P (x y)	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9
P0 25 59		56	59	209	93	100	176	171	411	156
P1 98 66	449		175	270	355	322	276	719	345	171
P2 49 31	86	341		607	249	331	409	140	428	241
P3 14 100	210	723	677		330	207	641	469	225	408
P4 76 26	448	312	257	928		191	113	239	426	31
P5 60 71	142	329	278	477	438		451	541	197	545
P6 91 29	299	153	187	802	149	350		48	94	15
P7 32 19	60		82	252	423	90	55		726	216
P8 59 93	259	160	249	128	70	214	621	679		512
P9 74 14	40	409	21	638	19	574	207	179	739	

5.4.1 Constructing a model

5.4.1.1 Decision variables

What we need to decide in this problem is simply the path to take, which means the sequence of points to follow. This is identical to our decision in the shortest path problem, therefore, assuming that P is the set of points, we define

$$x_{i,j} \in \{0,1\} \quad \forall i \in P, \forall j \in P$$

where $x_{i,j}$ with a value one will indicate that we need to connect points i and j. Beware: Even though this problem has the same decision variable and underlying graph structure as the shortest path problem, it is decidedly not a flow problem; it is considerably more complex.

[8] Not to avoid the political correctness gendarmes.

[9] Whether the trace comes back to the origin is irrelevant from a complexity standpoint. We can assume a distance zero between V_{cc} and V_{ee} if need be.

5.4.1.2 Objective

The objective function is correspondingly simple. Assuming that the distance matrix is D,

$$\min \sum_i \sum_j D_{i,j} x_{i,j}$$

Again, exactly as the shortest path objective function.

5.4.1.3 Constraints

Here is where the model differs from the shortest path model. We must ensure a tour, a single closed path covering every vertex in the graph exactly once. The 'exactly once' part is easy: for each vertex, we must choose an arc going in and one arc going out.

$$\sum_{j \in P \setminus \{i\}} x_{i,j} = 1 \qquad \forall i \in P \tag{5.6}$$

$$\sum_{j \in P \setminus \{i\}} x_{j,i} = 1 \qquad \forall i \in P \tag{5.7}$$

$$\tag{5.8}$$

Now comes the difficulty, and the interesting part. Are the above two constraints enough? Perhaps surprisingly, no. Every vertex is on a path, but there may be more than one such path. One path could be $0, 1, 3, 4, 0$, while the other covers all the others vertices. These problematic paths are known as subtours and must be eliminated. To eliminate the subtour $0, 1, 3, 4, 0$, we could add the following constraint

$$x_{0,1} + x_{1,0} + x_{1,3} + x_{3,1} + x_{3,4} + x_{4,3} + x_{4,0} + x_{0,4} \leq 3$$

This way, the solver will never include more than three arcs between the four problematic vertices, preventing a subtour among them. In general, if we have a subtour on k vertices, we need to add a constraint bounding the sum of all pairwise arcs between these vertices to be no larger than $k - 1$.

The difficulty is that there are many possible subtours, in fact, an exponential number: every subset of vertices of size larger than one is a potential subtour.

Can we add all the possible subtours? Programmatically, this is not difficult, but the resulting model would be unwieldy and many solvers would slow down unacceptably. The trick is to improve the model iteratively, as we did when optimizing a non-linear function 3.1.2.1. But here, we will use the result of a solver run to detect the subtours to eliminate.

Very simply: We execute a model with no subtour elimination constraints; if the solver returns a tour, we are done. If it returns a set of subtours, we add subtour elimination constraints for each of these. Eventually all relevant subtours are eliminated and the solver returns a tour. It takes longer to explain than to write the code to implement this approach (a half-dozen lines!)

5.4.1.4 Executable model

Let us translate this into executable code which we will split into two: a model that, given some set of subtours, will optimize after adding subtour elimination constraints for that particular set; and a main routine iteratively calling the first, adding subtours as they appear.

Code 5.5: TSP model with subtour elimination constraints

```
1 def solve_model_eliminate(D,Subtours=[]):
2   s,n = newSolver('TSP', True),len(D)
3   x = [[s.IntVar(0,1,'') for _ in range(n)] for _ in range(n)]
4   for i in range(n):
5     s.Add(1 == sum(x[i][j] for j in range(n)))
6     s.Add(1 == sum(x[j][i] for j in range(n)))
7     s.Add(0 == x[i][i])
8   for sub in Subtours:
9     K = [x[sub[i]][sub[j]]+x[sub[j]][sub[i]]\
10         for i in range(len(sub)-1) for j in range(i+1,len(sub))]
11    s.Add(len(sub)-1 >= sum(K))
12  s.Minimize(s.Sum(x[i][j]*D[i][j] \
13                   for i in range(n) for j in range(n)))
14  rc = s.Solve()
15  tours = extract_tours(SolVal(x),n)
16  return rc,ObjVal(s),tours
```

Line 3 defines our decision variable, a binary indicator of the arcs to take.

The loop starting at 4 enforces that there must be an arc in and an arc out of every node, exactly as in (5.6)-(5.7). We also enforce that all `x[i][i]` are zero to avoid loops.

For each subtour provided by the caller, we extract all arcs of the corresponding clique at line 8 and constrain the sum of these to be one less than the number of vertices in the clique.

We process the solution returned by the solver to extract, at line 15, the subtours and return them to the caller. This extraction code is shown at Code 5.6.

Code 5.6: Subtour extraction

```
def extract_tours(R,n):
  node,tours,allnodes = 0,[[0]],[0]+[1]*(n-1)
  while sum(allnodes) > 0:
    next = [i for i in range(n) if R[node][i]==1][0]
    if next not in tours[-1]:
      tours[-1].append(next)
      node = next
    else:
      node = allnodes.index(1)
      tours.append([node])
    allnodes[node] = 0
  return tours
```

The main loop is very simple: we iterate until the number of tours returned by the solver is one, taking care to accumulate subtours as they are discovered.

Code 5.7: TSP model mainline

```
def solve_model(D):
  subtours,tours = [],[]
  while len(tours) != 1:
    rc,Value,tours=solve_model_eliminate(D,subtours)
```

```
    if rc == 0:
      subtours.extend(tours)
  return rc,Value,tours[0]
```

The solution to our small example, at Table 5.11, one iteration per row, illustrates the subtours that were eliminated. In parentheses we see the optimal value, the total length. As we eliminate subtours, it increases, of course. A graphical representation is shown at Figure 5.1 where the subtours eliminated at each iterations are shown in distinct colors.

Table 5.11: Successive iterations of the TSP solver showing value and subtours

Iter (value)	Tour(s)
0-(1002)	[0, 1, 2]; [3, 5, 8]; [4, 9]; [6, 7]
1-(1057)	[0, 5]; [1, 2, 7]; [3, 8]; [4, 6, 9]
2-(1073)	[0, 2, 7, 1, 9, 4, 6, 8, 3, 5]

The key idea to remember about the TSP model is not that we can solve gigantic instances with it (specialized algorithms will do much better[10]). It is that, even if the number of constraints to completely specify a correct model is large, it may be possible to include a very small fraction of the required constraints and yet solve the problem to optimality. To do this effectively, one must have a deep understanding of the problem and good practical modeling skills. (And of course, a good modeling language, as Python and or-tools, helps: we did it in half a dozen lines!)

The reader should be aware that our subtour elimination constraints are not the only ones possible. There are a number of ways to eliminate subtours, but none are as simple to implement as those we described. In practice, problems that have a TSP sub-problem embedded in them have a number of other requirements, making the model relatively complex. Having a simple yet effective way to deal with the subtours is a skill all modelers need to know.

5.4.2 Variations

- A simple variation occurring often is that, instead of a tour (a closed path), one wants a simple path covering all vertices. For reference let us call this problem TSP-P. Since we know how to solve the tour problem, the easiest way to solve the path problem is to transform the latter into the former: we add another node to the network (let us call it the dummy node) and arcs of distance zero from the dummy node to every other node on the network. Then we solve the TSP on that new network. At optimality we will have a tour, therefore it will go into and out of the dummy node; deleting those two arcs yields the required path. Code to implement this variation is shown at Code 5.8 and the result of running the path model on our example yields Table 5.12.

Code 5.8: Code to solve the TSP-P problem

```
def solve_model_p(D):
  n,n1 = len(D),len(D)+1
  E = [[0 if n in (i,j) else D[i][j] for j in range(n1)] \
      for i in range(n1)]
```

[10] See http://www.math.uwaterloo.ca/tsp/concorde.html for instance

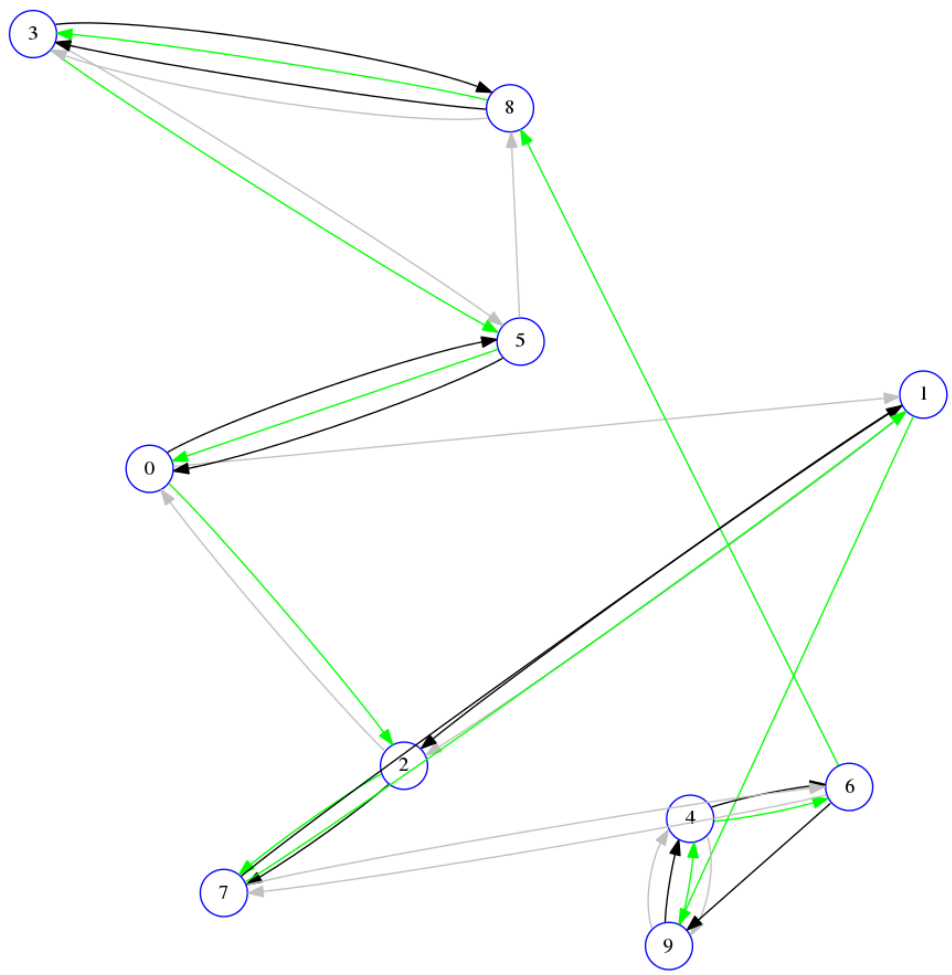


Figure 5.1: Successive (partial) solutions of TSP example.

```
rc,Value,tour = solve_model(E)
i = tour.index(n)
path = [tour[j] for j in range(i+1,n1)]+[tour[j] for j in range(i)]
return rc,Value,path
```

Table 5.12: Result of the path model on our example.

Nodes	3	8	4	6	7	1	9	2	0	5
Distance	0	225	70	113	48	0	171	21	86	100
Cumulative	0	225	295	408	456	456	627	648	734	834

- A more complex variation is to allow repeated visits to nodes. For reference, let us call this problem the TSP*. The justification for this problem is simple: It is conceivable that one could find a shorter overall walk if one is allowed to visit any node more than once.

Again the trick is to rely on our TSP model by transforming TSP* into TSP on a different network. The new network has exactly the same nodes, but the distance between the nodes is that of a shortest path between nodes of the original network.

We must take care of keeping track of these shortest paths to reconstruct the TSP* solution. Since we already implemented an all-pairs shortest paths model (Code 4.7), we will use it here. Code to implement TSP* is shown at Code 5.9 and its solution on our example is shown at Table 5.13 and on Figure 5.2. Note that the total length is less than the TSP length even though it repeats some nodes.

Code 5.9: Code to solve the TSP* problem

```
def solve_model_star(D):
 import shortest_path
 n = len(D)
 Paths, Costs = shortest_path.solve_all_pairs(D)
 rc,Value,tour = solve_model(Costs)
 Tour=[]
 for i in range(len(tour)):
  Tour.extend(Paths[tour[i]][tour[(i+1) % len(tour)]][0:-1])
 return rc,Value,Tour
```

Table 5.13: Result of the TSP* function on our example.

NB 12	0	1	9	4	9	2	7	6	8	3	5	0
Total dist 1064	0	56	171	19	31	21	140	55	94	128	207	142

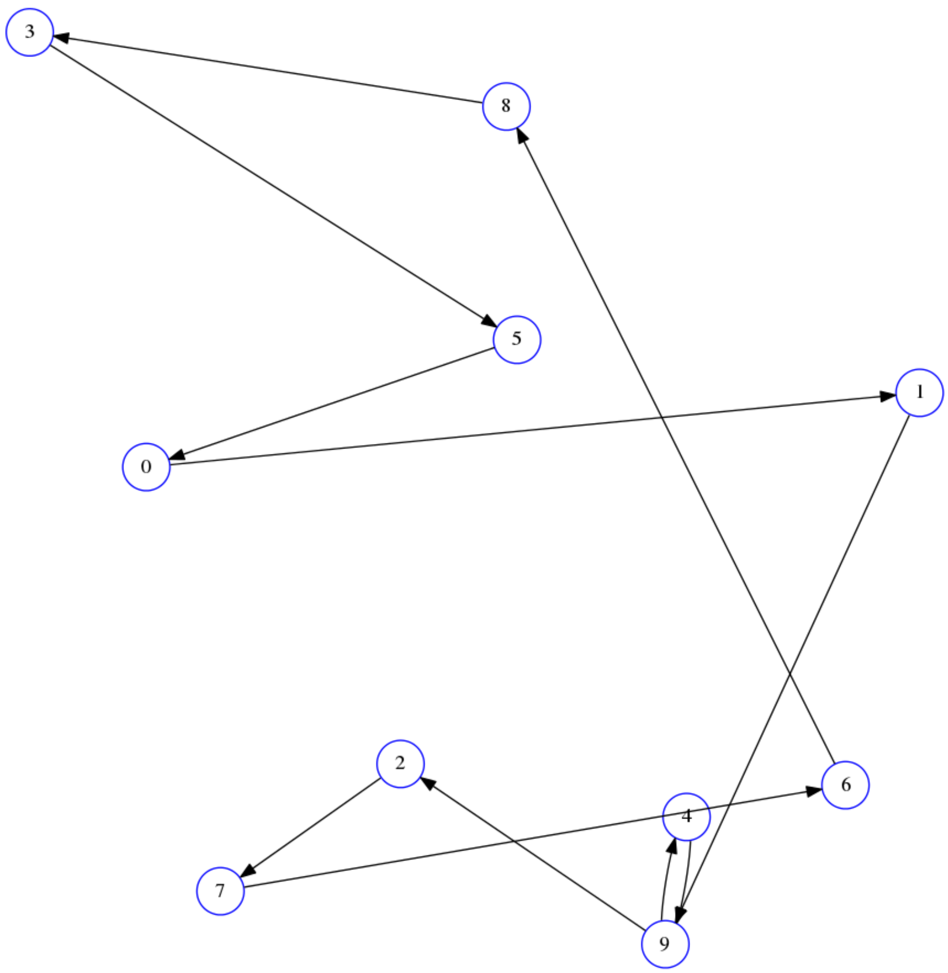


Figure 5.2: TSP* solution

Chapter 6

Mixed models

This section we develop models for problems requiring a mix of continuous and integral variables, as well as a variety of constraints. We cover some of the tricks optimizers have developped, over the years, to cajole and twist integer program solvers into providing solutions for problems that do not easily fit the Procrustean rules of classical optimization.

Traditionally these models have been called Mixed Integer Programs[1] (MIP).

[1] Another questionable moniker. What is mixed? The variables as some are continuous and other integral. Then should it not be simply a Mixed Progrem (MP), or a Mixed Continuous Integer Program MCIP?

6.1 Facility location

We will revisit the distribution problems we first encountered while discussing flows, but with a few added twists. Recall Section 4.2 where Solar-1138 inc. needed to decide which plant would distribute power to which city. Now let us assume, in addition, that Solar-1138 is in the planning stages and needs to decide first which plants to build to then distribute to cities. The data is similar. First we have the matrix of cost from each potential plant to each city as in Table 6.1

Table 6.1: Example of distribution cost

From/To	City0	City1	City2	City3	City4	City5	City6	Supply
Plant 0	20	23	23	24	28	25	13	544
Plant 1	19	18	30	25	19	17	14	621
Plant 2	29	13	19	17	22	15	11	635
Plant 3	16	23	29	22	29	26	11	549
Plant 4	23	20	10	27	23	19	20	534
Plant 5	21	12	23	29	14	15	22	676
Plant 6	13	18	22	13	11	25	23	616
Plant 7	21	12	20	20	20	13	11	603
Plant 8	24	24	29	17	18	16	20	634
Plant 9	28	11	22	26	25	19	11	564
Demand	553	592	472	495	504	437	634	

In addition, we have a cost to build each potential plant in Table 6.2. Since the cost of establishing the plants, as well as the cost of distribution, varies with the plants, we have a more complex problem on our hands than simple distribution. More complex both in terms of model and of solution technique.

Table 6.2: Example of plant building cost

Plant	0	1	2	3	4	5	6	7	8	9
Cost	5009	5215	6430	5998	4832	6365	6099	5499	5217	6153

The questions to answers are now: which plants to build and how much power to send from each plant to each city. We will forego any discussion of amortization, assuming that the appropriate calculations led to the fixed costs.

6.1.1 Constructing a model

6.1.1.1 Decision variables

Since we have two related but distinct decisions, we need to sets of decision variables. As with the previous distribution model, we will need to know how much from plant i goes to city j, therefore

$$x_{i,j} \quad \forall i \in P, \forall j \in C$$

As usual with distribution, the interpretation of say, $x_{2,4} = 5$ will be to send 5 units from plant 2 to city 4. This is a continuous variable (in our case); we consider it appropriate to send fractional values of power across the network. For some applications, it will be an integer variable.

Since we also need to know if plant i will be built, we need a binary variable,

$$y_i \in \{0,1\} \quad \forall i \in P$$

with the interpretation that $y_2 = 1$ will indicate that plant 2 must be built.

6.1.1.2 Objective

The objective has now two parts, traditionally known as the fixed and variable costs. The fixed costs are those related to the plant building; the variable costs are our distribution costs, as in (4.4), to which we add the fixed cost. As we build plant i only if variable y_i is one, assuming that the costs are in c_i,

$$\min \sum_i \sum_j C_{i,j} x_{i,j} + \sum_i c_i y_i$$

6.1.1.3 Constraints

As with the traditional minimum cost flow problem, we must have the supply and demand constraints,

$$\sum_j x_{i,j} \leq S_i \quad \forall i \in P \tag{6.1}$$

$$\sum_i x_{i,j} = D_j \quad \forall j \in C \tag{6.2}$$

Now we need to consider the new element of this model: how to link the variables related to plant building to those related to power distribution. We must not distribute from a plant that is not built and we must not build plants from which we will not distribute anything.

Since the objective function minimizes, it will tend to set all variable to zero, unless it cannot (assuming, of course, that costs are positive). We already know, from our work in distribution, that the $x_{i,j}$ variables will be properly set. Therefore we must ensure that the corresponding y_i are also set.

When should a particular y_i be one, indicating that plant i is to be built? When the optimal solution has some $x_{i,j}$ above zero for the same i and any j. This suggests a constraint of the form

$$\sum_j x_{i,j} \leq y_i \quad \forall i \in P$$

This constraint achieves half of our needs: no $x_{i,j}$ will be above zero if plant i is not build. Yet, as y_i is at most one, while the sum on the left side may be considerably larger, there is an inconsistency. The trick is to multiply y_i by a 'sufficiently' large constant, M. What would be large enough? The sum of the demands of all cities would certainly work since the sum of all the $x_{i,j}$ cannot possibly exceed all the demands. Therefore, assuming that we sum all demands is M,

$$\sum_j x_{i,j} \leq M y_i \quad \forall i \in P \tag{6.3}$$

Constraints of type (6.3) are known to optimizers as big-M constraints[2] and are to be used sparingly unless the constants M are small enough. Solvers, some more than others, may

[2] Yes, the reader can again fault optimizers sadly unimaginative nomenclature.

get into numerical trouble if the constants are too large. In practice this means, for the modeller: find the smallest possible M and try it. If the solver chokes, then find another modelling technique.

A different big-M approach may occur to the reader: since any of the $x_{i,j}$ non zero should trigger the corresponding y_i to be one, this set of constraints is possible:

$$x_{i,j} \leq Dy_i \quad \forall i \in P \forall j \in C$$

for some suitable multiplier D. Indeed, this solves our problem and considerably reduces the size of the multiplier at the cost of increasing the number of constraints. [3]

The model, as described above, will work, but we can improve it both by reducing M and by eliminating some constraints. Notice that constraints (6.1) and (6.3) have the same structure and the same left hand side. This condition suggests a merge of the constraints into

$$\sum_j x_{i,j} \leq S_i y_i \quad \forall i \in P \tag{6.4}$$

Indeed, had we given the size of M more thought, we might have concluded that S_i was the 'best' big-M to use.

6.1.1.4 Executable model

The reader will recognize most of this model as it is identical to Code 4.2. We will only highlight the differences.

The solver receives, in addition to the distribution cost matrix `D`, an array of fixed building costs, `F`.

Code 6.1: Facility Location model

```
1  def solve_model(D,F):
2    s = newSolver('Facility␣location␣problem', True)
3    m,n = len(D)-1,len(D[0])-1
4    B = sum(D[-1][j]*max(D[i][j] for i in range(m)) for j in range(n))
5    x = [[s.NumVar(0,D[i][-1],'') for j in range(n)] for i in range(m)]
6    y = [s.IntVar(0,1,'') for i in range(m)]
7    Fcost, Dcost = s.NumVar(0,B,''),s.NumVar(0,B,'')
8    for i in range(m):
9      s.Add(D[i][-1]*y[i] >= sum(x[i][j] for j in range(n)))
10   for j in range(n):
11     s.Add(D[-1][j] == sum(x[i][j] for i in range(m)))
12   s.Add(sum(y[i]*F[i] for i in range(m)) == Fcost)
13   s.Add(
14     sum(x[i][j]*D[i][j] for i in range(m) for j in range(n)) == Dcost)
15   s.Minimize(Dcost + Fcost)
16   rc = s.Solve()
17   return rc,ObjVal(s),SolVal(x),SolVal(y),SolVal(Fcost),SolVal(Dcost)
```

Line 9 links the decision to build a given plant to the amount transported from that plant as well as bounding the amount transported by the supply capacity of the plant.

[3] For the theretically-minded: there are cases where multiplying the number of constraints is actually preferable as it provides a tighter relaxation; this is not one of those cases.

The objective function at line 15 minimizes sum of the fixed building costs and the variable distribution costs.

Finally, we return all the transported material as well as the decisions to build. The solution to our example appears at Table 6.3 where we display only the transport from the plants included in the building decision.

Table 6.3: Optimal solution to the Facility Location

	City 0	City 1	City 2	City 3	City 4	City 5	City 6
Plant 0							280.0
Plant 3	549.0						
Plant 4			534.0				
Plant 5		52.0			624.0		
Plant 6	21.0			531.0	64.0		
Plant 7		78.0				206.0	319.0
Plant 9		564.0					
Plant 10			116.0			454.0	

6.1.2 Variations

- The main variation has to do with capacities. It may be that the path between producers and consumers has a maximum capacity, say $c_{i,j}$. This is trivially implemented by defining the variables with the appropriate domain, that is $0 \leq x_{i,j} \leq c_{i,j}$ or, in the executable, amend line 5 to read

  ```
  x = [[s.NumVar(0,C[i][j],'') for j in range(n)] for i in range(m)]
  ```

  In the capacited case, it may be worthwhile to re-visit the big-M constraint used to set the building decision variable and prefer an alternative approach based on the capacity of the flow out of the plants.

6.2 Multi-commodity flow

We have previously discussed flow problems and described them as ‘easy’ integer problems because the integrality comes for free: No need to even declare variables as integers; the optimal solution will be integral. But this is not the case when multiple *commodities* are carried by the same network, then we must expliciteley specify all variables that must be integral.

We can think of this problem as a series of transshipment problems overlaid on one network. Some nodes supply, some nodes demand, other can carry through and there is more than one element to carry so that a node acting as a supplier for one element can be a consumer for another. As such we will have a number of cost, demand and supply data Tables as 6.4, for a small instance. The goal is, as in transshipment, to satisfy all demands.

Table 6.4: Example of Multi-Commodity Flow cost matrices

Comm 0	N0	N1	N2	N3	N4	Supply
N0		20	23	23	24	532
N1	19		18	30	25	
N2				13	19	
N3	24	23			22	512
N4	23	10				
Demand		230	306		508	
Comm 1	N0	N1	N2	N3	N4	Supply
N0		23	29	14	15	533
N1	22		13	11	25	609
N2	20	20		13	11	634
N3		18	20		24	
N4			11	22		
Demand				354	1422	
Comm 2	N0	N1	N2	N3	N4	Supply
N0		30	17	19	30	
N1			21	19	27	564
N2				29	27	588
N3			12		27	
N4	27	15		16		
Demand	315			360	477	

6.2.1 Constructing a model

6.2.1.1 Decision variables

As it simply is a set of transshipment problems on the same network, we need a decision variable per problem. So, if we assume K commodities, and a set of N nodes,

$$x_{k,i,j} \in [0, 1, 2, \ldots] \quad \forall i \in N, \forall j \in N, \forall k \in K$$

If variable $x_{4,3,5}$ is 6, it will mean send 6 units of commodity 4 along arc $(3, 5)$.

6.2.1.2 Objective

The objective is a simple generalization of the transshipment objective,

$$\min \sum_k \sum_i \sum_j C_{k,i,j} x_{k,i,j}$$

6.2.1.3 Constraints

The constraints are the generalized conservation of flow, taking care that they must all hold for each commodity separately.

$$\sum_j x_{k,j,i} - \sum_j x_{k,i,j} = D_{k,i} - S_{k,i} \quad \forall k \in K, i \in N \tag{6.5}$$

6.2.1.4 Executable model

Let us translate this into an executable model. The function accpets a three-dimensional array `C` of costs on arcs for each of the commodity. It also accepts capacities in `D`, that can be either a scalar, indicating the same capacity on all arcs, or a two-dimensional array to specify a capacity per arc. Finally, it accepts a parameter `Z` to indicate, if `True`, that we must solve this as an integer program because the elements transported by the network are indivisible. If `False`, we are accepting fractional solutions.

Code 6.2: Multi-Commodity Flow model

```
def solve_model(C,D=None,Z=False):
  s = newSolver('Multi-commodity mincost flow problem', Z)
  K,n = len(C),len(C[0])-1,
  B = [sum(C[k][-1][j] for j in range(n)) for k in range(K)]
  x = [[[s.IntVar(0,B[k] if C[k][i][j] else 0,'') \
       if Z else s.NumVar(0,B[k] if C[k][i][j] else 0,'') \
       for j in range(n)] for i in range(n)] for k in range(K)]
  for k in range(K):
    for i in range(n):
      s.Add(C[k][i][-1] - C[k][-1][i] ==
          sum(x[k][i][j] for j in range(n)) - \
          sum(x[k][j][i] for j in range(n)))
  if D:
    for i in range(n):
      for j in range(n):
        s.Add(sum(x[k][i][j] for k in range(K)) <= \
           D if type(D) in [int,float] else D[i][j])
  Cost = s.Sum(x[k][i][j]*C[k][i][j] \
          for i in range(n) for j in range(n) for k in range(K))
  s.Minimize(Cost)
  rc = s.Solve()
  return rc,ObjVal(s),SolVal(x)
```

This code is essentially identical to the transshipment code, with some practical embellishments. The decision variable now has three dimension instead of two, the first one indicating which commodity, the last two, as usual, the arc. Moreover, if the parameter

Z is True it is defined as an integer variable. The reason for the choice is that, for a very large number of networks (the vast majority, in fact) and multicommodity flow problems, the variables can be declared as continuous and yet, if all demands, supplies and capacity are integral, so will the solution be. The modeller who is faced with long runtimes should try to relax the integrality constraint. It might very well be integral, saving oodles of CPU cycles. The conditions under which solutions will be integral are complex and difficult to verify ahead of time, which is why, for practical purposes, it is easier to try a continuous solver and adjust if the solution is not practical.

The solution to our simple problem is shown at Table 6.5. And, in fact, it was solved as a continuous problem.

Table 6.5: Optimal solution to the Multi-Commodity Flow

Comm 0	N0	N1	N2	N3	N4
N0		226	306		
N1					
N2					
N3		4			508
N4					
Comm 1	N0	N1	N2	N3	N4
N0					533
N1			155	354	100
N2					789
N3					
N4					
Comm 2	N0	N1	N2	N3	N4
N0					
N1				360	204
N2					588
N3					
N4	315				

6.2.2 Variations

6.2.2.1 All-pairs shortest paths (revisited)

Why did we cover multi-commodity flows if they are such trivial modifications to transshipment? Because they are often used by twisting a problem to fit the structure of multi-commodity flows[4]. Here is a simple example: we covered before the problem of finding shortest paths between all pairs of nodes in a network. We did it by solving a sequence of shortest paths model. It is possible to find all such paths by running a single multi-commodity flow, better yet, it will **never** require an integrality constraint, ensuring a very short runtime.

The trick is to consider every node to be a supplier of a particular commodity in quantity $n-1$. And every node is also a consumer of the $n-1$ commodities provided by the other nodes. Code 6.3 implements this approach. The code is slightly more general than that as it allows is to specify a set of source nodes for which we want emanating paths to all other nodes.

[4] There are a number of research papers on creating timetables using multi-commodity flows, for instance.

Running the code on the identical problem to the all pairs problem of Chapter Shortest Paths yields identical results. (See Table 4.11). The difference is that the flow code is much faster. It is, in fact, often much faster than running some specialized algorithm like Floyd-Warshall which has fixed complexity proportional to n^3 while our flow problem is often solved in time proportional to n.

Code 6.3: All-pairs shortest paths via Multi-Commodity Flow

```
def solve_all_pairs(D,sources=None):
  n,C = len(D),[]
  if sources is None:
    sources = [i for i in range(n)]
  for node in sources:
    C0 = [[0 if n in [i,j] else D[i][j] for j in range(n+1) ] \
          for i in range(n+1)]
    C0[node][-1] = n-1
    for j in range(n):
      if j!= node:
        C0[-1][j] = 1
    C.append(C0)
  rc,Val,x = solve_model(C)
  Paths = [[None for _ in range(n)] for _ in sources]
  Costs = [[0 for _ in range(n)] for _ in sources]
  if rc == 0:
    for source in sources:
      ix = sources.index(source)
      for target in range(n):
        if source != target:
          Path,Cost,node = [target],0,target
          while Path[0] != source and len(Path)<n:
            v = [j for j in range(n) if x[ix][j][node]>=0.1][0]
            Path.insert(0,v)
            Cost += D[v][node]
            node = v
          Paths[ix][target] = Path
          Costs[ix][target] = Cost
  return rc, Paths, Costs
```

We display on Table 6.6 the results of running the code and requesting the shortest paths emanating from nodes 0 and 2.

6.2.3 Instances

An interesting application appears in fiber optics networks. Consider a set of sources where signals emanate and a set of sinks that these signals must reach and a set of transshipment nodes that only carry signals (possibly boosting them if needed). It is possible for multiple signals to share the same cables at the same time if they use different wavelengths. The number of available wavelengths is limited. What we have here is multiple transshipment problems overlaid. The objective is, in this case, to maximize the number of established connections.

Table 6.6: Shortest paths from nodes 0 and from node 2 on Example `sp-example`

0-Target	Cost	[Path]					
1	46	[0,	1]				
2	17	[0,	2]				
3	24	[0,	3]				
4	51	[0,	4]				
5	48	[0,	2,	5]			
6	52	[0,	2,	5,	6]		
7	41	[0,	3,	7]			
8	68	[0,	2,	8]			
9	55	[0,	3,	7,	9]		
10	83	[0,	3,	7,	9,	10]	
11	99	[0,	2,	5,	11]		
12	89	[0,	3,	7,	9,	10,	12]
2-Target	Cost	[Path]					
0	79	[2,	5,	0]			
1	38	[2,	1]				
3	68	[2,	5,	6,	3]		
4	34	[2,	4]				
5	31	[2,	5]				
6	35	[2,	5,	6]			
7	58	[2,	5,	7]			
8	51	[2,	8]				
9	66	[2,	5,	9]			
10	88	[2,	5,	10]			
11	82	[2,	5,	11]			
12	94	[2,	5,	10,	12]		

6.3 Staffing level

We discuss here a problem that is often described by optimizers as a staff scheduling problem. But that is a terrible misnomer as no schedule is ever produced; only requirement levels. The true staff scheduling problem as it is known to practitioners is orders of magnitude more complex. So we will call this problem the *staffing level* problem.

The situation is the following: We have a grid, indexed in one dimension by time intervals. Those are most commonly presented in either days (Monday, Tuesday) or hours (8AM, 9AM) but could be in any valid time units. In the other dimension we have what are usually called shifts, units of a worker's schedule (Monday to Friday, Tuesday to Sunday or 8AM to 4PM, 9AM to 5PM). Associated with each time interval we have staffing needs (need 45 people on Monday or 62 people at noon). Finally we also have a cost associated to each shift. See Table 6.7 for an example.

Table 6.7: Staffing requirement matrix

	Shift 0	Shift 1	Shift 2	Shift 3	Shift 4	Shift 5	Shift 6	Need
00h	1			1				15
02h	1					1		17
04h	1					1		16
06h	1	1						17
08h		1					1	19
10h		1					1	18
12h		1	1					11
14h			1					12
16h			1					15
18h			1	1				14
20h				1	1			20
22h				1	1			18
Cost	\$69.37	\$67.03	\$64.55	\$72.06	\$29.24	\$21.67	\$24.52	

Note that the example includes two types of shifts, full-time (eight hours) and part time (four hours). The costs and the number of hours per shift are different. The goal is to determine, at the least total cost, how many people will work each shift, ensuring that the needs are satisfied and, also that the full-time workers get some form of preferences. This preference can take on various forms. It can be as simple as 'No part-time if there is no full-time working', or 'Every full-time shift must be staffed by at least x people' or 'We have a pool of x full-time staff who must work; fill the rest of the needs with part-time'. We will discuss some possible variations on these constraints.

Note that a solution to this problem determines only the staffing level of each shift. This is not a schedule. The true schedule, in the sense of which worker works which shift, is left to a further model, a much more complicated one.

The reader should notice here a common approach of optimizers for complex problems: decomposition. The staffing needs (the last column of Table 6.7) was conceivably obtained by some model; the levels will be produced by the model we are now creating; and finally, the proper schedules will be generated by yet another model. This approach will unlikely produce the overall optimal solution. Yet it is used in all industries. The airlines are notorious for decomposing problems. There are two reasons we approach complex problems this way, a good reason and a bad one.

The good reason is that, often, attacking the whole problem is not technologically feasible; we can write the whole problem model, certainly, but no solver can find a solution before the solar system expires. Another aspect of the same reason is that one part of the whole problem is best modeled as an integer program while another is much simpler as a constraint program. There are yet no entirely satisfying way to write such hybrid models.[5] The second reason, the bad one, is that, even in cases where a more holistic approach is feasible, there is massive inertia in organizations that would prevent its implementation.[6]

6.3.1 Constructing a model

6.3.1.1 Decision variables

What we need to decide in this problem is how many people working each shift. Because of the distinction between full-time and part-time shifts, it may be useful to distinguish between people working full-time and people working part-time. Hence, assuming N_f and N_p for the sets of full-time and part-time shifts,

$$x_i \in [0, 1, 2, \ldots] \quad \forall i \in N_f \cup N_p =: N$$

6.3.1.2 Objective

The objective is simple since we have a cost per shift C_i,

$$\min \sum_{i \in N} C_i x_i$$

6.3.1.3 Constraints

Let us tackle the staffing needs by looking at a specific time interval, say 0600h with a need of 17 people. If shift 0 covers from midnight to 8AM, and shift 1 covers 6AM to 2PM both of those shifts, and no other, must, together, have at least 17 people to cover the 6AM need. This means, algebraically $x_0 + x_1 \geq 17$.

Now, in more general terms, consider a matrix $M_{T \times N}$ where T is the set of time intervals, with the structure of Table 6.7 (without the last row and column). What we need is that, for every time interval t, the sum of the x_i that cover t is at least the required R_t. Algebraically

$$\sum_{i \in N} M_{t,i} x_i \geq R_t \quad \forall t \in T \tag{6.6}$$

This is enough for the simplest staffing level model. If there is no distinction between full and part time, we are done. Let us consider a few realistic constraints.

1. *Minimum full-time staff*

   If we have a requirement that, for every full-time shift i, there be at least Q_i people working, we can add

   $$x_i \geq Q_i \quad \forall i \in N_f$$

[5] It is being done, piecemeal, by very talented modelers, willing to dig deep into the bowels of solvers but there is no easy way to do this in general yet. Ask me again in five years; or-tools is very close to that goal.

[6] The paper pushers responsible for one aspect of the problem will fight tooth and nail the paper pushers responsible for another aspect, lest they relinquish part of their fiefdom. I have seen massive savings of time and money squandered in internecine turf wars. Optimizers beware, the problem is often between the user keyboard and the screen!

Because of the structure of the shifts (full-time shifts cover all time intervals), this constraint also satisfies a requirement of the form "No part-time if no full-time working" (assuming that $Q_i > 0$ of course).

2. *Full-time must work*

   A possibly more realistic requirement is that, given a pool of P full-time people, we require that they all work. Only if the staffing requirements cannot be met should we involve part-time staff. This is accomplished by

$$\sum_{i \in N_f} x_i = P \tag{6.7}$$

   This constraint will be infeasible unless the number of full-time workers is below or at the staffing needs, of course.

3. *No part-time if no full-time present*

   Consider a particular time interval, say 10AM to noon. Say it can be staffed by full-time shift 1 and 2 and also by part-time shift 6. We want to ensure that x_6 is zero unless either x_1 or x_2 is non-zero. As a first attempt, let us try

$$x_1 + x_2 \geq x_6 \tag{6.8}$$

   This ensures that if there are no full-time staff in either full-time shift, then there will be no part-time staff either. But it constrains the problem more than necessary as it prevents the number of part-time staff to exceed the full-time staff. What we need is to scale up the left-hand side of Equation (6.8) by some 'large enough' constant.

   This is again an instance of the technique optimizers call the big-M method. How large is large enough? The sum of the required staff is surely enough and yet, not so large as to cause numerical problem. Therefore

$$\sum_{j \in T} R_j \sum_{i \in N_f} M_{t,i} x_i \geq \sum_{i \in N_p} M_{t,i} x_i \quad \forall t \in T \tag{6.9}$$

   For each time interval, equation (6.8) sums, on the left, the number of full-time workers, scaled up by a large constant. On the right it sums the number of part-time workers.

6.3.1.4 Executable model

The model will receive a time vs shift 0-1 matrix `M`, exactly as Table 6.4, along with the number of shifts considered full-time. This number will be equal to the number of columns of `M` if the problem has no part-time staff. The last column indicates minimal requirements for each time interval. The last row indicates cost of one worker in the indicated shift.

To make this a more practical model, it may be given an array `Q`, indexed by shifts indicating the minimum number of people in each shift. It may also be given a scalar P of full-time people who must work. Finally it may also be given a flag `no_part` which, if `True`, indicates that there will be no part-time people working unless full-time people are present.

Code 6.4: Staffing model

```
1  def solve_model(M,nf,Q=None,P=None,no_part=False):
2    s = newSolver('Staffing', True)
3    nbt,n,B = len(M)-1,len(M[0])-1,sum(M[t][-1] for t in range(len(M)-1))
4    x = [s.IntVar(0,B,'') for i in range(n)]
5    for t in range(nbt):
6      s.Add(sum([M[t][i] * x[i] for i in range(n)]) >= M[t][-1])
7    if Q:
8      for i in range(n):
9        s.Add(x[i] >= Q[i])
10   if P:
11     s.Add(sum(x[i] for i in range(nf)) >= P)
12   if no_part:
13     for t in range(nbt):
14       s.Add(B*sum([M[t][i] * x[i] for i in range(nf)]) \
15             >= sum([M[t][i] * x[i] for i in range(nf,n)]))
16   s.Minimize(sum(x[i]*M[-1][i] for i in range(n)))
17   rc = s.Solve()
18   return rc, ObjVal(s), SolVal(x)
```

After extracting the number of time intervals, number of shifts and computing a bound for the decision variables on line 3, we define our decision variable as a non-negative integer. The bound we use is clearly valid as it is the sum of all required staff.

We implement the basic constraint of Equation (6.6) at line 6. Then, the three following `if` statements implement the optional constraints of *Minimum full-time staff*, *Full-time must work*, and *No part-time if no full-time present*.

Results of the basic constraint only are shown at Table 6.8 while setting the `no_part` flag to `True` yields Table 6.9. Note that, for this example, the total number of people working has not changed, but the distribution of people among shifts has changed, raising the total cost.

Table 6.8: Optimal solution to the Basic Staffing Level problem

$3187.84	Shift 0	Shift 1	Shift 2	Shift 3	Shift 4	Shift 5	Shift 6
Nb:71	15	2	15	0	20	2	17

Table 6.9: Optimal solution with *no part-time unless full-time* constraint

$3225.47	Shift 0	Shift 1	Shift 2	Shift 3	Shift 4	Shift 5	Shift 6
Nb:71	14	3	15	1	19	3	16

6.3.2 Variations

We have covered some variations by implementing optional parameters already. There are as many other variations as there are companies using staffing levels. The next step involves transforming this into a real scheduling model, assigning specific people to shifts. We will begin to tackle this later, but the reader is encouraged to ponder ways to implement true scheduling.

6.4 Cutting stock

The cutting stock problem originated in the paper and textile industries. A typical mill produces paper in very large product rolls (large in length and in width), which are then cut to the width required by the customers. We will call the latter consumer rolls. For instance tabloid printers want their paper in 17in width rolls, while broadsheet printers want twice that width. These consumer rolls are cut to page size after printing. The cutting problem, for the paper producer, is to cut the large width rolls to satisfy customer demands while minimizing waste.

Table 6.10 is a random instance of orders we will solve to illustrate the model. The first column is the number of consumer rolls in the order and the second column is the required width that has been pre-processed to be measured in percentage of the product roll width.

Table 6.10: Example of Cutting Stock problem

Order	Nb rolls	% Width
0	6	25
1	12	21
2	7	26
3	3	23
4	8	33
5	2	15
6	2	34

We want to minimize paper waste, which really means minimizing the total number of product rolls used. But, in order to do the cutting, we will need more information, namely, giving any product roll, how is it to be cut? Say we have a number of orders for 21% and for 36%, do we cut at 21%, 42%, 63% and 99% with a waste of 1% or at 36%, 72% and 93% for a waste of 7%? It may be that we need to cut half our rolls with the first pattern and half with the second to satisfy all demands. Which patterns to cut and how many rolls of a given pattern are the key problems.

6.4.1 Constructing a model

Originally the problem was attacked by specifying various cutting patterns, sometimes statically, sometimes dynamically. This was more or less forced by the technology of the time. Things have changed; both processors and solvers are considerably faster. Moreover, a good modeler should try the simplest approach first and make it more complex if and only if it fails. With this in mind, we will leave to the solver the problem of deciding on the patterns.

6.4.1.1 Decision variables

If we leave the pattern decision to the solver, the problem becomes, given a roll, where do we cut? But this is over-specifying. The pattern 21, 42, 63 and 99 is not distinguishable from the pattern 21, 57, 78 and 99. Those satisfy exactly the same customer demands, three 21% and one 36% width. And we know that having multiple indistinguishable solutions is very bad for a solver. Therefore, what we should ask is really, given a roll, how many cuts of with w do we make?

With this in mind, assuming that we have N orders and at most K rolls, a reasonable decision variable is

$$x_{i,j} \in [0, 1, 2, \ldots] \quad \forall i \in N, \forall j \in K$$

where $x_{2,5} = 7$, for instance, will mean that we cut roll 5 to satisfy 7 customer rolls of with specified in order 2. The ordering of the cuts is irrelevant but we might very well post-process the solution to produce cutting patterns to be used.

And since we do not know ahead of time how many rolls we will need, there has to be a corresponding variable to indicate roll use,

$$y_j \in \{0, 1\} \quad \forall j \in K$$

Where $y_5 = 1$ means that roll 5 (out of a possible K) is used.

The maximum number of rolls, K, does not have to be precisely determined; any upper bound will do at this point.

6.4.1.2 Objective

The objective is to minimize the number of rolls. We could minimize the sum of all the y_j, but that leaves open the possibility of a solution that says “Use rolls 1 and 3, but not 2.” To avoid such annoyance, let us use a small trick which will make each new roll more costly that the last,

$$\min \sum_{j \in K} j * y_j$$

But now, our objective function value at optimality will not represent the number of rolls used, so we introduce an auxiliary variable, say t, and add

$$t = \sum_j y_j$$

6.4.1.3 Constraints

We have two types of constraints. The first is to ensure that customer demands are satisfied. So, across all rolls used, we must verify that we cut enough rolls of the required order quantity, say b_i,

$$\sum_{j \in K} x_{i,j} \geq b_i \quad \forall i \in N$$

The second type of constraints is to make sure that the consumer rolls we cut off the product roll do not exceed the width of that large roll, or, assuming that order i has width w_i

$$\sum_{i \in N} x_{i,j} \leq 100 \quad \forall j \in K$$

But that is incomplete since we need to connect the x and y variables. No $x_{i,j}$ should be positive if the corresponding y_j is zero. We could introduce a large number of constraint or, realizing that we have encountered this situation before, simply modify the last constraint to read:

$$\sum_{i \in N} x_{i,j} \leq 100 y_j \quad \forall j \in K$$

6.4.1.4 Executable model

Let us translate this into an executable model. It accepts a matrix `D`, exactly as Table 6.10.

Code 6.5: Cutting Stock model (solver finds the patterns)

```
1  def solve_model(D):
2    s,n = newSolver('Cutting␣Stock', True), len(D)
3    k,b = bounds(D)
4    y = [s.IntVar(0,1,'') for i in range(k[1])]
5    x = [[s.IntVar(0,b[i],'') for j in range(k[1])] for i in range(n)]
6    w = [s.NumVar(0,100,'') for j in range(k[1])]
7    nb = s.IntVar(k[0],k[1],'')
8    for i in range(n):
9      s.Add(sum(x[i][j] for j in range(k[1])) >= D[i][0])
10   for j in range(k[1]):
11     s.Add(sum(D[i][1]*x[i][j] for i in range(n)) <= 100*y[j])
12     s.Add(100*y[j] - sum(D[i][1]*x[i][j] for i in range(n)) == w[j])
13     if j < k[1]-1:
14       s.Add(sum(x[i][j] for i in range(n)) >= \
15             sum(x[i][j+1] for i in range(n)))
16   Cost = s.Sum((j+1)*y[j] for j in range(k[1]))
17   s.Add(nb == s.Sum(y[j] for j in range(k[1])))
18   s.Minimize(Cost)
19   rc = s.Solve()
20   rnb = SolVal(nb)
21   return rc,rnb,rolls(rnb,SolVal(x),SolVal(w),D),SolVal(w)
```

At line 3 we call a routine `bounds` to find lower and upper bounds on the numbers of rolls we will require and on the number of rolls of each order that can fit in a roll. The upper bound on the number of rolls used on line 4 to create as many `y` as we could possibly need (and also on subsequent constraints related to each roll). The bound of the number of rolls of each type is used at the next line to establish the range for each `x`: on any given roll, we can have zero up to the number of elements that can fit or the number required by the customer, hence the `min` expression.

Line 9 ensures that we satisfy every customer demand by summing over all rolls the elements of a given order. We could also have used an inequality here, namely $\geq$. The idea is that we may cut more than the customer demanded and we will store these into inventory, awaiting the next order. This sometimes makes sense, but is often better left out of the model. Once we have a solution that satisfies exactly customer demand, someone with an understanding of the situation and in possession of planning tools can decide how to best cut the leftover of the rolls. In either case, this will not change the total number of rolls used.

Line 11 ensures that the consumer rolls cut off a roll do not add up to more than 100% of the roll. And the next line, not a constraint, simply, computes the waste of each roll to help return a meaningful solution.

The loop starting at 13 breaks the symmetry of multiple solutions that are equivalent for our purposes: any permutation of the rolls. These permutation, and there are $K!$ of them, cause most solvers to spends an exorbitant time solving. With this constraint, we tell the solver to prefer those permutations with more cuts in roll j than in roll $j+1$. The reader is encouraged to solve a medium-sized problem with and without this symmetry-breaking constraint. I have seen problems take 48 hours to solve without the constraint and 48 minutes with. (Of course, for problems that are solved in seconds, the constraint will not help; it may even hinder. But who cares if a cutting stock instance solves in two or in three seconds. We care much more about the difference between two minutes and three hours, which is what this constraint is meant to address.

We could use, for objective function, the simple sum of the rolls used variable, `y` and pre-process the unused rolls. But we used a little trick to make each additional roll more 'expensive' by including an ordinal factor. This guarantee that if, say the number of rolls is estimated to be between 11 and 14 and we end up using 12, they will be the first 12. There will be no 'holes'.

There are alternative objective functions. For example, we could have minimized the sum of the waste. This makes sense, especially if the demand constraint is formulated as an inequality. Then minimizing the sum of waste will spend more CPU cycles trying to find more efficient patterns that over-satisfy demand. This is especially good if the demand widths re-occur regularly and keeping cut rolls in inventory to satisfy future demand is possible. Beware that the running time will grow quickly with such an objective function.

Finally we rearrange the solution to make sense for the caller. Instead of our decision variables `x` and `y`, we return a list of all rolls, with the cutting patterns and the waste of each.

As we need bounds on the number of rolls and on the maximum number of cuts satisfying a given order on a single roll, Code 6.6 implements a simple heuristic. The minimum is clearly the sum of all widths divided by the width of a roll, 100, as we assumed all width to be entered as percentages. The upper bound is computed by a first fit heuristic: we add each roll, in order, to a roll until no more fits. Then we start a new roll. This is not brilliant, but it serves its purpose.

Code 6.6: Cutting Stock bounds computation

```
def bounds(D):
  n, b, T, k, TT = len(D), [], 0, [0,1], 0
  for i in range(n):
    q,w = D[i][0], D[i][1]
    b.append(min(D[i][0],int(round(100/D[i][1]))))
    if T+q*w <= 100:
      T,TT = T+q*w,TT + q*w
    else:
      while q:
```

```
    if T+w <= 100:
     T,TT,q = T+w,TT+w, q-1
    else:
     k[1],T = k[1]+1, 0
 k[0] = int(round(TT/100+0.5))
 return k, b
```

Code 6.7 reformats the solution to make it more meaningful to the caller. It returns an array containing for each roll used, the percentage of waste incurred on that particular roll as well as the cut pattern used. Of course the indicated cut pattern could be permuted with no change to the waste percentage. Output of our small example can be seen in Table 6.11.

Code 6.7: Cutting Stock model solution post-process

```
def rolls(nb, x, w, D):
 R,n = [],len(x)
 for j in range(len(x[0])):
  RR=[abs(w[j])]+[int(x[i][j])*[D[i][1]] for i in range(n) \
           if x[i][j]>0]
  R.append(RR)
 return R
```

Table 6.11: Optimal solution to the Cutting Stock

rolls	Waste 85.0	Pattern
0	5.0	{ 26; 23; 23; 23 }
1	16.0	{ 21; 21; 21; 21 }
2	4.0	{ 25; 25; 25; 21 }
3	4.0	{ 21; 21; 21; 33 }
4	1.0	{ 21; 26; 26; 26 }
5	1.0	{ 21; 26; 26; 26 }
6	0.0	{ 25; 21; 21; 33 }
7	0.0	{ 33; 33; 34 }
8	45.0	{ 25; 15; 15 }
9	1.0	{ 33; 33; 33 }
10	8.0	{ 25; 33; 34 }

6.4.2 Pre-allocate cutting patterns

The previous approach is optimal but will not scale very well, even with our symmetry-breaking constraints. We will describe here a, generally sub-optimal approach that can be used to solve much larger instances.

The basic idea is to fix the cutting patterns and only optimize the number of rolls using those patterns while satisfying demand. Imagine, for instance, that we were given the distinct patterns of the last column of Table 6.11 in a matrix `A`, as well as Table 6.10 in a array `D`. Then we could have a decision variable `y` indexed by the patterns of `A`, representing how many rolls to cut according to that pattern. Symbolically, we would have the model

$$\min \sum_j y_j$$
$$A_{j,i} y_j \geq D_i \quad \forall i$$
$$y_j \in [0, 1, 2, \ldots]$$

It seems simple enough, until one carefully considers the number of patterns. If we do not know ahead of time which patterns to use, then the simple option seems to be to list them all. How many are there? An astronomical number, even for small examples.

So, here is the brilliant idea: start with a certain set of patterns, optimize. Then, in a manner yet to be determined, find some 'better' patterns to add. In all its generality, this is known to optimizers[7] as *column generation*. . Repeat the optimization until we can no longer find 'better' patterns or until we run out of time or are satisfied with the solution. Code 6.8 implements the high-level approach.

Code 6.8: Cutting Stock model (using given patterns)

```
1  def solve_large_model(D):
2   n,iter = len(D),0
3   A = get_initial_patterns(D)
4   while iter < 20:
5    rc,y,l = solve_master(A,[D[i][0] for i in range(n)])
6    iter = iter + 1
7    a,v = get_new_pattern(l,[D[i][1] for i in range(n)])
8    for i in range(n):
9     A[i].append(a[i])
10  rc,y,l = solve_master(A,[D[i][0] for i in range(n)],True)
11  return rc,A,y,rolls_patterns(A, y, D)
12
13 def solve_master(C,b,integer=False):
14  t = 'Cutting␣stock␣master␣problem'
15  m,n,u = len(C),len(C[0]),[]
16  s = newSolver(t,integer)
17  y = [s.IntVar(0,1000,'') for j in range(n)] # right bound?
18  Cost = sum(y[j] for j in range(n))
19  s.Minimize(Cost)
20  for i in range(m):
21   u.append(s.Add(sum(C[i][j]*y[j] for j in range(n)) >= b[i]))
22  rc = s.Solve()
23  y = [int(ceil(e.SolutionValue())) for e in y]
24  return rc, y, [0 if integer else u[i].DualValue() for i in range(m)]
```

At line 4 we loop on the optimization a certain number of times. This is an easy termination criteria, which we can tune according to the size of the problem we are solving and how long we are willing to wait. There are better criteria, including looping until we have a true optimal solution, but these would lead us deep into the theory of optimization [8]

[7]The expression column generation stemmed, in the minds of optimizers, as literary as ever, from looking at the set of constraints as a matrix.

[8]The interested reader should look up 'Reduced Cost' and 'Column generation of the Cutting Stock' to pursue these matters.

The master problem is essentially the one described above: given the set of allowable patterns, do your best to minimize the number of rolls. There are two subtleties: The first one is that we solve the optimization problem as a linear program, not an integer program, even though we really want an integer solution, the number of rolls. We do this to gain speed. At the end, we simply round up the number of rolls, since obviously, if 4.6 rolls satisfy demand, then surely 5 rolls also satisfy demand.

The second subtlety involves the information we need to find better patterns. Consider one imaginary example of a constraint of the form of line 21 : for, say product roll 5, imagine we need 28 consumer rolls to satisfy demand. Given the patterns we have already, we might have that pattern 1 has this roll 3 times, pattern 3 has it 5 times and pattern 10 has it once (and no other pattern has roll 5). So the constraint reads

$$3y_1 + 5y_3 + 1y_{10} \geq 28$$

where the solution is y. What is the effect of changing the 28 by exactly one unit, keeping everything else constant? It will change the optimal solution by a small value, the marginal value of roll 5. We can do this, conceptually, for every roll and get each of their marginal value. By design, all solvers already have these marginal value computed; they are a side-effect of the solution techniques. So we simply request them at line 24 by the call for `DualValues`.

At the end, we reformat the solution to make it meaningful to the caller. We return an array containing for each roll used, the waste incurred on that particular roll as well as the cut pattern used.

Code 6.9: Cutting Stock model (getting a new pattern)

```
1 def get_new_pattern(l,w):
2   s = newSolver('Cutting stock slave', True)
3   n = len(l)
4   a = [s.IntVar(0,100,'') for i in range(n)]
5   Cost = sum(l[i]*a[i] for i in range(n))
6   s.Maximize(Cost)
7   s.Add(sum(w[i]*a[i] for i in range(n)) <= 100)
8   rc = s.Solve()
9   return SolVal(a), ObjVal(s)
```

The model to find a new pattern for the master model to optimize over (Code 6.9) uses the marginal value of each roll, provided by the solution to the master model above and maximizes the sum of the values times the number of occurrence of that roll in a patter, while ensuring that the pattern stays within the total width of the large roll at line 7. This is a knapsack problem and will be solved very fast.

Code 6.10: Cutting Stock model (getting a new pattern)

```
def get_initial_patterns(D):
  n = len(D)
  return [[0 if j != i else 1 for j in range(n)] for i in range(n)]

def rolls_patterns(C, y, D):
  R,m,n = [],len(C),len(y)
  for j in range(n):
    for _ in range(y[j]):
```

```
    RR=[]
    for i in range(m):
      if C[i][j]>0:
        RR.extend([D[i][1]]*int(C[i][j]))
    w=sum(RR)
    R.append([100-w,RR])
  return R
```

We have left two elements undescribed: the initial patterns, and the re-shaping of the solution to make it meaningful. From Code 6.10, the initial patterns must be such that they will allow a feasible solution, that is one satisfying all demands. We could be very creative here or not. Considering the already complex model, we chose the latter route: our initial patterns have exactly one roll per pattern. As obviously feasible as inefficient.

A solution to our small example with this column generation approach is shown in Table 6.12. Note that it may use more rolls, but it actually cuts each rolls very efficiently; it simply over-satisfies demands, unsurprising as we rounded up.

Table 6.12: Possibly sub-optimal solution to the Cutting Stock using column generation

rolls	Waste 1	Pattern
0	0	{ 33; 33; 34 }
1	0	{ 25; 21; 21; 33 }
2	0	{ 25; 21; 21; 33 }
3	0	{ 25; 21; 21; 33 }
4	0	{ 25; 21; 21; 33 }
5	0	{ 25; 21; 21; 33 }
6	0	{ 26; 26; 33; 15 }
7	0	{ 26; 26; 33; 15 }
8	0	{ 25; 26; 26; 23 }
9	0	{ 25; 26; 26; 23 }
10	1	{ 21; 21; 23; 34 }

6.5 Non-convex trickery

When discussing piecewise objectives (in Section 3.1), we noted that if the function was not convex, then the approach suggested did not work. The situation was illustrated with a cost function modeling economies of scale, that is: unit prices decreased as we increased the number of units. For example Table 6.13.

Table 6.13: Example of non-convex piecewise function

(From	To]	Unit cost	(Total cost	Total cost]
0	194	18	0	3492
194	376	16	3492	6404
376	524	14	6404	8476
524	678	13	8476	10478
678	820	11	10478	12040
820	924	6	12040	12664

If we tried to do something even as simple as minimizing this function, subject to $x \geq 250$ with our modal approach, we would get Table 6.14, which clearly does not solve our problem since

$$\begin{aligned} f(250) &= 194 \cdot 18 + (250 - 194) \cdot 16 \\ &= 3492 + 56 \cdot 16 \\ &= 3492 + 896 \\ &= 4388 \end{aligned}$$

Of course, if we had tried to maximize, it would have worked. The easy cases are minimizing convex functions and maximizing concave functions [9]. We will discuss here the hard cases.

Recall that the approach was to introduce decision variables λ_i, one per break in the function, to indicate which segment contains the optimal point and where on that segment the point is located (by the convex combination $x = \lambda_i P_i + \lambda_{i+1} P_{i+1}$. The model then became:

$$\begin{aligned} \min \sum_{i=1}^{n} \lambda_i \sum_{j=1}^{i} (B_j - B_{j-1}) \times C_{j-1} \\ \sum_i \lambda_i = 1 \\ x = \sum_i \lambda_i B_i \\ \lambda_i \in [0,1] \\ \text{and other constraints} \end{aligned}$$

[9] Optimizers working in engineering or applied mathematics traditionally minimize convex functions. There is a whole area of research appropriately called *Convex Analysis* devoted to the theory of such problems. In contrast, optimizers in business usually maximize concave functions. The theory is the same, but everything is upside down. Maybe we should call one group the optimizers and the other the pessimizers?

Table 6.14: Incorrect solution to non-convex piecewise objective with $x \geq 250$

Interval	0	1	2	3	4	5	6	Solution
δ_i	0.7294	0.0	0.0	0.0	0.0	0.0	0.2706	$\sum \delta$=1.0
x_i	0	194	376	524	678	820	924	x=250.0
$f(x_i)$	0	3492	6404	8476	10478	12040	12664	Cost=3426

The problem with this model is that, even though at optimality the sum of the λ_i is one, we have two non adjacent λ_i being non-zero. We must have two adjacent λ_i non-zero to determine which segment of the function to consider. We achieve this condition by introducing another set of binary variables, say $\delta_i \in \{0, 1\}$, summing to one, and add the conditions

$$\begin{aligned}
\lambda_0 &\leq \delta_0 \\
\lambda_1 &\leq \delta_0 + \delta_1 \\
\lambda_2 &\leq \delta_1 + \delta_2 \\
\lambda_3 &\leq \delta_2 + \delta_3 \\
&\ldots \\
\lambda_{n-1} &\leq \delta_{n-2} + \delta_{n-1} \\
\lambda_n &\leq \delta_{n-1}
\end{aligned}$$

See what happens when one of the δ_i is one: exactly two of the above inequalities, adjacent to each other, will have a right hand side of one. Hence exactly two λ_i, adjacent, will be allowed to be non-zero.

This approach (two layers of binary variables) works with all integer solvers, but the situation of "At most two adjacent variables non-zero" occurs so often in practice that some solvers have special code to handle it. These variables are known as SOS2 (Special Ordered Set of type 2)[10]. Which suggest the question: Is there a type 1? Indeed, there is: "Exactly one variable non-zero." But let us consider some useful generalizations with special cases as SOS1 and SOS2.

6.5.1 Selecting k variables out of n to be non-zero.

Consider a situation where we have a set of n variables $x_i \in [0, u_i]$ and we want to allow exactly k to be non-zero. For instance if you are considering investing in various projects and are setting up a model to choose the best, say k of them. (If $k = 1$ we have the so-called SOS1 case.) We introduce n binary variables, λ_i and add the constraints

$$x_i \leq u_i \lambda_i \quad \forall i \tag{6.10}$$

$$\sum_i \lambda_i = k \tag{6.11}$$

Replace the equality in (6.11) with $\leq$ if we want 'at most' and with $\geq$ if we want 'at least'. Since this occurs regularly in modeling, let us create a general function that we might use within any given model.

[10] Yet another example of the sadly unimaginative naming tradition of optimizers. Maybe this explains it. My Ph.D. advisor, said, only half in jest: "Do not name your algorithms with interesting names if you ever want them to be known by your name." The implication, was that, if one names his algorithms Alg-1 and Alg-2 or something equally pedestrian, colleagues will have no choice but to refer to them as Smith-star or Jones-revised. Alas, such hope at posterity is belied by the use of SOS2 and other mutts of the same ilk.

Code 6.11: How to select k out of n variables

```
1  def k_out_of_n(solver,k,x,rel='=='):
2    n = len(x)
3    binary = sum(x[i].Lb()==0 for i in range(n)) == n and \
4             sum(x[i].Ub()==1 for i in range(n)) == n
5    if binary:
6      l = x
7    else:
8      l = [solver.IntVar(0,1,'') for i in range(n)]
9      for i in range(n):
10       if x[i].Ub() > 0:
11         solver.Add(x[i] <= x[i].Ub()*l[i])
12       else:
13         solver.Add(x[i] >= x[i].Lb()*l[i])
14   S = sum(l[i] for i in range(n))
15   if rel == '==' or rel == '=':
16     solver.Add(S == k)
17   elif rel == '>=':
18     solver.Add(S >= k)
19   else:
20     solver.Add(S <= k)
21   return l
```

We craft Code 6.11 to handle multiple cases. First, we need to single out binary variables, as they do not need the additional layer of variables. We detect the binary case on line 4 by checking if all lower bounds are zero and all upper bounds are one. Any one variable not satisfying these conditions will set `binary` to `False`.

In the binary case, we simply rename the parameter `x` to be `l`; in the other cases, we create the binary variable array `l` and then set the forcing bound of Equation (6.10) at line 11 if $x \in [0, u]$ and, correspondingly at line 13 if $x \in [l, 0]$.

Finally we set one of three relations on `l`, depending on whether the caller wants 'exactly', 'at most' or 'at least' k variables selected. Beware that the relation '¿=' means 'at least k variables are *allowed* to be non-zero'. It does not means that k variables *will* be non-zero [11]

The reader might remember that in the first chapter (Section 2.1), when discussing variations, we left unsatisfied requirements of the form: *If food 3 is used, then food 4 must not be (and vice versa).* This exclusive-or is now easily accommodated. Recall from Code 2.1 that our food selection was using decision variable `f`. Then we can add one line to the model,

```
k_out_of_n(s, 1, [f[3],f[4]])
```

where the foods 3 and 4 are inserted into an array to be passed to our newly-minted routine.

6.5.2 Selecting k adjacent variables out of n to be non-zero..

If we want to generalize the non-zero adjacent constraint we used for the non-convex objective, we will need multiple layers of binary variables. Let us illustrate this by considering

[11] Though it is possible to model such constraint, it rarely makes much sense for continuous variables.

a set $x_i \in [0, u_i]$ of variables, out of which we want three adjacent to be non-zero. We introduce binary variables λ_i, δ_i and γ_i, satisfying the following

$$\begin{array}{lll}
x_0 \leq \lambda_0 u_0 & \lambda_0 \leq \delta_0 & \delta_0 \leq \gamma_0 \\
x_1 \leq \lambda_1 u_1 & \lambda_1 \leq \delta_0 + \delta_1 & \delta_1 \leq \gamma_0 + \gamma_1 \\
x_2 \leq \lambda_2 u_2 & \lambda_2 \leq \delta_1 + \delta_2 & \delta_2 \leq \gamma_1 + \gamma_2 \\
x_3 \leq \lambda_3 u_3 & \lambda_3 \leq \delta_2 + \delta_3 & \delta_3 \leq \gamma_2 + \gamma_3 \\
\dots & & \\
x_{n-1} \leq \lambda_{n-1} u_{n-1} & \lambda_{n-1} \leq \delta_{n-2} + \delta_{n-1} & \delta_{n-1} \leq \gamma_{n-2} \\
x_n \leq \lambda_n u_n & \lambda_n \leq \delta_{n-1} & \\
\sum \lambda_i = 3 & \sum \delta_i = 2 & \sum \gamma_i = 1
\end{array}$$

To see how this set of constraints works, read backward from γ to λ. Only one of the γ_i is non-zero. This allows two adjacent δ_i to be non-zero, which, in turn, allows three adjacent λ_i to be non-zero. These last binary variables then correspond to the three adjacent x_i that will be allowed to be non-zero. A nice recursive structure implemented in Code 6.12. We allow the caller some flexibility by accepting the number of variables selected to be zero or all of them; not that it makes sense in usual, but it may help to structure a loop to include all boundary cases.

Code 6.12: How to select k adjacent variables out of n to be non-zero.

```
1  def sosn(solver,k,x,rel='<='):
2   def sosnrecur(solver,k,l):
3    n = len(l)
4    d = [solver.IntVar(0,1,'') for _ in range(n-1)]
5    for i in range(n):
6     solver.Add(l[i] <= sum(d[j] \
7                         for j in range(max(0,i-1),min(n-2,i+1))))
8    solver.Add(k == sum(d[i] for i in range(n-1)))
9    return d if k <= 1 else [d,sosnrecur(solver,k-1,d)]
10  n = len(x)
11  if 0 < k < n:
12   l = k_out_of_n(solver,k,x,rel)
13   return l if k <= 1 else [l,sosnrecur(solver,k-1,l)]
```

The first layer of constraints is different since the variables might be continuous. This is handled at line 12 by calling the function creating a layer of binary variables, setting bounds on each continuous one and returning the binary array.

Then the recursive call to `sosnrecur`, an private function, at line 13 implements the successive layers, each one smaller by one variable. All the internal layers are returned to the caller.

The result of a very simple test choosing non-adjacent and adjacent integers from a randomly created array to maximize their sum is shown at Table 6.15.

Let us now return to the non-convex objective function and see how to easily solve our problem.

Code 6.13: Piecewise model for (possibly) non-convex function

Table 6.15: Choosing the largest sum of k and of k adjacent variables

Max sum of	6	10	13	12	13	9	13	10	5	10	5	15
1/12												x
Adjacent 1/12												x
2/12					x							x
Adjacent 2/12				x	x							
3/12			x		x							x
Adjacent 3/12			x	x	x							
4/12			x		x		x					x
Adjacent 4/12		x	x	x	x							
5/12			x	x	x		x					x
Adjacent 5/12			x	x	x	x	x					
6/12			x	x	x		x			x		x
Adjacent 6/12		x	x	x	x	x	x					
7/12		x	x	x	x		x	x				x
Adjacent 7/12		x	x	x	x	x	x	x				
8/12		x	x	x	x		x	x		x		x
Adjacent 8/12	x	x	x	x	x	x	x	x				
9/12		x	x	x	x	x	x	x		x		x
Adjacent 9/12		x	x	x	x	x	x	x	x	x		
10/12	x	x	x	x	x	x	x	x		x		x
Adjacent 10/12	x	x	x	x	x	x	x	x	x	x		
11/12	x	x	x	x	x	x	x	x	x	x		x
Adjacent 11/12	x	x	x	x	x	x	x	x	x	x	x	
12/12	x	x	x	x	x	x	x	x	x	x	x	x
Adjacent 12/12	x	x	x	x	x	x	x	x	x	x	x	x

```
def minimize_piecewise_linear(Points,B,convex=True):
   s,n = newSolver('Piecewise', True),len(Points)
   x = s.NumVar(Points[0][0],Points[n-1][0],'x')
   l = [s.NumVar(0,1,'l[%i]' % (i,)) for i in range(n)]
   s.Add(1 == sum(l[i] for i in range(n)))
   d = sosn(s, 2, l)
   s.Add(x == sum(l[i]*Points[i][0] for i in range(n)))
   s.Add(x >= B)
   Cost = s.Sum(l[i]*Points[i][1] for i in range(n))
   s.Minimize(Cost)
   rc = s.Solve()
   return rc,SolVal(l),SolVal(d[1])
```

Executing Code 6.13 on the same example (Table 6.13) will now produce the correct solution as can be seen in Table 6.16. Note that $\delta_1 = 1$, allowing only λ_1 and λ_2 to be non-zero. So we are now on the correct segment of the piecewise function, between point 1 and point 2 and we can correctly determine both the solution x and the optimal value $f(x)$.

Table 6.16: Correct solution to non-convex piecewise objective with $x \geq 250$

0	1	2	3	4	5	6	Solution
0.0	0.6923	0.3077	0.0	0.0	0.0	0.0	$\sum \lambda$=1.0
0	194	376	524	678	820	924	x=250.0
0	1	0	0	0	0		$\sum \delta$=1
0	3492	6404	8476	10478	12040	12664	f(x)=4388.00

6.5.3 Selecting k constraints out of n.

A related trick is to select a certain number of constraints to be satisfied (and allowing others to be violated). Let us consider the case of one constraint, say

$$\sum_i a_i x_i \leq b \tag{6.12}$$

where x is the decision variable. We could want to either raise an indicator variable if the constraint is satisfied or force the constraint if the indicator is raised.

$$\delta = 1 \Rightarrow \sum_i a_i x_i \leq b \tag{6.13}$$

$$\sum_i a_i x_i \leq b \Rightarrow \delta = 1 \tag{6.14}$$

This technique of associating a binary variable to the state of a constraint is known as *reifying* a constraint.[12]

Let us consider first the simpler Equation (6.13). We need bounds

$$u_b := \max_x \sum a_i x_i - b$$
$$l_b := \min_x \sum a_i x_i - b$$

The bounds need not be exactly computed though, as we will see, this is easily done. Any valid bound will work, with the usual caveat that, in order to avoid numerical difficulties, one should not use introduce 'large' numbers. Armed with these, we can add the constraint

$$\sum a_i x_i \leq b + u_b(1 - \delta)$$

If $\delta = 0$, then the constraint is vacuous over the domain of x. If, on the other hand, $\delta = 1$, then the constraint must hold.

The other direction, (6.14) is neither as useful, nor as simple, but is an amusing exercise in translating logical expressions into algebraic ones.

First, let formulate use the contrapositive of (6.14), namely

$$\delta = 0 \Rightarrow \sum_i a_i x_i \not\leq b \quad \text{or}$$
$$\delta = 0 \Rightarrow \sum_i a_i x_i > b$$

[12] From *res* (genitive rei), latin for object. The unusually creative nomenclature is due, not to optimizers but to computer scientists working in the related (some would say adversarial) field of Constraint Programming.

In the case where a, b and x are all integer variables, then the meaning of $\sum_i a_i x_i > b$ is clear. It means $\sum_i a_i x_i \geq b + 1$. The main difficulty occurs when x is a continuous variable. Then we need to decide the meaning of $>$ and it will be dependent on the problem we are modeling.

We need to decree that some ϵ violation of the inequality is enough. If x represent wavelengths of visible light in meters, the value of ϵ might be in the order of 10^{-9}. If x represent the money the US government spends on its military, then 10^5 might be fine. In any case, what we now want to implement is

$$\delta = 0 \Rightarrow \sum_i a_i x_i \geq b + \epsilon \tag{6.15}$$

Once the modeler has decreed ϵ, we add

$$\sum a_i x_i \geq b + l_b \delta + \epsilon(1 - \delta) \tag{6.16}$$

If $\delta = 0$, then this reduces to (6.15). If $\delta = 1$, then the lower bound comes into play and the constraint becomes vacuous. The case of 'smaller than' is handled similarly or, even more easily, by multiplying everything by -1 and using the above. The case of equality is handled by transforming it into two inequalities.

Armed with these, we can now select k constraint out of n, by creating one indicator variable δ_i per constraint and using our, previously defined `k_out_of_n` function on the δ. First, since we needs bounds and we can easily setup a linear program to find them, let us do so. Code 6.14 will find the tightest upper bound and lower bound on $\sum a_i x_i - b$ given a, x and b.

Code 6.14: How to bound a linear constraint on a box

```
1  def bounds_on_box(a,x,b):
2   Bounds,n = [None,None],len(a)
3   s = pywraplp.Solver('Box',pywraplp.Solver.GLOP_LINEAR_PROGRAMMING)
4   xx = [s.NumVar(x[i].Lb(), x[i].Ub(),'') for i in range(n)]
5   S = s.Sum([-b]+[a[i]*xx[i] for i in range(n)])
6   s.Maximize(S)
7   rc = s.Solve()
8   Bounds[1] = None if rc != 0 else ObjVal(s)
9   s.Minimize(S)
10  s.Solve()
11  Bounds[0] = None if rc != 0 else ObjVal(s)
12  return Bounds
```

The reader might wonder why, at line 4, we create essentially a copy of the provided parameter `x` instead of using `x`. The reason is that `x` is attached to the caller's solver object, while the function `bounds_on_box` is creating a new solver instance and may even be called multiple times with the same `x`, each solver likely trying to bind `x` to different values. Hence the need for different variables.

Now we are in a position to create the function to reify a constraint to a zero-one variable δ and either force it if δ is one. This is done in Code 6.15.

Code 6.15: How to reify a constraint and force it

```
def reify_force(s,a,x,b,delta=None,rel='<=',bnds=None):
```

```
2   # delta == 1 ---> a*x <= b
3   n = len(a)
4   if delta is None:
5     delta = s.IntVar(0,1,'')
6   if bnds is None:
7     bnds = bounds_on_box(a,x,b)
8   if rel in ['<=','==']:
9     s.Add(sum(a[i]*x[i] for i in range(n)) <= b+bnds[1]*(1-delta))
10  elif rel in ['>=','==']:
11    s.Add(sum(a[i]*x[i] for i in range(n)) >= b+bnds[0]*(1-delta))
12  return delta
```

The function `reify_force` accepts the required parameters defining the affine function $\sum a_j x_j - b$ in `a`, `x`, and `b` (note the sign). It also accepts a number of optional parameters. If the caller has another use for the indicator array, it may be created and passed in. If not, then it is created at line 5. In either case it is returned.

We use our `bound_on_box` function to find tight lower and and upper bounds (assuming that the domain of `x` is tight) if the user does not provide such bounds. The reader should not be alarmed at the possibly large numbers of solvers that will get executed. Each instance is very small and runs extremely fast.

Finally we add the appropriately modified constraint; either an implementation of Equation (6.16) for a 'less than or equal to' relation or the corresponding constraint for a 'greater than or equal than'. In the case the caller requires equality, we add both constraints since

$$\sum_j a_j x_j \leq b \bigwedge \sum_j a_j x_j \geq b \Rightarrow \sum_j a_j x_j = b$$

Code 6.16: How to reify a constraint and raise an indicator if it is satisfied.

```
def reify_raise(s,a,x,b,delta=None,rel='<=',bnds=None,eps=1):
  # a*x <= b ---> delta == 1
  n = len(a)
  if delta is None:
    delta = s.IntVar(0,1,'')
  if bnds is None:
    bnds = bounds_on_box(a,x,b)
  if rel == '<=':
    s.Add(sum(a[i]*x[i] for i in range(n)) >= \
          b+bnds[0]*delta+eps*(1-delta))
  if rel == '>=':
    s.Add(sum(a[i]*x[i] for i in range(n)) <= \
          b+bnds[1]*delta-eps*(1-delta))
  elif rel == '==':
    gm = [s.IntVar(0,1,'') for _ in range(2)]
    s.Add(sum(a[i]*x[i] for i in range(n)) >= \
          b+bnds[0]*gm[0]+eps*(1-gm[0]))
    s.Add(sum(a[i]*x[i] for i in range(n)) <= \
          b+bnds[1]*gm[1]-eps*(1-gm[1]))
    s.Add(gm[0] + gm[1] - 1 == delta)
  return delta
```

```

def reify(s,a,x,b,d=None,rel='<=',bs=None,eps=1):
  # d == 1 <---> a*x <= b
  return reify_raise(s,a,x,b,reify_force(s,a,x,b,d,rel,bs),rel,bs,eps)
```

The function `reify_raise` shares much structure, including the set of required and optional parameters with `reify_force`. The first difference is that, as we discussed above, the caller must supply, in the case of continuous variables, the meaning of a violation, `eps`. The default is one, which works perfectly in the case of discrete variables.

The other difference is that we cannot rely of the parameter `delta` in all cases. We can in the cases where the relation is '¡=' or '¿=', but not in the case of equality. The problem is that there are two ways in which an equality can fail: the left-hand side can be either greater or smaller than the right-hand side. This is why we introduce two other binary variables `gm[0]` (really γ_0) and `gm[1]` (really γ_1) to reflect each type of violation:

$$\sum_j a_j x_j > b + \epsilon \Rightarrow \gamma_0 = 1$$
$$\sum_j a_j x_j < b - \epsilon \Rightarrow \gamma_1 = 1$$

The γ array is then used to set δ using the following little trick. Since the γ cannot both be zero they sum to either one or two, exactly when δ needs to be zero or one, hence $\gamma_0 + \gamma_1 - 1 = \delta$.

As a final touch, we create the `reify` function that implements the *if and only if* condition that were implemented separately by `force` and `raise`.

6.5.4 Maximax and Minimin

As an application of this trickery, recall that in Section 2.3.2.1 we left unsolved the problem of modeling the *maximax*,

$$\max_x \max_i \sum_j a_{i,j} x_j + b_i$$

Subject to some constraints

(or the equivalently pernicious *minimin*).

The technique is to first transform each affine function into a constraint of the form

$$\sum_j a_{i,j} x_j + b_i = z$$

then transforming each equality into a pair of inequalities, which we reify and then apply the disjunctive trick to enforce one of them. And finally we set the objective function to $\max z$. (I did mention it was somewhat difficult to handle, but, with the routines we have developed in this section, it a matter of a few lines of code (6.17).

As an example, we will solve the following

$$\max_{x \in [2,5]} \max\{2x - 3, -2x + 12\}$$

which, graphically, is represented by Figure 6.1 where the two functions are displayed, and the maximum is in a thicker line. Note that this is decidedly a non-convex objective. Note that one could solve such a simple problem differently, but not that much more efficiently. For example one might evaluate all functions on the vertices of the polyhedron. But in order to do that, one needs to find the vertices. To find those, one needs to either solve an exponential number of linear programs or an exponential number of linear system of equations. In the worst case, our approach can theoretically require as much work, but in practice, it never does. [13]

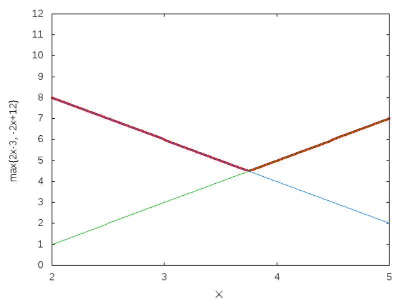


Figure 6.1: $\max\{2x-3, -2x+12\}$ over the domain [2.5]

Code 6.17: How to solve maximax problems

```
def maximax(s,a,x,b):
  n = len(a)
  d = [bounds_on_box(a[i],x,b[i]) for i in range(n)]
  zbound = [min(d[i][0] for i in range(n)), \
          max(d[i][1] for i in range(n))]
  z = s.NumVar(zbound[0],zbound[1],'')
  delta = [reify(s,a[i]+[-1],x+[z],-b[i],None,'==') for i in range(n)]
  k_out_of_n(s,1,delta)
  s.Maximize(z)
  return z,delta
```

The function `maximax` receives the solver `s` to which to add the maximax constraints, along with matrix `a` and arrays `x` and `b`, representing the n affine functions $\sum_j a_{i,j}x_j + b_i$ for $i \in [0, n-1]$. We create the additional variable `z` which will become the objective to maximize. To set meaningful bounds on `z`, we use our `bounds_on_box` to find the minima and maxima of all the affine functions on the domain of `x`. We use the minimum and maximum of those for our bounds.

Each of these functions is then set equal to `z` and reified on a corresponding `delta[i]` so that $\delta_i = 1 \Leftrightarrow \sum_j a_{i,j}x_j + b_i = z$.

Finally we force exactly one of the `delta[i]` to be one, or equivalently, one of the affine function constraints to be active. We set the objective and return both the objective and

[13] The reader who wants to understand the theory behind the previous statement about running time needs to research the branch-and-bound technique of Integer Programming to understand along with the pruning techniques used to reduce the number of sub-problems.

the array of indicators. This will provide the caller with all the necessary information at optimality.

The solution to our small example is clearly $x = 2$ with objective value $-2x + 12 = 8$. Indeed, running the Code 6.17 produces exactly that result, and also tells us that the second affine function is the active one.

Table 6.17: Solution to problem illustrated at Figure 6.1.

x	=	2.0
z	=	8.0
δ	=	[0, 1]

6.6 Staff scheduling

Staff scheduling is not one problem but a vast array of problems, each with its own set of requirements and quirks. We will discuss one interesting variation. It involves assigning instructors to course sections. The main characteristic of this problem, which makes it interesting, is the handling of instructor preferences.

Here is the generic situation: course sections have been assigned meeting times during the week. For instance MOR142[14] is worth three credit; Section 1 meets Monday, Wednesday and Friday from 9 to 10 while section 2 meets Tuesday and Thursday from 9 to 10:30. Each of these sections needs one instructor. There are dozens of these sections for various courses, each with assigned times and credits, requiring instructors.

On the other hand, we have a set of instructors, some full-time who will teach a fixed number of credits, and some part-time, who can teach up to a certain number of credits. Moreover, no instructor has managed to clone himself in order to teach two paralell sections offered at the same times. These are hard constraints.

In addition, each instructor has expressed preferences (or dislikes) for certain courses (course preferences), and days or times (We will call these preference sets). For instance we could have a set of ”Sections offered on Monday, Wednesday, Friday” and another one ”Sections meeting at night.” The instructor could have given a weighted thumbs up (or down) to this set.

If these were all the required constraints, then the model would simply be that of an assignment problem. But real scheduling is never, ever, as trivial as assignments. So let us consider one additional constraint.

This is where it gets interesting: each instructor has also indirectly expressed preferences (or disllikes) to pairs of sections. For instance there could be an abstract pair ”A section meeting on Monday night and another one meeting on Tuesday morning” or ”A section meeting, followed within an hour by another meeting”. One can easily imagine why an instructor might want to avoid (or might prefer) to have back-to-back section meetings.

To illustrate will assume an instance where all these preferences and preferences pairs have been processed and expressed, in their simplest form in Table 6.18 for the sections, Table 6.19 and Table 6.21. A good deal of pre-processing might be required to extract the data and format it in these tables, but this is not currently our concern.

In Table 6.18 the first column is an ordinal indicating the section while the second indicates the course. In our example the first two rows might correspond to the first two sections of MOR142. The third column is an indication of the time (time 12 might be Monday, Wednesday, Friday at 9).

In Table 6.19 the first column is the ordinal identification of the instructor, followed by the course load range. The third column has the preferences (positive integer) or dislikes (negative integer) of the instructor for each of the courses, in the order of Table 6.18. The fourth column holds the preferences corresponding to the sets to which sections belong in the order of Table 6.20. The last column is the preference for the pairs found in Table 6.21.

We have Table 6.20 listing the sections corresponding to each preference set.

Finally we have Table 6.20 listing the sections corresponding to each preference set.

[14] Mathematics of Operations Research, the umbrella title for the topics of this book,

Table 6.18: List of sections offered.

Id	Course id	Meeting Time
0	0	12
1	0	19
2	1	11
3	1	12
4	2	11
5	3	16
6	3	2
7	3	7
8	4	17
9	5	1
10	5	20
11	5	20
12	6	13
13	6	4
14	6	1

Table 6.19: List of preferences of each instructor

Id	Load	Course weights	Set weights	Pair weights
0	{ 2; 3 }	{ 0; 2; 0; 0; 0; 0; -4 }	{ 0; 0; 7; -5; -6; 0 }	{ 0; 0 }
1	{ 2; 2 }	{ 0; 3; 2; 0; 0; 10; 0 }	{ 0; 0; 0; 8; 4; 9 }	{ 0; 8 }
2	{ 1; 3 }	{ 2; -2; 2; 0; 8; -2; 2 }	{ 0; 0; 0; 0; 0; 9 }	{ 0; 0 }
3	{ 1; 2 }	{ 3; 0; 0; 0; 9; -2; -4 }	{ 0; 7; 9; 0; 0; 0 }	{ 0; 0 }
4	{ 2; 2 }	{ 0; -10; 1; 0; 0; 0; -6 }	{ 0; -1; 3; 10; -6; 0 }	{ 0; -7 }

Table 6.20: List of preferences sets

Id	Sections
0	{ 0; 7; 8; 9; 11; 14 }
1	{ 0; 1; 2; 7; 9; 11; 12 }
2	{ 0; 2; 5; 6; 10; 11 }
3	{ 1; 3; 6; 8; 13; 14 }
4	{ 1; 4; 5; 7; 10 }
5	{ 0; 2; 5; 7; 8; 11; 12 }

6.6.1 Constructing a model

6.6.1.1 Decision variables

What we need to decide in this problem is which instructor to assign to which section. So, clearly by now, the decision variable could be binary, indexed by the set of instructors I and the set of sections S as

$$x_{i,j} \in \{0,1\} \quad \forall i \in I; \forall j \in S$$

where $x_{13,61} = 1$ indicates that instructor id 13 is assigned to section id 61.

We will likely need a considerable number of auxiliary variables to construct a readable models. Let us start on the constraints and introduce the auxiliaries as need be.

Table 6.21: List of preferences pairs

Id	Section pairs
0	{ (3 7); (9 12); (10 14) }
1	{ (10 11); (11 13); (11 14) }

6.6.1.2 Constraints

Each section needs to be assigned at most one instructor.

$$\sum_i x_{i,j} \leq 1 \quad \forall j \in S$$

Each instructor must be assigned a number of courses within a certain range, say $[L_i, U_i]$,

$$L_i \leq \sum_j x_{i,j} \leq U_i \quad \forall i \in I$$

Now for the no-cloning constraint, that is, each instructor can be busy with at most one section per meeting time. Assume that the set of meeting times is T, then

$$\sum_{j:T_j=t} x_{i,j} \leq 1 \quad \forall t \in T$$

where T_j is the meeting time of section j.

6.6.1.3 Objective

The objective needs to maximize the preference weights of all the instructors. We will split the objective into three terms, one for the weights of instructor i on course c ($wc_{i,c}$), on preference set s ($ws_{i,s}$) and on preference pair p ($wp_{i,p}$).

For the courses, this is simple. Assuming that the set of sections S is partitioned into subsets S_c for the sections of course id c, then the contribution of the course preference weights is

$$WC = \sum_{c \in C} \sum_{i \in I} wc_{i,c} \sum_{j \in S_c} x_{i,j} \tag{6.17}$$

The contribution of the set preference weights is also fairly straightforward. We need sum over all the preference sets and all the instructors the product of the weight an instructor puts on a set with the sum over all sections of the indicator of set membership of that section and the indicator of assignment of that section to the instructor,

$$WR = \sum_{s \in R} \sum_{i \in I} ws_{i,s} \sum_{j \in R_s} x_{i,j} \tag{6.18}$$

where R_s is the last column of Table 6.20.

Now for the more interesting weights, on pairs. Let us look at a specific example. Say that pair id 4 indicates consecutive meetings and that it includes a pair of section 2 and 5. Also instructor 13 has put a weight of -15 on such consecutive pairs. Then if we assign section 2 and 5 to instructor 13, we need to add -15 to the objective value. So we need an indicator for "section 2 and 5 are assigned to instructor 13". Let us call this indicator $z_{13,4}$.

According to our model, we know that $x_{13,2}$ and $x_{13,5}$ will be one. How can we set $z_{13,4}$ if and only if both are one? By

$$
\begin{aligned}
x_{13,2} + x_{13,5} - 1 &\leq z_{13,4} \\
z_{13,4} &\leq x_{13,2} \\
z_{13,4} &\leq x_{13,5}
\end{aligned}
$$

The first inequality raises z when both x are one. The last two lower z to zero when either x is zero.

Now, in all generality, we obtain, assuming sets of pairs P_p as in the last column of Table 6.21,

$$x_{i,s_1} + x_{i,s_2} - 1 \leq z_{i,p} \quad i \in I, (s_1, s_2) \in P_p \tag{6.19}$$

$$x_{i,s_1} \geq z_{i,p} \tag{6.20}$$

$$x_{i,s_2} \geq z_{i,p} \tag{6.21}$$

An alternative approach, using what we have previously developed in the section on non-convex tricks (Section 6.5), is to use the `reify` high-level constraint which will implement

$$x_{i,s_1} + x_{i,s_2} \geq 2 \Leftrightarrow z_{i,p} = 1$$

We can now sum over all preference pairs and all instructor the product of the weight and the indicator

$$WP = \sum_p \sum_i z_{i,p} wp_{i,p} \tag{6.22}$$

We now have the complete objective function

$$\max WC + WS + WP$$

6.6.1.4 Executable model

Let us translate this into an executable model.

Code 6.18: Staff Scheduling model

```
def solve_model(S,I,R,P):
  s = newSolver('Staff Scheduling',True)
  nbS,nbI,nbSets,nbPairs,nbC = len(S),len(I),len(R),len(P),S[-1][1]+1
  nbT = 1+max(e[2] for e in S)
  x = [[s.IntVar(0,1,'') for _ in range(nbS)] for _ in range(nbI)]
  z = [[[s.IntVar(0,1,'') for _ in range(len(P[p][1]))] \
        for p in range(nbPairs)] for _ in range(nbI)]
  for j in range(nbS):
    k_out_of_n(s,1,[x[i][j] for i in range(nbI)],'<=')
  for i in range(nbI):
    s.Add(sum(x[i][j] for j in range(nbS)) >= I[i][1][0])
    s.Add(sum(x[i][j] for j in range(nbS)) <= I[i][1][1])
    for t in range(nbT):
      k_out_of_n(s,1,[x[i][j] for j in range(nbS) if S[j][2]==t],'<=')
  WC = sum(x[i][j] * I[i][2][c] for i in range(nbI) \
```

```
16         for j in range(nbS) for c in range(nbC) if S[j][1] == c)
17   WR = sum(I[i][3][r] * sum(x[i][j] for j in R[r][1]) \
18         for r in range(nbSets) for i in range(nbI))
19   for i in range(nbI):
20     for p in range(nbPairs):
21       if I[i][4][p] != 0:
22         for k in range(len(P[p][1])):
23           reify(s,[1,1],[x[i][P[p][1][k][0]],x[i][P[p][1][k][1]]],\
24              2,z[i][p][k],'>=')
25   WP = sum(z[i][p][k]*I[i][4][p] for i in range(nbI) \
26         for p in range(nbPairs) for k in range(len(P[p][1])) \
27         if I[i][4][p] != 0)
28   s.Maximize(WC+WR+WP)
29   rc,xs = s.Solve(),[]
30   for i in range(nbI):
31     xs.append([i,[[j,(I[i][2][S[j][1]],\
32          sum(I[i][3][r] for r in range(nbSets) if j in R[r][1]),
33          sum(SolVal(z[i][p][k])*I[i][4][p]/2
34          for p in range(nbPairs) for k in range(len(P[p][1]))
35           if j in P[p][1][k]))] for j in range(nbS) \
36           if SolVal(x[i][j])>0]])
37   return rc,SolVal(x),xs,ObjVal(s)
```

The function `solve_model` receives in `S` the section data in the form of Table 6.18; in `I` the instructor data in the form of Table 6.19, in `R` the preference sets data in the form of Table 6.20, and in `P` the preference pairs data in the form of Table 6.21.

The decision variable `x` on line 5 is declared as a two-dimensional array, indexed by section and instructor id. On the following line, the auxiliary variable `z` indexed by instructor id, preference pair id and the ordinal of the pairs of sections within a preference pair will be one if we assign one of the pairs to the instructor.

The loop on line 8 ensures that each section has at most one instructor. We are assuming here that there are more sections to teach than instructors to teach them. The subloop on the set of meeting times at 13 ensures that no instructor is required to be at two places at the same time.

The loop on 10 is an availability constraint; it ensures that each instructor teaches as many courses as he is supposed to teach.

At that point we have all we need to compute two of the objective terms, the weighted course preference at line 16 and the weighted set preference at line 18. These correspond to Equation (6.17) and (6.18).

To implement Equation (6.22) at line 27 we need to loop over all instructors, all sets of pairs, and all pairs (the triple loop at line 19) and reify pairs of sections assigned to an instructor. We do this if and only if the instructor used a non-zero weight on such pairs. There is no point in adding such complex constraints if the net effect on the objective function is zero.

Finally, after we solve, we massage the solution to return to the caller a meaningful answer: a list, indexed by instructor, containing all of his assigned sections with, for verification purposes, the three weights that participated in this assignment as seen in Table 6.22. The weight on preference pairs is split in two for the two sections that triggered this weight to participate in the optimal value. This is useful information for the user to lift the veil on the optimization model performance[15]

Table 6.22: Optimal solution to the Staff Scheduling

Instructor	Section (WC WR WP)		
0	[′ 2 : (2 7 0)′	′ 5 : (0 1 0)′	′10 : (0 1 0)′]
1	[′11 : (10 9 4)′	′14 : (0 8 4)′]	
2	[′ 7 : (0 9 0)′	′ 8 : (8 9 0)′	′12 : (2 9 0)′]
3	[′ 0 : (3 16 0)′	′ 1 : (3 7 0)′]	
4	[′ 6 : (0 13 0)′	′13 : (-6 10 0)′]	

6.6.2 Variations

Without modifying the overall structure of the model above, a number of variations and additional constraints are possible:

- In a typical department, not all instructors are qualified to teach all courses. A ′Qualified′ boolean could be attached to each pair Instructor-Course to prevent some assignments. This is simple to accomodate by setting to zero the corresponding decision variables for all the sections of the course.

- The department might have a policy whereby a subset of the instructors, say tenured professors, are forced to teach one lower-level course per semester, no matter what their preferences states. This would be implemented as a `k-out-of-n` type constraint.

- For certain courses with large number of sections, the department might want at least one tenured professor teaching a section, while all other sections are instructed by adjuncts. Again a `k-out-of-n` type constraint, with the appropriate set.

[15] And priceless during the inevitable discussions with staff complaining vociferously that their preferences were not met. A modeller′s work is incomplete until the user accepts the solution.

6.7 Sports timetabling

By sports timetabling, we mean here the construction of a schedule of games for a league[16]. If you do not care about spectator sports, read on anyway as the problem is interesting, very difficult, and leads us into the fascinating and complex area of *relaxation tightening* that can be applied to other complex problems.

Here is the generic problem we will try to solve: the league has a number of divisions, each with a certain number of teams. The league specifies the number of times, during a complete season, each team must face each other team of the same division and of each other division as well as the maximum number of games in a week each team will play.

For a simple instance we will use to illustrate, see Table 6.23. The "Intra" parameter is the number of times each team faces each other team of the same division. The "Inter" is for each team of other divisions. "G/W" is the number of games per week of each team and "Weeks" is the number of weeks of the season.

Table 6.23: Example of Sports Timetabling Data

(Intra Inter G/W Weeks)	{ 2; 1; 1; 19 }
Division 0 teams	{ 0; 1; 2; 3; 4; 5; 6 }
Division 1 teams	{ 7; 8; 9; 10; 11; 12; 13 }

6.7.1 Constructing a model

6.7.1.1 Decision variables

What is the end result of this model? A calendar, of sorts. Something that will display that on Week 5, for instance, Teams 1&3, Teams 2&7 etc … are facing each other. And this, for every week of the season. How can we encode this information? One possibility is for a three-dimensional binary variable $x_{i,j,w}$ where i and j are team indices ($i < j$) and w is a week index. The interpretation is, for example if $x_{2,5,13}$ is one, then teams 2 and 5 meet during week 13.

Does this strike the reader as profoundly inefficient? **It is!**

It has the redeeming value that some of the constraints will be wonderfully simple to express. If this model works, we are done. If not, then we can try harder. Let us pursue this avenue further.

6.7.1.2 Constraints

The first constraint is that we have a fixed number (say n_A) of intra-division games between teams (say T_d) of each division

$$\sum_w x_{i,j,w} = n_A \quad \forall i \in T_d; \forall j \in T_d; i < j; \forall d \in D$$

The inter-division constraint is similar. For number of games n_R,

$$\sum_w x_{i,j,w} = n_R \quad \forall i \in T_d; \forall j \in T_e; \forall d \in D; \forall e \in D; d < e$$

[16]Think NBA if you are US American, NHL if Canadian, ARL if Australian or EPL if you are the colonizer of the previous three.

The number of games per week that a team plays is actually an upper bound. Imagine a simple boundary case of one division with three teams and one game per week. One of the teams cannot possibly play. Therefore we need an inequality. For number of games n_G,

$$\sum_{i<j} x_{i,j,w} + \sum_{j<i} x_{j,i,w} \leq n_G \quad \forall i \in T; \forall w \in W$$

Notice the two sums. Since we fix the team i, we must look at the games with teams with both larger and smaller ordinals.

6.7.1.3 Objective function

This problem is complex enough that even feasibility is difficult. Besides, the possible objective functions likely vary considerably with leagues. For the sake of illustration, let us assume that we want, as much as possible, to push intra-divisional matches towards the end of the calendar. The later, the better.

Let us consider two teams of the same division, i and j. If they face each other during week w, then the variable $x_{i,j,w}$ will be one. How can we put a weight on this according to 'lateness'? We could multiply by w. This leads us to the following objective

$$\sum_{w \in W} \sum_{dinD} \sum_{i \in T_d} \sum_{j \in T_d | i<j} w x_{i,j,w}$$

Unfortunately, this objective performs rather badly sometimes. For our purposes, all solutions that have intra-divisional games towards the end are good. There is no reason to favour the last week over the second to last week. So let us do some calculations and compute the number of weeks required for intra-divisional games. For n_A games and $|T_d|$ teams in a division and a maximum of n_G games, we get that we need n_w weeks,

$$n_w = \frac{|T_d| n_A}{n_G}$$

So if we assign one for intra-divisional games in the last n_w weeks, and zero otherwise, we obtain the objective

$$\sum_{w=|W|-n_w}^{|W|} \sum_{d \in D} \sum_{i \in T_d} \sum_{j \in T_d | i<j} x_{i,j,w}$$

which performs much better, usually[17]. Notice that this computation of the required number of weeks is not always correct; it can be off by one but it serves our purposes.

6.7.1.4 Executable model

Let us translate this into an executable model. It will receive a list of divisions, each containing the teams of that division. It would be simpler if all divisions had the same number of teams, but that is never the case.

It also accepts a list of parameters, the number of intra-divisional games, `nbIntra`, of inter-divisional games, `nbInter`, of games per week for a team, `nbPerWeek` (note that this has to be a maximum, not a strict constraint) and, finally the number of weeks of the season, `nbWeeks`.

[17] For the theoretically-minded: the primal-dual gap is smaller; optimality detection is easier.

Code 6.19: Sports Timetabling model

```
1 def solve_model(Teams,(nbIntra,nbInter,nbPerWeek,nbWeeks)):
2  nbTeams = sum([1 for sub in Teams for e in sub])
3  nbDiv,Cal = len(Teams),[]
4  s = newSolver('Sports␣schedule', True)
5  x = [[[s.IntVar(0,1,'') if i<j else None for _ in range(nbWeeks)]
6       for j in range(nbTeams)] for i in range(nbTeams-1)]
7  for Div in Teams:
8   for i in Div:
9    for j in Div:
10    if i<j:
11     s.Add(sum(x[i][j][w] for w in range(nbWeeks)) == nbIntra)
12 for d in range(nbDiv-1):
13  for e in range(d+1,nbDiv):
14   for i in Teams[d]:
15    for j in Teams[e]:
16     s.Add(sum(x[i][j][w] for w in range(nbWeeks)) == nbInter)
17 for w in range(nbWeeks):
18  for i in range(nbTeams):
19   s.Add(sum(x[i][j][w] for j in range(nbTeams) if i<j) +
20         sum(x[j][i][w] for j in range(nbTeams) if j<i )<=nbPerWeek)
21 Value=[x[i][j][w] for Div in Teams for i in Div for j in Div \
22       for w in range(nbWeeks-len(Div)*nbIntra/nbPerWeek,nbWeeks) \
23       if i<j]
24 s.Maximize(sum(Value))
25 rc = s.Solve()
26 if rc == 0:
27  Cal=[[(i,j) for i in range(nbTeams-1) for j in range(i+1,nbTeams)
28        if SolVal(x[i][j][w])>0] for w in range(nbWeeks)]
29 return rc,ObjVal(s),Cal
```

Line 5 declares our decision variables. Note that this is a list of lists of lists. The first dimension is one less than the number of teams since we only will consider matches i vs j where $i < j$. The second dimension is the number of teams but note that half of the entries (below the diagonal) will never be used so we set them to `None`. The last dimension is the number of weeks.

Line 7 starts a loop to set the number of intra-division games. We loop on each division, and then on every pair (i, j) of teams in the division, respecting the $i < j$ condition to only use the upper triangle.

Similarly for the loop starting at 12 where we loop on each division, then on every division with a larger ordinal, then for every pair of teams, each in one of the two divisions.

Finally, after we solve, we massage the solution to return to the caller a meaningful result, a list of matches, indexed my the week's ordinal. For our small instance, a result is shown in Table 6.24.

This approach works for smallish instances but will not scale very well as the reader can attest by trying larger instances. (Try something the size of a professional league and be prepared to wait a while for a solution.) The problem stems from the interaction between the model feasible solution space and the techniques used by integer programming solvers to find optimal solutions. Solvers typically work by fixing some of the variables in the

Table 6.24: Optimal solution to the Sports Timetabling

Week	Matches						
0	0 vs 12	1 vs 11	2 vs 7	3 vs 13	4 vs 9	5 vs 8	6 vs 10
1	0 vs 9	1 vs 10	2 vs 13	3 vs 11	4 vs 8	5 vs 12	6 vs 7
2	0 vs 11	1 vs 12	2 vs 8	3 vs 7	4 vs 13	5 vs 10	6 vs 9
3	0 vs 13	1 vs 7	2 vs 10	3 vs 9	4 vs 12	5 vs 11	6 vs 8
4	0 vs 8	1 vs 13	2 vs 9	3 vs 12	4 vs 10	5 vs 7	6 vs 11
5	0 vs 2	1 vs 4	3 vs 5	6 vs 13	7 vs 12	8 vs 10	9 vs 11
6	0 vs 3	1 vs 4	2 vs 6	5 vs 9	7 vs 13	8 vs 11	10 vs 12
7	0 vs 4	1 vs 8	2 vs 3	5 vs 6	7 vs 13	9 vs 10	11 vs 12
8	0 vs 1	2 vs 12	3 vs 6	4 vs 5	7 vs 8	9 vs 11	10 vs 13
9	0 vs 5	1 vs 6	2 vs 4	3 vs 10	7 vs 11	8 vs 12	9 vs 13
10	0 vs 6	1 vs 3	2 vs 5	4 vs 11	7 vs 10	8 vs 13	9 vs 12
11	0 vs 1	2 vs 6	3 vs 4	5 vs 13	7 vs 11	8 vs 12	9 vs 10
12	0 vs 2	1 vs 5	3 vs 8	4 vs 6	7 vs 9	10 vs 11	12 vs 13
13	0 vs 6	1 vs 9	2 vs 3	4 vs 5	7 vs 12	8 vs 11	10 vs 13
14	0 vs 5	1 vs 6	2 vs 11	3 vs 4	7 vs 10	8 vs 13	9 vs 12
15	0 vs 4	1 vs 3	2 vs 5	6 vs 12	7 vs 9	8 vs 10	11 vs 13
16	0 vs 7	1 vs 2	3 vs 5	4 vs 6	8 vs 9	10 vs 11	12 vs 13
17	0 vs 3	1 vs 2	4 vs 7	5 vs 6	8 vs 9	10 vs 12	11 vs 13
18	0 vs 10	1 vs 5	2 vs 4	3 vs 6	7 vs 8	9 vs 13	11 vs 12

model and then solving for the others by letting them take on fractional solution; iterating multiple times. For our model, this relaxation is fairly weak. We will not go into the details of why, but we will see how to fix it, once we realize that the solvers is slow to solve. The key insight is that we can easily add more constraints.

Here is an example. Imagine an instance with one game per week and consider three teams, say i, j and k. If the decision variable is allowed to take on fractional values at some point during the execution, then, for a given week w it could happen that :

$$x_{i,j,w} = \frac{1}{2} \quad x_{i,k,w} = \frac{1}{2} \quad x_{j,k,w} = \frac{1}{2}$$

Note that this solution is allowed by the constraint that says "One game per week per team." since

$$x_{i,j,w} + x_{i,k,w} = 1$$
$$x_{i,j,w} + x_{j,k,w} = 1$$
$$x_{i,k,w} + x_{j,k,w} = 1$$

But we know that this is not a valid solution since the sum of the three variables must not exceed one. If i and j face each other, then k cannot face either of them. Similarly for the pair (i, k) and for (j, k). Therefore, knowing that there is only one game per week, we could add a constraint for every triple of team i, j, k for for every week w

$$x_{i,j,w} + x_{i,k,w} + x_{j,k,w} \leq 1$$

Note that this constraint, if the variables are integers, is redundant. But it is a valid constraint for nevertheless and it is useful for fractional values, which is happening internally in the solver. We can also consider tuples of four or even five teams. And number of games per week of two or three. See Table 6.25 for the bounds given small numbers of teams and of games per week. Note that many of these bounds will never be violated by fractional solutions so they are not very useful for our purposes.

Table 6.25: Bounds on small sums of tuples of decision variables

nb of teams	nb of games per week	bound on sum
3	1	1
	2	3
4	1	2
	2	4
	3	6
5	1	2
	2	5
	3	7
	4	9

The number of those additional constraints grows fast. The approach will therefore add a considerable number of constraints to the model. If that causes solvers to slow down unacceptably, an alternative is the approach we used to solve the TSP: adding only the constraints that we need. We would do that by solving the relaxation, looking for tuples violating the bound, and adding those. This is the aim of Code 6.20. This is an example of how easily one can add relaxation tightening constraints to a model written with or-tools.

Code 6.20: Sports timetabling with additional cuts

```
def solve_model_big(Teams,(nbIntra,nbInter,nbPerWeek,nbWeeks)):
 nbTeams,nbDiv,cuts = sum([1 for sub in Teams for e in sub]),len(Teams),[]
 for iter in range(2):
  s = newSolver('Sports schedule', False)
  x = [[[s.NumVar(0,1,'') if i<j else None for _ in range(nbWeeks)]
       for j in range(nbTeams)] for i in range(nbTeams-1)]
  basic_model(s,Teams,nbTeams,nbWeeks,nbPerWeek,nbIntra,\
         nbDiv,nbInter,cuts,x)
  rc = s.Solve()
  bounds = {(3,1):1, (4,1):2, (5,1):2, (5,3):7}
  if nbPerWeek <= 3:
   for w in range(nbWeeks):
    for i in range(nbTeams-2):
     for j in range(i+1,nbTeams-1):
      for k in range(j+1,nbTeams):
       b = bounds.get((3,nbPerWeek),1000)
       if sum([SolVal(x[p[0]][p[1]][w]) \
           for p in pairs([i,j,k],[])])>b:
        cuts.append([[i,j,k],[w,b]])
        for l in range(k+1,nbTeams):
         b = bounds.get((4,nbPerWeek),1000)
```

```
22         if sum([SolVal(x[p[0]][p[1]][w]) \
23             for p in pairs([i,j,k,l],[])])>b:
24          cuts.append([[i,j,k,l],[w,b]])
25         for m in range(l+1, nbTeams):
26          b = bounds.get((5,nbPerWeek),1000)
27          if sum([SolVal(x[p[0]][p[1]][w]) \
28              for p in pairs([i,j,k,l,m],[])])>b:
29           cuts.append([[i,j,k,l,m],[w,b]])
30   else:
31    break
32  s = newSolver('Sports␣schedule', True)
33  x = [[[s.IntVar(0,1,'') if i<j else None for _ in range(nbWeeks)]
34     for j in range(nbTeams)] for i in range(nbTeams-1)]
35  basic_model(s,Teams,nbTeams,nbWeeks,nbPerWeek,nbIntra,\
36        nbDiv,nbInter,cuts,x)
37  rc,Cal = s.Solve(),[]
38  if rc == 0:
39   Cal=[[(i,j) for i in range(nbTeams-1) for j in range(i+1,nbTeams)
40      if SolVal(x[i][j][w])>0] for w in range(nbWeeks)]
41  return rc,ObjVal(s),Cal
```

The code starts with a loop on line 3 that will run a specific number of times, solving the model with fractional solutions. Line 8 is essentially all the constraints of Code 6.19 (with the additional cuts) packaged in a procedure because we will need to use it multiple times, with fractional variables within the loop and finally with integer variables after the loop. After each solve we consider tuples of teams, see if the sum of their decision variables exceed the prescribed bound, and add their ordinal, along with the week under consideration and the bound to a list of cuts if it does.

Finally, we create an integer solver instance at line 32, add all the cuts previously found and solve for real. The routine to add the cuts is simply

Code 6.21: Cuts adding routine

```
for t,w in cuts:
 s.Add(s.Sum(x[p[0]][p[1]][w[0]] for p in pairs(t,[])) <= w[1])
```

Where the `pairs` function generates all ordered pairs from ordered tuple `t`

Code 6.22: Ordered pairs generation

```
def pairs(tuple, accum=[]):
 if len(tuple)==0:
  return accum
 else:
  accum.extend((tuple[0],e) for e in tuple[1:])
  return pairs(tuple[1:],accum)
```

Let us stress that, contrary to TSP, where the subtour elimination constraints are required for the model to be valid, the constraints we are adding here are not required; they simply are added to nudge the solver in the right direction and accelerate the solution process. Therefore, for some solvers they will help tremendously; for others they will be useless;

they might actually slow the whole process down. Without deep knowledge of the internal workings of a particular solver, the effect on runtime is nearly impossible to predict[18]. The point is that, once a modeler is aware of this technique of relaxation tightening, he may easily try it on a particular combination of model-solver that is proving recalcitrant.

6.7.2 Variations

The variations on this model are multiple. Some of them affect the objective function (or, equivalently, are handled as soft constraints); some are hard constraints; some could be either.

- There could be a list of pairs (week, team, team) with the goal of having this specified match during that specified week.
- Instead of pushing intra-divisional matches towards the end, we could be asked to follow a specific pattern say Intra-Intra-Inter.
- We could be asked to spread out (or, bunch in) the multiple matches between pairs of teams.
- Instead of 'weeks' we could be asked to schedule on specific dates.
- We could add the concept of "Home" and "Away" games with the understanding that the number of "Home" games is fixed.
- There could also be a pattern of "Home" and "Away" games to follow. This might even be considered in the context of team cities with an eye towards a "reasonable" travel schedule. (What we are describing here is the addition of *multiple* TSP layers on top of an already difficult timetabling problem! Not for the faint of heart).

[18] Actually, even with deep internal knowledge, it may be near impossible to tell. Trying the approach to see if it works is orders of magnitude easier than reading the entrails of current integer solvers. Written in C or worse C++, they have layers upon layers of complex cut generation routines with subtle interactions, not to mention cruft accumulated over years of development and debugging.

6.8 Puzzles

There is a long tradition in Constraint Programming of solving puzzles. Because it is both amusing and educational. Solving puzzles using Integer Programming is not tried as often, yet it can be as amusing and educational if sometimes more difficult. Let us not be deterred by the difficulty. The tricks used in puzzles and the mental gymnastics used to model the problems can be used for "real" problems later.

6.8.1 Pseudo-chess problems

As a warm-up, let us consider a square chessboard of some specified size n on which we wish to place as many rooks as possible so that no rook is attacking any other.[19]

The question to answer is "Where to place the rooks?". Therefore an answer must be a set of positions occupied by rooks. Since the chessboard is square, an obvious formulation of the decision variable is a two dimensional array of binary variables.

$$x_{i,j} \quad \forall i \in \{0, \ldots, n-1\}, \forall j \in \{0, \ldots, n-1\},$$

If $x_{2,5}$ is one, then there is a rook at position $2, 5$.

The objective function is simple: since we want to place as many rooks as possible, we sum our decision variables.

$$\sum_i \sum_j x_{i,j}$$

Now what would the constraints be to prevent a rook from attacking another? Rooks attack anything in the same column or row. So, we need to have at most one rook per column and per row. This is a constraint we know well: the one-out-of-n constraint.

$$\sum_i x_{i,j} \leq 1 \quad \forall j \in \{0, \ldots, n-1\}$$
$$\sum_j x_{i,j} \leq 1 \quad \forall i \in \{0, \ldots, n-1\}$$

We have all we need. Let us translate this into executable code.

Code 6.23: Maxrook model

```
def get_row(x,i):
   return [x[i][j] for j in range(len(x[0]))]
def get_column(x,i):
   return [x[j][i] for j in range(len(x[0]))]
def solve_maxrook(n):
   s = newSolver('Maxrook',True)
   x = [[s.IntVar(0,1,'') for _ in range(n)] for _ in range(n)]
   for i in range(n):
      k_out_of_n(s,1,get_row(x,i),'<=')
      k_out_of_n(s,1,get_column(x,i),'<=')
   Count = s.Sum(x[i][j] for i in range(n) for j in range(n))
   s.Maximize(Count)
   rc = s.Solve()
```

[19] A rook attacks any piece in the same column or row.

```
  xx = [[['␣','R'][int(SolVal(x[i][j]))] \
       for j in range(n)] for i in range(n)]
  return rc,xx
```

We used our `k_out_of_n` routine along with a couple of utility functions: one to extract all the row variables of a given row and one to extract all the column variables. Running Code 6.25 on a board of size 8 can produce a solution as in 6.26. It can solve boards of size 128 with the same ease. So let us consider a slightly more difficult problem, the famous N-Queens.

Table 6.26: An optimal solution to the Maxrook puzzle

	1	2	3	4	5	6	7	8
1		R						
2						R		
3								R
4							R	
5					R			
6			R					
7				R				
8	R							

This is the same problem but we are asked to place queens instead of rooks. Queens attack on the diagonals as well as on the rows and columns. All we need is some convention on naming the diagonals and a function to extract them. To make this interesting, let us generalize our `maxrook` to a `maxpiece` accepting the type of chess piece to place.

Code 6.24: Maxpiece helper functions

```
def get_se(x,i,j,n):
 return [x[i+k % n][j+k % n] for k in range(n-i-j)]
def get_ne(x,i,j,n):
 return [x[i-k % n][j+k % n] for k in range(i+1-j)]
```

Code 6.25: Maxpiece model (Queen, rook, bishop)

```
def solve_maxpiece(n,piece):
 s = newSolver('Maxpiece',True)
 x = [[s.IntVar(0,1,'') for _ in range(n)] for _ in range(n)]
 for i in range(n):
   if piece in ['R' ,'Q']:
     k_out_of_n(s,1,get_row(x,i),'<=')
     k_out_of_n(s,1,get_column(x,i),'<=')
   if piece in ['B', 'Q']:
     for j in range(n):
       if i in [0,n-1] or j in [0,n-1]:
         k_out_of_n(s,1,get_ne(x,i,j,n),'<=')
         k_out_of_n(s,1,get_se(x,i,j,n),'<=')
 Count = s.Sum(x[i][j] for i in range(n) for j in range(n))
 s.Maximize(Count)
 rc = s.Solve()
```

```
    xx = [[[' ',piece][int(SolVal(x[i][j]))] \
          for j in range(n)] for i in range(n) ]
    return rc,xx
```

We decided on naming diagonals by either SE, for southeast or NE for northeast and created two utilities to extract the corresponding variables, `get_se` and `get_ne`. We can execute this for Queens, Rooks and Bishops. A solution for each is shown in Table 6.27.

Note that there is an obvious generalization to any piece, since each occupied position on the board describes a set of positions, hence variables and the sum of those must be one.

Looking at the example solutions two obvious questions arise: Can we get all solutions? Can we get "interesting" solutions? We will leave the first question aside for now and consider the second. What would constitute an interesting solution? Maybe one that exhibits some symmetry. We could try to minimize the sum of the distances between pieces, or the maximum distance and see what happens. But enough of pseudo-chess.

Table 6.27: An optimal solution to the N-Queens and max Bishops puzzles.

	1	2	3	4	5	6	7	8
1				Q				
2								Q
3	Q							
4					Q			
5							Q	
6		Q						
7						Q		
8			Q					

	1	2	3	4	5	6	7	8
1			B	B			B	
2	B							
3								B
4								B
5	B							
6	B							
7								B
8	B	B			B	B		B

6.8.2 Sudoku

The sudoku puzzles is as follows: given a 9 by 9 grid, partially filled with numbers in the range 1 to 9, fill the rest of the grid such that:

- every row contains all numbers 1 to 9;
- every column contains all numbers 1 to 9;
- every 3 by 3 disjoint subgrid contains all numbers 1 to 9.

We can we represent a solution by specifying, for every grid position, which numbers is in there. So a simple decision variable is:

$$x_{i,j} \in \{1,\ldots,9\} \quad \forall i \in [1,2,3] \forall j \in [1,2,3]$$

The constraints are interesting. Each one of them is of the form :

- Given this set of nine positions, all numbers from 1 to 9 must appear.

In constraint programming, this is handled by a single constraint, usually named the `all_different` constraint. We will create a simplified equivalent constraint for sudoku and improve it for the next puzzle.

For every one of our variable $x_{i,j}$ we will create an array v_k^{ij} of length 9 of binary variables. Each of these is an indicator for the value k. So that we add the constraint

$$x_{i,j} = v_1^{ij} + 2v_2^{ij} + 3v_3^{ij} + \ldots + 8v_8^{ij} + 9v_9^{ij} \tag{6.23}$$

Then, for every set S of variables that need to be all different, we will ensure that the sum of the corresponding indicator variables sums to one.

This is a feasibility problem so no objective function is required. So we are ready to create an executable model. We will use our previously defined `get_row` and `get_column` to which we are adding a `get_subgrid`.

Code 6.26: Some helper functions for sudoku

```
def get_subgrid(x,i,j):
 return [x[k][l] for k in range(i*3,i*3+3) for l in range(j*3,j*3+3)]
def all_diff(s,x):
 for k in range(1,len(x[0])):
  s.Add(sum([e[k] for e in x]) <= 1)
```

Code 6.27: Sudoku model

```
1  def solve_sudoku(G):
2   s,n,x = newSolver('Sudoku',True),len(G),[]
3   for i in range(n):
4    row=[]
5    for j in range(n):
6     if G[i][j] == None:
7      v=[s.IntVar(1,n+1,'')]+[s.IntVar(0,1,'') for _ in range(n)]
8      s.Add(v[0] == sum(k*v[k] for k in range(1,n+1)))
9     else:
10     v=[G[i][j]]+[0 if k!=G[i][j] else 1 for k in range(1,n+1)]
11    row.append(v)
12   x.append(row)
13  for i in range(n):
14   all_diff(s,get_row(x,i))
15   all_diff(s,get_column(x,i))
16  for i in range(3):
17   for j in range(3):
18    all_diff(s,get_subgrid(x,i,j))
19  rc = s.Solve()
20  return rc,[[SolVal(x[i][j][0]) for j in range(n)] for i in range(n)]
```

The routine accepts a grid of data with either a number or `None` to indicate that it must be filled.

Most of the work is to create a clean set of decision variables in the loop from line 3 to line 12: a three dimensional array indexed by the position on the grid for the first two dimension. At index 0 in the third dimension is our real decision variable, holding the value that the grid will hold (either because it is data or after the solution process); each other index from 1 to 9 holds the corresponding indicator variable. As we create the variables we add the value constraint of Equation (6.23) at line 8.

After the variable declaration we call the `all_diff` function of each row, column and subgrid. This function is a simple `k_out_of_n` for each value in the range 1 to 9.

Finally we return the grid values (not the eight hundred or so indicator variables!) As an example see Table 6.28. The data are in bold.[20]

Table 6.28: Solution to a sudoku puzzle

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

6.8.3 Send more money!

Here is a crypt-arithmetic puzzle, famous in the Constraint Programming community: Replace each of the letters 'S', 'E', 'N', 'D', 'M', 'O', 'R', 'Y', with a distinct digit from 0 to 9 such that the following sum is correct

```
SEND + MORE = MONEY
```

There are two high-level constraints in this puzzle; the first is arithmetic: we need the equation to hold. We can do this by decomposing each integer into its place-value. `SEND` is a four-digit number (presumably in base ten) so it really is:

```
S*1000 + E*100 + N*10 + D*1.
```

Each of `MORE` and `MONEY` is handle the same way. Then we constrain the equation to hold.

The second constraint is the requirement that all letter receive a distinct digit. Here is another case where we could profitably use an `all-different` constraint, so let us generalize what we did for the pseudo-chess models so that we can invoke `all-different` in any model. Our trick relies on each variable having an array of associated indicator variables, one for each potential integer value. So, in addition to our previously defined constraint, which we will slightly generalize, we need a routine for variable creation. This is the intent of `newIntVar` of Code 6.28.

Code 6.28: all-different structure and constraint

```
def newIntVar(s, lb, ub):
  l = ub-lb+1
  x = [s.IntVar(lb, ub, '')]+[s.IntVar(0,1,'') for _ in range(l)]
  s.Add(1 == sum( x[k] for k in range(1,l+1)))
  s.Add(x[0] == sum((lb+k-1)*x[k] for k in range(1,l+1)))
  return x
```

[20] I have run this code on over twenty thousand puzzles. In most cases, the model runs in a small fraction of a second; occasionally it will take a few seconds.

```
def all_different(s,x):
  lb,ub=min(int(e[0].Lb()) for e in x),max(int(e[0].Ub()) for e in x)
  for v in range(lb,ub+1):
    all = []
    for e in x:
      if e[0].Lb() <= v <= e[0].Ub():
        all.append(e[1 + v - int(e[0].Lb())])
    s.Add(sum(all) <= 1)
def neq(s,x,value):
  s.Add(x[1+value-int(x[0].Lb())] == 0)
```

We notice that, although unstated, there is an additional assumption on 'S' and 'M': they cannot take on value 0 if the numbers are truly four and five digits long. So we should constrain them to be non-zero. Interestingly, the data structure we have chosen for the `all_different` implementation allows us to trivially create a disequality as we see in function `neq` of Code 6.28.

Armed with this, we can now solve the puzzle. Its solution is shown at 6.29.

Code 6.29: Send More Money

```
def solve_smm():
  s = newSolver('Send␣more␣money',True)
  ALL = [S,E,N,D,M,O,R,Y] = [newIntVar(s,0,9) for k in range(8)]
  s.Add( 1000*S[0]+100*E[0]+10*N[0]+D[0]
       + 1000*M[0]+100*O[0]+10*R[0]+E[0]
       == 10000*M[0]+1000*O[0]+100*N[0]+10*E[0]+Y[0])
  all_different(s,ALL)
  neq(s,S,0)
  neq(s,M,0)
  rc = s.Solve()
  return rc,SolVal([a[0] for a in ALL])
```

The solution is shown at Table 6.29. The reader can verify that the equation holds ($9567 + 1085 = 10652$).

Table 6.29: Solution to the Send More Money puzzle

S	E	N	D	M	O	R	Y
9	5	6	7	1	0	8	2

6.8.4 Ladies and tigers

Raymond Smullyan in "The Lady or the Tiger" [21] presents a number of logic puzzles. One chapter culminates in the following:

A prisoner is faced with nine doors, one of which he must open. Behind one door awaits a lady; behind the others, a tiger, if anything. One assumes that the prisoner prefers opening the lady's door to an empty room which is preferable to a tiger's den. What turns this into a logic puzzle is that on each door is posted a logic statement (it can therefore be

[21] Dover recreational math; ISBN-13: 978-0486470276

either true or false). Statements on rooms with tigers are false. The statement on the Lady's room is true.

- Door 1: The lady is in an odd-numbered room
- Door 2: This room is empty.
- Door 3: Either sign 5 is right or sign 7 is wrong.
- Door 4: Sign 1 is wrong.
- Door 5: Either sign 2 or sign 4 is right.
- Door 6: Sign 3 is wrong.
- Door 7: The lady is not in room 1.
- Door 8: This room contains a tiger and room 9 is empty.
- Door 9: This room contains a tiger and sign 6 is wrong.
  Where is the lady?

To know where the lady awaits, we may need to know where are the tigers. So a reasonable decision variable, given a set $R = \{1, \ldots, 9\}$ of rooms and a set $B = \{1, 2, 3\}$ of beasts (say 1 for empty, 2 for Lady and 3 for tiger):

$$r_i \in B \quad \forall i \in R$$

So that a Lady in room 5 would be indicated by $r_5 = 2$ and a tiger in room 4 would be $r_4 = 3$. A statement about the lady being in an odd numbered room is easy to accommodate if we declare our variables using our `newIntVar` function. The associated array of indicator variables the perfect tool. For the sake of presentation, let us assume that for each r_i variable we have an array $r_{i,j}$ of indicator variables for $j \in B$.

Now for the logic part. We are given statements which can be true or false and their truth value influences the constraints. If we introduce a binary variable for each statement, then we can use our reify logic to associate the variable to each constraint. So let us introduce

$$s_i \in \{0, 1\} \quad \forall i \in R$$

So that $s_2 = 1$ will mean that the statement on door 2 is true.

The executable model is Code 6.30. We will decompose it one constraint at a time.

Code 6.30: Lady or tiger model

```
def solve_lady_or_tiger():
  s = newSolver('Lady or tiger', True)
  Rooms = range(1,10)
  R = [None]+[newIntVar(s,0,2) for _ in Rooms]
  S = [None]+[s.IntVar(0,1,'') for _ in Rooms]
  i_empty,i_lady,i_tiger = 1,2,3
  k_out_of_n(s,1,[R[i][i_lady] for i in Rooms])
  for i in Rooms:
    reify_force(s,[1],[R[i][i_tiger]],0,S[i],'<=')
    reify_raise(s,[1],[R[i][i_lady]],1,S[i],'>=')
```

```
11 reify(s,[1]*5,[R[i][i_lady] for i in range(1,10,2)],1,S[1],'>=')
12 reify(s,[1],[R[2][i_empty]],1,S[2],'>=')
13 reify(s,[1,-1],[S[5],S[7]],0,S[3],'>=')
14 reify(s,[1],[S[1]],0,S[4],'<=')
15 reify(s,[1,1],[S[2],S[4]],1,S[5],'>=')
16 reify(s,[1],[S[3]],0,S[6],'<=')
17 reify(s,[1],[R[1][i_lady]],0,S[7],'<=')
18 reify(s,[1,1],[R[8][i_tiger],R[9][i_empty]],2,S[8],'>=')
19 reify(s,[1,-1],[R[9][i_tiger],S[6]],1,S[9],'>=')
20 rc = s.Solve()
21 return rc,[SolVal(S[i]) for i in Rooms],\
22  [SolVal(R[i]) for i in Rooms]
```

At line 3 we define the range of integers that identify each door. Since the problem is stated with rooms numbered from one, we will comply instead of renumbering from zero as we usually do. In order to index from one, we create the decision variables on the next two lines as arrays where the first element contains `None`. We then define, at line 6, some constants to access the indicator variables of each room.

On line 7 we ensure that there is exactly one lady.

All the other constraints involve a relation between a statement variable `S` and a logical statement, hence our `reify` functions will prove invaluable. The first one is that if a room contains a tiger, its statement is false. The statement "If room i contains a tiger then statement i is false" is in the wrong direction for us. We could write a new function but is is simpler to use the contra-positive and state "If statement i is true, there is no tiger behind door i". This is an instance of a Boolean true enforcing a constraint, which line 9 implements.

The next constraint "A room with a lady has a true statement on its door" is in the right direction for us: a satisfied constraint raising a Boolean, as implemented at line 10.

"The lady is in an odd-numbered room." is simple. We need to sum the `i_lady` indicator variables on odd doors and set them above one if and only if statement one is true. The indices of odd rooms is obtained by `range(1,10,2)` and the inequality

```
R[1][i_lady]+R[3][i_lady]+R[5][i_lady]+R[7][i_lady]+R[9][i_lady] >= 1
```

is reified to `S[1]` at line 11.

"This room is empty." is a simple reification to `S[2]` of

```
R[2][i_empty] >= 1
```

at line 12. The reader might wonder why not use equality instead of inequality. The reason is that we know, having written the constraint `reify`, that equalities are more complex, introducing more auxiliary constraints and or variables. If we are certain, as we are here, that an inequality is sufficient, it is preferable to use one.

"Either sign 5 is right or sign 7 is wrong." is a disjunction of binary variables with the minor difficulty that the second is negated. The transformation to an algebraic statement is mechanical if we know how to negate Booleans and implement disjunctions. We have seen the disjunctions often. Something like $x_1 \vee \ldots x_n$ gets implemented as $\sum_i x_i \geq 1$. The negation of x_i is handled by replacing x_i by $1 - x_i$. Therefore in our case we need

```
S[5] + (1-S[7]) >= 1
```

which simplifies to

```
S[5] - S[7] >= 0
```

reified to `S[3]` at line 13.

”Sign 1 is wrong.” is `S[1]==0` reified to `S[4]` at line 14. ”Sign 3 is wrong.” is handled identically at line 16.

”Either sign 2 or sign 4 is right.” is a simple disjunction so the usual transformation to addition applies and is reified to `S[5]` at line 15.

”The lady is not in room 1.” needs to reify to `S[7]` the constraint

```
R[1][i_lady] <= 0
```

as at line 17.

”This room contains a tiger and room 9 is empty.” is interesting. The sub-statements are

```
R[8][i_tiger] >= 1
```

and

```
R[9][i_empty] >= 1
```

The conjunction is handled by summing the right sides and the left sides to create

```
R[8][i_tiger] + R[9][i_empty] >= 2
```

reified to `S[8]` at line 18.

Finally ”This room contains a tiger and sign 6 is wrong.” contains

```
R[9][i_tiger] >= 1
```

and

```
S[6] <= 0
```

We transform the latter into

```
- S[6] >= 0
```

then sum the two form the conjunction, reified to `S[9]` at line 19.

And we are done in the sense that we can find a solution, for instance the first solution of Table 6.30. But are there additional solutions and if so, how to find them? In this case[22], it is simple since our only real goal is to find the lady. Our first solution has the lady in room 1, so we can simply add a constraint preventing the lady to be in room 1, for instance

```
s.Add(R[1][i_lady] == 0)
```

We will get another solution of one exists or else the solver will indicate that the problem is infeasible. We can proceed thus until we exhaust all solutions if need be. The second solution of Table 6.30 is one such additional solution. (Interestingly this second solution is the unique solution if we add a constraint stating ”Room 8 is not empty”.)

[22]In general, for integer programs, it is very difficult to find all solutions; but it is possible for many practical cases.

Table 6.30: Two solutions to the Lady or Tiger puzzle.

1	The lady is in an odd-numbered room.	T	Lady	T	
2	This room is empty.	T		F	Tiger
3	Either sign 5 is right or sign 7 is wrong.	T		F	
4	Sign 1 is wrong.	F		F	
5	Either sign 2 or sign 4 is right.	T		F	
6	Sign 3 is wrong.	F		T	
7	The lady is not in room 1.	F		T	Lady
8	This room contains a tiger and room 9 is empty.	F		F	Tiger
9	This room contains a tiger and sign 6 is wrong.	F		F	Tiger

Quick reference for OR-Tools MPSolver in Python

This is by no means a complete reference, but it is enough for all the models described in the book. Also described are the wrapper functions used to simplify model writing.

To use the LP and IP solvers a model must start with:

Code 6.31: Library declaration.

```
from linear_solver import pywraplp
```

A solver instance is created by:

Code 6.32: Creation of a solver instance in or-tools..

```
s = pywraplp.Solver(NAME,pywraplp.Solver.TYPE)
```

where `s` is the returned solver, `NAME` is any string and `TYPE` is one of

`GLOP_LINEAR_PROGRAMMING`	LP
`CLP_LINEAR_PROGRAMMING`	LP
`GLPK_LINEAR_PROGRAMMING`	LP
`SULUM_LINEAR_PROGRAMMING`	LP
`GUROBI_LINEAR_PROGRAMMING`	LP
`CPLEX_LINEAR_PROGRAMMING`	LP
`SCIP_MIXED_INTEGER_PROGRAMMING`	MIP
`GLPK_MIXED_INTEGER_PROGRAMMING`	MIP
`CBC_MIXED_INTEGER_PROGRAMMING`	MIP
`SULUM_MIXED_INTEGER_PROGRAMMING`	MIP
`GUROBI_MIXED_INTEGER_PROGRAMMING`	MIP
`CPLEX_MIXED_INTEGER_PROGRAMMING`	MIP
`BOP_INTEGER_PROGRAMMING`	IP (Binary)

All the models in the book use `GLOP` for LP problems and `CBC` for all MIPs. This is how the current wrapper is set:

Code 6.33: Creation of a solver instance via wrapper.

```
s = newSolver(NAME,[False|True])
```

where `False` is the default and returns an LP solver instance (`GLOP`) and `True` returns a MIP solver (`CBC`).

To a solver instance we add decision variables by:

Code 6.34: Continuous decision variable declaration via OR_Tools.

```
var = s.NumVar(LOW,HIGH,NAME)
```

for continuous variables or

Code 6.35: Integer decision variable declaration via OR_Tools.

```
VAR = s.IntVar(LOW,HIGH,NAME)
```

for integer variables where VAR is the returned variable object, NAME is any string. Names must be unique within a solver instance. Given the empty string, an automatic unique name Will be internally generated; a precious feature, especially for routines that will be repeatedly reused within a solver instance. The range is described by LOW, any number or -solver.infinity() and by HIGH, any number larger than LOW or solver.infinity(). It is a good rule of thumb to restrict the range as much as possible.

The easiest way to create an array of decision variables is

Code 6.36: Decision variable array declaration example.

```
x = [s.NumVar(LOW,HIGH,'') for _ in range(N)]
```

where N is the number of elements required in the array. Arrays are, of course, indexed starting at zero. Similarly for high-dimensional arrays. For example, an M by N matrix is created by

Code 6.37: Decision variable 2-D array declaration example.

```
m = [[s.NumVar(LOW,HIGH,'') for _ in range(N)] for _ in range(M)]
```

After variable declarations, constraints follow. The simplest constraint declaration is

Code 6.38: Generic constraint declaration via or-tools.

```
s.Add(REL)
```

where REL is (almost) any linear algebraic relation using decision variables, numbers, the arithmetic operators +,-,*,/ and the equality and inequality relationals. For example

Code 6.39: Simple algebraic constraints in or-tools.

```
s.Add(2 * x[12] + 30 * x[13] <= 100)
s.Add(25 == x[100] - x[101])
s.Add(x[99] >= x[100])
```

Never use the strict inequalities. For continuous variables, they make no sense and for integer variables, they can trivially be changed to the non-strict inequalities by adding one. Also remember that we can never use products of decision variables.

A useful helper function is the sum, invoked as

Code 6.40: Sum operator in or-tools.

```
s.Sum(LIST)
```

where LIST is any list (or tuple) of decision variables. This can be used, for instance,

Code 6.41: Examples of sum in or-tools.

```
s.Add(s.Sum(x) <= 100)
s.Add(s.Sum(m[i][j] for i in range(M) for j in range(N))<= 100)
```

In addition to the constraints, a model usually has an objective function of either of the forms

Code 6.42: Objective function declaration in or-tools.

```
s.Maximize(EXPR)
s.Minimize(EXPR)
```

where EXPR is any linear algebraic expression in the decision variables.

To invoke the solver on the model created:

Code 6.43: Solver invocation in or-tools.

```
rc = s.Solve()
```

where rc is the returned value and is zero if all goes well. It can be one of

OPTIMAL
FEASIBLE
INFEASIBLE
UNBOUNDED
ABNORMAL
NOT_SOLVED

So, to be pedantic, one should check the return value against those defined constants but the programmers at Google have followed the decades-old tradition of returning zero when all goes well[23].

After a solve, one accesses the optimal value and optimal solution via

Code 6.44: Optimal value and optimal solution in or-tools.

```
value = s.Objective().Value()
varval = var.SolutionValue()
```

These are wrapped in helper functions

Code 6.45: Wrapper functions for optimal value and solutions.

```
value = ObjVal(s)
xval = SolVal(x)
```

The returned variable xval will have the same dimensions as the parameter x.

In addition, the wrapper library provides the following higher-level constraints

Code 6.46: High-level constraints

```
l = k_out_of_n(s,k,x,rel='==')
l = sosn(solver,k,x,rel='<=')
delta = reify_force(s,a,x,b,delta=None,rel='<=',bnds=None)
delta = reify_raise(s,a,x,b,delta=None,rel='<=',bnds=None,eps=1)
delta = reify(s,a,x,b,d=None,rel='<=',bnds=None,eps=1)
```

where :

- k_out_of_n adds the necessary constraints to solver s so that exactly, at least or at most (depending on rel) k (a positive integer) variables from the list x are allowed to be non-zero. It returns l an array of binary variables of the same length as x.
- sosn adds the necessary constraints to solver s so that exactly, at least or at most (depending on rel) k adjacent variables from the list x are allowed to be non-zero. It returns l an array of binary variables of the same length as x.

[23] I believe the tradition started with Dennis Ritchie and Unix as simplifying medication against the Multics headache inducing complexities.

- `reify_force` adds the necessary constraints to solver `s` so that the relation $\sum_i a_i x_i \approx b$ is forced (for relation $\approx$ determined by `rel`) when `d` (a binary integer variable) is one. This last variable need not be declared ahead of the call to `reify_force`. It is returned whether created internally or not.
- `reify_raise` implements the opposite implication to `force`.
- `reify` calls both `force` and `raise` to implement an "if and only if" condition.

It also provides the helper function

Code 6.47: High-level helper optimization function.

```
bounds_on_box(a,x,b)
```

Where `bounds_on_box` finds the smallest and largest possible value of $\sum a_i x_i - b$ on the domain of variable `x`.