Mathematical Optimization Documentation

Release 1

João Pedro Pedroso and Abdur Rais and Mikio Kubo and Masaka

Jan 29, 2019

Contents

This book is an introduction to optimization based on PyScipOpt — a Python interface to the SCIP optimization software.

Please visit the documentation's source at https://github.com/joaopedroso/scipbook for corrections and suggestions.

Readable documentation can be found at http://scipbook.readthedocs.io.

Copyright (C) João Pedro Pedroso, Abdur Rais, Mikio Kubo and Masakazu Muramatsu

Contents:

Forward

Forward

This book is loosely based on *"Mathematical Optimization: Solving Problems using Python and Gurobi"* by M. Kubo, J.P. Pedroso, M. Muramatsu, and A. Rais, in Japanese, published in 2012 by Kindaikagakusha in Tokyo. Readers fluent in Japanese and aiming at using Gurobi as a solver are kindly directed to that book. Our interests in preparing this version in English are twofold: we wish to widen the readership, and we would like to give the possibility of using a cutting edge solver to potential users who prefer the licensing policy of SCIP.

This book is an open project: we expect that new editions will incorporate contributions by our readers and SCIP users, and to extend it to exploit SCIP's potential in some specific areas, in particular nonlinear optimization.

Formulations and programs proposed in the book have been extensively tested computationally; the results are available in XXXXXX. % !!!!

Forward to the Japanese edition

Mathematical optimization (previously known as *mathematical programming*), is a branch of applied mathematics with more than half a century history. Being an area where the theory and abundant and elegant applications, it has been called the queen of applied mathematics.

In this book, rather than presenting an old-fashioned, theoretical introduction the mathematical optimization, our intention is to provide the basis for mastering the technique of solving the problem at hand by means of an "optimization solver", unveiling tricks and tips to model and solve real-world problems.

Solving mathematical optimization problems involves extensive numerical calculations. It required acquaintance with computers and proficiency in specialized programming languages, besides familiarity with mathematical modeling and optimization algorithms. The hurdle was very high, and it was extremely rare for companies to have human resources making full use of the power of mathematical optimization for solving their real problems.

Recently, however, mathematical optimization problems became easily solvable by means of general-purpose, high performance solvers. Besides, currently available very high-level programming languages significantly reduced the barrier for actually using these solvers. Therefore, it became possible to quickly tackle even very complex real-world problems.

In order to respond to such changes in paradigm, it was the authors intention to write a new type of introduction to mathematical optimization. As much as possible, the theoretical descriptions have been limited to subjects that are useful in practice. There is no hindrance to the usage of a mathematical optimization solver as black box, because there is no need to know all of its details for successfully using it. Nevertheless, as most of the practical problems involving combinatorial optimization belong to the class of to the so-called NP-hard problems, it can not be avoided that their solution can be very time consuming. Therefore, for successfully tackling these problems it is necessary a basic understanding of the theory, along with some modeling tricks. In this book, as well as commented examples of using the basic theory, we provide the readers with indications on how correctly and quickly solve practical problems. We hope this book will be a primer for the usage of mathematical optimization in a new era.

The information contained in this document is as follows. Chapter 1 is an introduction to the basics of mathematical optimization. First of all, it presents the terminology and the most fundamental class of mathematical optimization problems, the linear optimization problem. Then, it explains with examples how to formulate simple models and how to use a mathematical optimization solver to find a solution.

In the section presenting the transportation problem it will be explained the concept of duality. Besides its practical applications, duality allows a better understanding the theory underlying linear optimization. In the section of the multi-constraint knapsack problem we explain the basic solving technique for problems involving integer variables, the branch and bound method. In the section concerning the nutrition problem we discuss the case where there is no optimal solution (infeasible or unbounded instances), and propose workarounds.

In Chapter 2, we describe some precautions that should be taken when formulating integer optimization problems. Illustrations include the capacity constrained facility location problem, the k-median problem, and a commented example of the k-center problem.

In the third chapter we introduce a formulation for the bin packing problem. We present a formulation for the variant called the cutting stock problem, and introduce a solution technique that utilizes duality, in the so-called column generation approach.

In the fourth chapter, we introduce combinatorial optimization problems related to graphs: the graph partitioning problem, the maximum stable set problem, and the graph coloring problem. In the section on the graph coloring problem, we describe an ingenuous formulation to deal with symmetry.

Chapter 5 describes routing problems. After dealing with the basic traveling salesman problem, we propose a formulation for this problem with time windows, and some formulations for the capacity constrained vehicle routing problem. In addition, sections of the traveling salesman problem introduce the cutting plane method.

Chapter 6 focuses on scheduling problems. Several types of formulation are proposed; the one to select depends on particular case at issue.

In Chapter 7 the dynamic lot sizing problem is analyzed with formulations for the multiple item case, and for the multi-stage lot sizing problem.

Chapter 8 describes techniques to approximate a nonlinear function with a piecewise linear function, explaining the concept of special ordered set.

Chapter 9 deals with multi-objective optimization, describing the basic theory and the usage of SCIP/Python for solving this class of problems

> Nothing in the world takes place without optimization, and there is no doubt that all aspects of the world that have a rational basis can be explained by optimization methods. Leonhard Euler, 1744 (translation found in "Optimization Stories", edited by Martin Grötschel).

Introduction

This introductory chapter is a run-up to Chapter 2 onwards. It is an overview of mathematical optimization through some simple examples, presenting also the main characteristics of the solver used in this book: SCIP (http://scip.zib.de).

The rest of this chapter is organized as follows. Section *Mathematical Optimization* introduces the basics of mathematical optimization and illustrates main ideas via a simple example. Section *Linear Optimization* presents a real-world production problem to discuss concepts and definitions of linear-optimization model, showing details of SCIP/Python code for solving a production problem. Section *Integer Optimization* introduces an integer optimization model by adding integer conditions to variables, taking as an example a simple puzzle sometimes used in junior high school examinations. A simple transportation problem, which is a special form of the linear optimization problem, along with its solution is discussed in Section *Transportation Problem*. Here we show how to model an optimization problem as a function, using SCIP/Python. Section *Duality* explains duality, an important theoretical background of linear optimization, by taking a transportation problem as an example. Section *Multi-product Transportation Problem* presents a multi-commodity transportation problem, which is an generalization of the transportation, and describes how to handle sparse data with SCIP/Python. Section *Blending problem* introduces mixture problems as an application example of linear optimization. Section *Fraction optimization problem* presents the fraction optimization problem, showing two ways to reduce it to a linear problem. Section *Multi-Constrained Knapsack Problem* illustrates a knapsack problem with details of its solution procedure, including an explanation on how to debug a formulation. Section *The Modern Diet Problem* considers how to cope with nutritional problems, showing an example of an optimization problem with no solution.

2.1 Mathematical Optimization

Let us start by describing what mathematical optimization is: it is the science of finding the "best" solution based on a given objective function, i.e., finding a solution which is at least as good and any other possible solution. In order to do this, we must start by describing the actual problem in terms of mathematical formulas; then, we will need a methodology to obtain an optimal solution from these formulas. Usually, these formulas consist of constraints, describing conditions that must be satisfied, and by an objective function.

In other words, a mathematical optimization problem is usually expressed as:

- *objective function* (which we want to maximize of minimize);

- *conditions of the problem:* constraint 1, constraint 2, ...

For the solution obtained to be meaningful, this model must capture the objective of optimization accurately, along with all esse

1. a set of *variables*: the unknowns that need to be found as a solution to the problem;

2. a set of *constraints*: equations or inequalities that represent requirements in the problem as relationships between the variables

3. an *objective function*: an expression, in terms of the defined variables, which determines e.g. the total cost, or the profit of the targeted problem.

The problem is a minimization when smaller values of the objective are preferrable, as with costs; it is a maximization when larger values are better, as with profits. The essence of the problem is the same, whether it is a minimization or a maximization (one can be converted into the other simply by putting a minus sign in the objective function).

In this text, the problem is described by the following format.

- **Maximize** or **minimize**
  - Objective function
- **Subject to:**
  - Constraint 1
  - Constraint 2
  - ...

The optimization problem seeks a solution to either *minimize* or *maximize* the objective function, while satisfying all the constraints. Such a desirable solution is called *optimum* or *optimal solution* — the best possible from all candidate solutions measured by the value of the objective function. The variables in the model are typically defined to be non-negative real numbers.

There are many kinds of mathematical optimization problems; the most basic and simple is *linear optimization*[1]. In a linear optimization problem, the objective function and the constraints are all linear expressions (which are straight lines, when represented graphically). If our variables are $x_1, x_2, \ldots, x_n$, a linear expression has the form $a_1x_1 + a_2x_2 + \ldots + ax_n$, where $a_1, \ldots, a_n$ are constants.

For example,

$$\text{minimize}$$
$$3x + 4y$$
$$\text{subject to:}$$
$$5x + 6y \geq 10$$

$$7x + 5y \geq 5$$

$$x, y \geq 0$$

is a linear optimization problem.

One of the important features of linear optimization problems is that they are easy to solve. Common texts on mathematical optimization describe in lengthy detail how a linear optimization problem can be solved. Taking the extreme case, for most practitioners, how to solve a linear optimization problem is not important. For details on how methods for solving these problems have emerged, see *Margin seminar 1*. Most of the software packages for mathematical

---

[1] As said before, until recently these were called *linear programming* problems, which had been abbreviated as *LP*; complying to the new nomenclature, the abbreviation we will use is *LO*, for *linear optimization* problems.

optimization support linear optimization. Given a description of the problem, an optimum solution (i.e., a solution that is guaranteed to be the best answer) to most of the practical problems can be obtained in an extremely short time.

Unfortunately, not all the problems that we find in the real world can be described as a linear optimization problem. Simple linear expressions are not enough to accurately represent many complex conditions that occur in practice. In general, optimization problems that do not fit in the linear optimization paradigm are called *nonlinear optimization* problems.

In practice, nonlinear optimization problems are often difficult to solve in a reliable manner. Using the mathematical optimization solver covered in this document, SCIP, it is possible to efficiently handle some nonlinear functions; in particular, quadratic optimization (involving functions which are a polynomial of up to two, such as $x^2 + xy$) is well supported, especially if they are convex.

A different complication arises when some of the variables must take on integer values; in this situation, even if the expressions in the model are linear, the general case belongs to a class of difficult problems (technically, the NP-hard class[2]). Such problems are called integer optimization problems; with ingenuity, it is possible to model a variety of practical situations under this paradigm. The case where some of the variables are restricted to integer values, and other are continuous, is called a *mixed-integer* optimization problem. Even for solvers that do not support nonlinear optimization, some techniques allow us to use mixed-integer optimization to approximate arbitrary nonlinear functions; these techniques (piecewise linear approximation) are described in detail in Chapter *Piecewise linear approximation of nonlinear functions*.

---

[2] A class of problems which, even though no one proved it, are believed to be difficult to solve; i.e., solving these problems requires resources that grow exponentially with the size of the input.

2.2 Linear Optimization

We begin with a simple linear optimization problem; the goal is to explain the terminology commonly used optimization.

$$\text{maximize}$$
$$15x_1 +$$
$$18x_2 +$$
$$30x_3$$
$$\text{subject to:}$$
$$2x_1 +$$
$$x_2 +$$
$$x_3 \leq$$
$$60$$

$$x_1 +$$
$$2x_2 +$$
$$x_3 \leq$$
$$60$$

$$x_3 \leq$$
$$30$$

$$x_1,$$
$$x_2,$$
$$x_3 \geq$$
$$0$$

Let us start by explaining the meaning of x_1, x_2, x_3: these are values that we do not know, and which can change continuously; hence, they are called *variables*.

The first expression defines the function to be maximized, which is called the *objective function*.

The second and subsequent expressions restrict the value of the variables $x1, x2, x3$, and are commonly referred to as *constraints*. Expressions ensuring that the variables are non-negative $(x1, x2, x3 \geq 0)$ have the specific name of *sign restrictions* or *non-negativity constraints*. As these variables can take any non-negative real number, they are called *real variables*, or *continuous variables*.

In this problem, both the objective function and the constraint expressions consist of adding and subtracting the variables x_1, x_2, x_3 multiplied by a constant. These are called *linear expressions*. The problem of maximizing (or minimizing) a linear objective function subject to linear constraints is called a *linear optimization problem*.

The set of values for variables x_1, x_2, x_3 is called a *solution*, and if it satisfies all constraints it is called a *feasible solution*. Among feasible solutions, those that maximize (or minimize) the objective function are called *optimal solutions*. The maximum (or minimum) value of the objective function is called the *optimum*. In general, there are multiple solutions with an optimum objective value, but usually the aim is to find just one of them.

Finding such point can be explored in some methodical way; this is what a linear optimization solver does for finding the optimum. Without delay, we are going to see how to solve this example using the SCIP solver. SCIP has

been developed at the Zuse Institute Berlin (ZIB), an interdisciplinary research institute for applied mathematics and computing. SCIP solver can be called from several programming languages; for this book we have chosen the very high-level language *Python*. For more information about SCIP and Python, see appendices SCIPintro and PYTHON-intro, respectively.

The first thing to do is to read definitions contained in the SCIP module (a *module* is a different file containing programs written in Python). The SCIP module is called `pyscipopt`, and functionality defined there can be accessed with:

```python
from pyscipopt import Model
```

The instruction for using a module is `import`. In this statement we are importing the definitions of `Model`. We could also have used `from pyscipopt import *`, where the asterisk means to import all the definitions available in `pyscipopt. ..` ; we have imported just some of them, and we could have used other idioms, as we will see later. One of the features of Python is that, if the appropriate module is loaded, a program can do virtually anything[3].

The next operation is to create an optimization model; this can be done with the `Model` class, which we have imported from the `pyscipopt` module.

```python
model = Model("Simple linear optimization")
```

With this instruction, we create an object named `model`, belonging the class `Model` (more precisely, `model` is a *reference* to that object). The model description is the (optional) string `"Simple linear optimization"`, passed as an argument.

There is a number of actions that can be done with objects of type `Model`, allowing us to add variables and constraints to the model before solving it. We start defining variables x_1, x_2, x_3 (in the program, `x1, x2, x3`). We can generate a variable using the method `addVar` of the model object created above (a *method* is a function associated with objects of a class). For example, to generate a variable `x1` we use the following statement:

```python
x1 = model.addVar(vtype="C", name="x1")
```

With this statement, the method `addVar` of class `Model` is called, creating a variable `x1` (to be precise, `x1` holds a reference to the variable object). In Python, *arguments* are values passed to a function or method when calling it (each argument corresponds to a *parameter* that has been specified in the function definition). Arguments to this method are specified within parenthesis after `addVar`. There are several ways to specify arguments in Python, but the clearest way is to write `argument name = argument value` as a *keyword argument*.

Here, `vtype = "C"` indicates that this is a continuous variable, and `name = "x1"` indicates that its name (used, e.g., for printing) is the string `"x1"`. The complete signature (i.e., the set of parameters) for the `addVar` method is the following:

```python
addVar(name="", vtype="C", lb=0.0, ub=None, obj=0.0, pricedVar = False)
```

Arguments are, in order, the name, the type of variable, the lower bound, the upper bound, the coefficients in the objective function. The last parameter, `pricedVar` is used for *column generation*, a method that will be explained in Chapter *Bin packing and cutting stock problems*. In Python, when calling a method omitting keyword arguments (which are optional) default values (given after =) are applied. In the case of `addVar`, all the parameters are optional. This means that if we add a variable with `model.addVar()`, SCIP will create a continuous, non-negative and unbounded variable, whose name is an empty string, with coefficient 0 in the objective (`obj=0`). The default value for the lower bound is specified with `lb=0.0`, and the upper bound `ub` is implicitly assigned the value infinity (in Python, the constant `None` usually means the absence of a value). When calling a function or method, keyword arguments without a default value cannot be omitted.

Functions and methods may also be called by writing the arguments without their name, in a predetermined order, as in:

---

[3] Of course "anything" is an exaggeration. In a Python lecture found at the Massachusetts Institute of Technology home page there is a reference to an antigravity module. Please try it with `import antigravity`.

```
x1 = model.addVar("x1", "C", 0, None, 15)
```

Other variables may be generated similarly. Note that the third constraint $x3 \leq 30$ is the upper bound constraint of variable x_3, so we may write `ub = 30` when declaring the variable.

Next, we will see how to enter a constraint. For specifying a constraint, we will need to create a *linear expression*, i.e., an expression in the form of $c_1x_1 + c_2x2 + \ldots + c_nx_n$, where each c_i is a constant and each x_i is a variable. We can specify a linear constraint through a relation between two linear expressions. In SCIP's Python interface, the constraint $2x1 + x2 + x3 \leq 60$ is entered by using method `addConstr` as follows:

```
model.addConstr(2*x1 + x2 + x3 <= 60)
```

The signature for `addConstr` (ignoring some parameters which are not of interest now) is:

```
addConstr(relation, name="", ...)
```

SCIP supports more general cases, but for the time being let us concentrate on linear constraints. In this case, parameter `relation` is a linear constraint, including a *left-hand side* (lhs), a *right-hand side* (rhs), and the sense of the constraint. Both *lhs* and *rhs* may be constants, variables, or linear expressions; *sense* maybe `"<="` for less than or equal to, `">="` for greater than or equal to, or `"=="` for equality. The name of the constraint is optional, the default being an empty string. Linear constraints may be specified in several ways; for example, the previous constraint could be written equivalently as:

```
model.addConstr(60 >= 2*x1 + x2 + x3)
```

Before solving the model, we must specify the objective using the `setObjective` method, as in:

```
model.setObjective(15*x1 + 18*x2 + 30*x3, "maximize")
```

The signature for `setObjective` is:

```
setObjective(expression, sense="minimize", clear="true"):
```

The first argument of `setObjective` is a linear (or more general) expression, and the second argument specifies the direction of the objective function with strings `"minimize"` (the default) or `"maximize"`. (The third parameter, `clear`, if `"true"` indicates that coefficients for all other variables should be set to zero.) We may also set the direction of optimization using `model.setMinimize()` or `model.setMaximize()`.

At this point, we can solve the problem using the method `optimize` of the `model` object:

```
model.optimize()
```

After executing this statement — if the problem is feasible and bounded, thus allowing completion of the solution process —, we can output the optimal value of each variable. This can be done through method `getVal` of `Model` objects; e.g.:

```
print(model.getVal(x1))
```

The complete program for solving our model can be stated as follows:

```
from pyscipopt import Model

model = Model("Simple linear optimization")

x1 = model.addVar(vtype="C", name="x1")
x2 = model.addVar(vtype="C", name="x2")
```

(continues on next page)

```
7   x3 = model.addVar(vtype="C", name="x3")
8
9   model.addCons(2*x1 + x2 + x3 <= 60)
10  model.addCons(x1 + 2*x2 + x3 <= 60)
11  model.addCons(x3 <= 30)
12
13  model.setObjective(15*x1 + 18*x2 + 30*x3, "maximize")
14
15  model.optimize()
16
17  if model.getStatus() == "optimal":
18      print("Optimal value:", model.getObjVal())
19      print("Solution:")
20      print("  x1 = ", model.getVal(x1))
21      print("  x2 = ", model.getVal(x2))
22      print("  x3 = ", model.getVal(x3))
23  else:
24      print("Problem could not be solved to optimality")
```

If we execute this Python program, the output will be:

```
1   [solver progress output omitted]
2   Optimal value: 1230.0
3   Solution:
4     x1 =   10.0
5     x2 =   10.0
6     x3 =   30.0
```

The first lines, not shown, report progress of the SCIP solver (this can be suppressed) while lines 2 to 6 correspond to the output instructions of lines 14 to 16 of the previous program.

Note: Margin seminar 1

Linear programming

Linear programming was proposed by George Dantzig in 1947, based on the work of three Nobel laureate economists: Wassily Leontief, Leonid Kantrovich, Tjalling Koopmans. At that time, the term used was "optimization in linear structure", but it was renamed as "linear programming" in 1948, and this is the name commonly used afterwards. The simplex method developed by Dantzig has long been the almost unique algorithm for linear optimization problems, but it was pointed out that there are (mostly theoretical) cases where the method requires a very long time.

The question as to whether linear optimization problems can be solved efficiently in the theoretical sense (in other words, whether there is an algorithm which solves linear optimization problems in polynomial time) has been answered when the ellipsoid method was proposed by Leonid Khachiyan (Khachian), of the former Soviet Union, in 1979. Nevertheless, the algorithm of Khachiyan was only theoretical, and in practice the supremacy of the simplex method was unshaken. However, the interior point method proposed by Narendra Karmarkar in 1984[4] has been proved to be theoretically efficient, and in practice it was found that its performance can be similar or higher than the simplex method's. Currently available optimization solvers are usually equipped with both the simplex method (and its dual version, the *dual simplex method*) and with interior point methods, and are designed so that users can choose the most appropriate of them.

---

[4] Sometimes it is called a barrier method.

2.3 Integer Optimization

For many real-world optimization problems, sometimes it is necessary to obtain solutions composed of integers instead of real numbers. For instance, there are many puzzles like this: *"In a farm having chicken and rabbits, there are 5 heads and 16 feet. How many chicken and rabbits are there?"* Answer to this puzzle is meaningful if the solution has integer values only.

Let us consider a concrete puzzle.

Adding the number of heads of cranes, turtles and octopuses totals 32, and the number of legs sums to 80. What is the minimum number of turtles and octopuses?

Let us formalize this as an optimization problem with mathematical formulas. This process of describing a situation algebraically is called the *formulation* of a problem in mathematical optimization.

Then, the number of heads can be expressed as $x + y + z$. Cranes have two legs each, turtles have four legs each, and each octopus has eight legs. Therefore, the number of legs can be expressed as $2x + 4y + 8z$. So the set of x, y, z must satisfy the following "constraints":

$$\text{subject to:}$$
$$x +$$
$$y +$$
$$z =$$
$$32$$

$$2x +$$
$$4y +$$
$$8z =$$
$$80$$

Since there are three variables and only two equations, there may be more than one solution. Therefore, we add a condition to minimize the sum $y + z$ of the number of turtles and octopuses. This is the "objective function". We

obtain the complete model after adding the non-negativity constraints.

$$\text{minimize}$$

$$y + z$$

$$\text{subject to:}$$

$$x + y + z = 32$$

$$2x + 4y + 8z = 80$$

$$x, y, z \geq 0$$

When we use a linear optimization solver, we obtain the solution $x = 29.3333, y = 0, z = 2.66667$. This is obviously a strange answer. Cranes, tortoises and octopuses can be divided when they are lined up as food on the shelves, but not when they are alive. To solve this model, we need to add conditions to force the variables to have integer values. These are called *integrality constraints*: x, y, z must be non-negative integers. Linear optimization problems with conditions requiring variables to be integers are called *integer optimization problems*. For the puzzle we are solving, thus, the

correct model is:

$$\text{minimize}$$

$$y + z$$

$$\text{subject to:}$$

$$x + y + z = 32$$

$$2x + 4y + 8z = 80$$

$$x, y, z \geq 0, \text{integer}$$

Below is a simple Python/SCIP program for solving it. The main difference with respect to the programs that we have seen before concerns the declaration of variables; in this case, there is an argument to `addVar` for specifying that variables are integer: `vtype="I"`. Continuous variables (the default) can be explicitly declared with `vtype="C"`, and binary variables — a special case of integers, restricted to the values 0 or 1 — are declared with `vtype="B"`.

```python
from pyscipopt import Model

model = Model("Simple linear optimization")

x = model.addVar(vtype="I", name="x")
y = model.addVar(vtype="I", name="y")
z = model.addVar(vtype="I", name="x")

model.addCons(x + y + z == 32,"Heads")
model.addCons(2*x + 4*y + 8*z == 80,"Legs")
model.setObjective(y + z, "minimize")

model.optimize()

if model.getStatus() == "optimal":
    print("Optimal value:", model.getObjVal())
    print("Solution:")
    print("  x = ", model.getVal(x))
    print("  y = ", model.getVal(y))
    print("  z = ", model.getVal(z))
else:
    print("Problem could not be solved to optimality")
```

For small integer optimization problems like this, the answer can be quickly found: $x = 28$, $y = 2$, and $z = 2$, meaning that there are 28 cranes, 2 turtles and 2 octopuses. Notice that this solution is completely different of the

continuous version's; in general, we cannot guess the value of an integer solution from the continuous model. In general, integer-optimization problems are much harder to solve when compared to linear-optimization problems.

[source code]

2.4 Transportation Problem

The next example is a classical linear optimization problem called the *transportation problem*. Consider the following scenario.

You are the owner of a sports equipment sales chain. Your products are manufactured at three factories, and you have to deliver them to five customers (demand points) (Figure *Transportation problem*). After elaborating a survey, you found that the production capacity at each factory, the transportation cost to customers, and the demand amount at each customer are as shown in Table *Data for the transportation problem*. So, which of the transport routes would you choose to minimize the total cost?
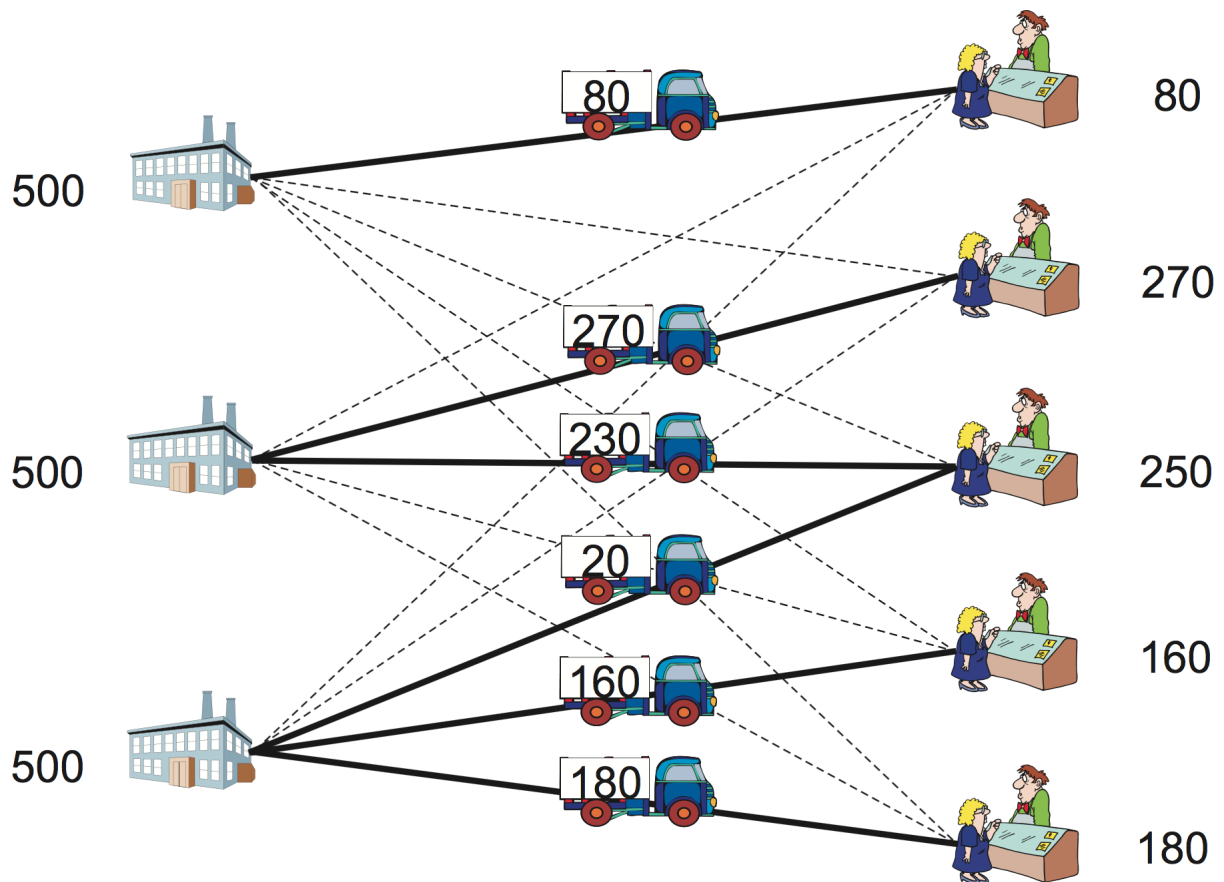

Fig. 1: Transportation problem
Graph representation of a transportation problem and its optimal transport volume.

Table 1: Data for the transportation problem

Transportation cost c_{ij}		Customers i					capacity M_j
		1	2	3	4	5	
plant j	1	4	5	6	8	10	500
	2	6	4	3	5	8	500
	3	9	7	4	2	4	500
demand d_i		80	270	250	160	180	

Table *Data for the transportation problem* shows customer demand volumes, shipping costs from each factory to each customer, and production capacity at each factory. More precisely, d_i is the demand of customer i, where $i = 1$ to 5. Each plant j can supply its customers with goods but their production capacities are limited by M_j, where $j = 1$ to 3. Transportation cost for shipping goods from plant i to customer j is given in the table by c_{ij}.

Let us formulate the above problem as a linear optimization model. Suppose that the number of customers is n and the number of factories is m. Each customer is represented by $i = 1, 2, \ldots, n$, and each factory by $j = 1, 2, \ldots, m$. Also, let the set of customers be $I = 1, 2, \ldots, n$ and the set of factories $J = 1, 2, \ldots, m$. Suppose that the demand amount of customer i is d_i, the transportation cost for shipping one unit of demand from plant i to customer j is c_{ij}, and that each plant j can supply its customers with goods, but their production capacities are limited by M_j

We use continuous variables as defined below.

$$x_{ij} = \text{amount of goods to be transported from factory } j \text{ to customer } i$$

Using the above symbols and variables, the transport problem can be formulated as the following linear optimization problem.

$$\text{minimize}$$
$$\sum_{i \in I} \sum_{j \in J} c_{ij} x_{ij}$$
$$\text{subject to}$$
$$\sum_{j \in J} x_{ij} = d_i \quad \forall i \in I$$

$$\sum_{i \in I} x_{ij} \leq M_j \quad \forall j \in J$$

$$x_{ij} \geq 0 \quad \forall i \in I, j \in J$$

The objective function is the minimization of the sum of transportation costs. The first constraint requires that the demand is satisfied, and the second constraint ensures that factory capacities are not exceeded.

Let us solve this with Python/SCIP. First, we prepare the data needed for describing an instance[5]. In the transportation problem, it is necessary to prepare data defining demand amount d_i, transportation costs c_{ij}, capacities M_j. In the following program, we will use the same symbol used in the formulation for holding a Python's dictionary. A dictionary is composed of a key and a value as its mapping, and is generated by arranging pais of keys and values in brackets, separated by commas: `{key1:value1, key2:value2, ...}`. (For details on dictionaries see appendix A.2.5).

The demand amount d_i is stored in a dictionary `d` with the customer's number as the key and the demand amount as the value, and the capacity M_j is stored in the dictionary `M` with the factory number as the key and the capacity as the value.

---

[5] A problem with all the parameters substituted by numerical values is called *an instance*; this meaning is different of "objects generated from a class", used in object-oriented programming, which are also called "instances" of the class.

```
d = {1:80 , 2:270 , 3:250 , 4:160 , 5:180}
M = {1:500 , 2:500 , 3:500}
```

In addition, a list `I` of customers' numbers and a list `J` of factory numbers can be prepared as follows.

```
I = [1,2,3,4,5]
J = [1,2,3]
```

Actually, the dictionaries and lists above can be created at once by using the `multidict` function available in Python/SCIP, as follows.

```
I, d = multidict({1:80, 2:270, 3:250, 4:160, 5:180})
J, M = multidict({1:500, 2:500, 3:500})
```

When the dictionary is entered as an argument, the `multidict` function returns a pair of values; the first is the list of keys, and the second value is the dictionary sent as argument. Later, we will see that this function is very useful when we want to associate more than one value to each key. (For a more detailed usage of `multidict`, see appendix B.4.)

Shipping cost c_{ij} has two subscripts. This is represented in Python by a dictionary `c` with a tuple of subscripts (customer and factory) as keys, and the corresponding costs as values. A tuple is a sequence, like a list; however, unlike a list, its contents can not be changed: a tuple is *immutable*. Tuples are created using parentheses and, due to the fact that they are immutable, can be used as keys of a dictionary (see appendix A.2.4 for details on tuples).

```
c = {(1,1):4,     (1,2):6,     (1,3):9,
      (2,1):5,     (2,2):4,     (2,3):7,
      (3,1):6,     (3,2):3,     (3,3):3,
      (4,1):8,     (4,2):5,     (4,3):3,
      (5,1):10,    (5,2):8,     (5,3):4,
      }
```

With this dictionary `c`, the transportation cost from factory j to customer i can be accessed with `c[(i,j)]` or `c[i,j]` (in a tuple, we can omit parenthesis).

> **Attention:** As a programming habit, it is preferable not to use a one-letter variables such as `d`, `M`, `c` above. We have done it so that the same symbols are used in the formulation and in the program. However, in larger programs it is recommended to use meaningful variables names, such as `demand`, `capacity`, `cost`.

Let us write a program to solve the instance specified above.

```
1  model = Model("transportation")
2  x = {}
3  for i in I:
4      for j in J:
5          x[i,j] = model.addVar(vtype="C", name="x(%s,%s)" % (i,j))
```

First, we define a Python variable `x`, which initially contains an empty dictionary (line 2). We then use dictionary `x` to store variable's objects, each of them corresponding to an x_{ij} of our model (lines 3 to 5). As `I` is a list of customers' indices, the `for` cycle of line 3 iterates over all customers i. Likewise, since `J` is a list of factory indices, the `for` cycle of line 4 iterates over the quantity transported from factory j to customer i (see appendix A.4.2 for more information about iteration). In the rightmost part of line 5 the variable is named `x(i,j)`; this uses Python's string format operation `%`, where `%s` represents substitution into a character string.

Next we add constraints. First, we add the constraint

$$\sum_{j \in J} x_{ij} = d_i \quad \forall i \in I$$

which imposes that the demand is satisfied. Since this is a constraint for all customers i, a constraint $\sum_{j=1}^{m} x_{ij} = d_i$ is added by the `addCons` method (line 2) at each iteration of the `for` cycle of line 1.

```
for i in I:
    model.addCons(quicksum(x[i,j] for j in J if (i,j) in x) == d[i], name="Demand(%s)
    " % i)
```

Notice that here we also give a name, `Demand(i)`, to constraints. Although, as for variables, the name of a constraint may be omitted, it is desirable to add an appropriate name for later reference (an example of this will be seen in *Duality*). The `quicksum` function on the second line is an enhanced version of the `sum` function available in Python, used in Python/SCIP to do the computation of linear expressions more efficiently. It is possible to provide `quicksum` explicitly with a list, or with a list generated by iteration with a `for` statement, as we did here; these *generator* work in the same way as in list comprehensions in Python (see appendix A.4.2). In the above example, we calculate a linear expression by summing variables x_{ij} for element $j \in J$ by means of `quicksum(x[i,j] for j in J)`. (For a more detailed explanation of `quicksum`, see appendix B.4.)

Similarly, we add the factory capacity constraint

$$\sum_{i \in I} x_{ij} \leq M_j$$

$$\forall j \in J$$

to the model as follows:

```
for j in J:
    model.addCons(quicksum(x[i,j] for i in I if (i,j) in x) <= M[j], name="Capacity(
    %s)" % j)
```

Again, we give a name `Capacity(j)` to each constraint. In the following, to simplify the description, names of constraints are often omitted; but in fact it is safer to give an appropriate name.

The objective function

$$\text{minimize} \quad \sum_{i \in I} \sum_{j \in J} c_{ij} x_{ij}$$

is set using the `setObjective` method, as follows.

```
model.setObjective(quicksum(c[i,j]*x[i,j]  for (i,j) in x), "minimize")
```

Finally, we can optimize the model and display the result.

```
model.optimize()
print("Optimal value:", model.getObjVal())
EPS = 1.e-6
for (i,j) in x:
    if model.getVal(x[i,j]) > EPS:
        print("sending quantity %10s from factory %3s to customer %3s" % (model.
        getVal(x[i,j]),j,i))
```

In this code, `for (i,j) in x` in line 4 is an iteration over dictionary `x`, holding our model's variable. This iteration goes through all the tuples (i, j) of customers and factories which are keys of the dictionary. Line 5 is a conditional statement for outputting only non-zero variables. Line 6 uses Python's string formatting operator `%`, where `%10s` is converted into a 10-digit character string and `%3s` is converted into a 3-digit character string.

When the above program is executed, the following result is obtained. The results are shown in Table *Optimal solution for the transportation problem* and Figure *Transportation problem*.

```
1  [solver progress output omitted]
2  SCIP Status          : problem is solved [optimal solution found]
3  Solving Time (sec) : 0.00
4  Solving Nodes        : 1
5  Primal Bound         : +3.35000000000000e+03 (1 solutions)
6  Dual Bound           : +3.35000000000000e+03
7  Gap                  : 0.00 %
8  Optimal value: 3350.0
9  sending quantity     230.0 from factory   2 to customer   3
10 sending quantity      20.0 from factory   3 to customer   3
11 sending quantity     160.0 from factory   3 to customer   4
12 sending quantity     270.0 from factory   2 to customer   2
13 sending quantity      80.0 from factory   1 to customer   1
14 sending quantity     180.0 from factory   3 to customer   5
```

Table 2: Optimal solution for the transportation problem

Customer i	1	2	3	4	5		
Amount demanded	80	270	250	160	180		
Plant j	Optimum volume transported					total	capacity
1	80					80	500
2		270	230			500	500
3			20	160	180	360	500

[source code]

2.5 Duality

Consider the following scenario.

You are the owner of the sports equipment sales chain that appeared on Section *Transportation Problem*. You feel that factory's capacity has become tight, so you are considering an expansion. What kind of expenses can be expected to be reduced by expanding each of the factories? Also, what is the additional gain that you can you get if you have additional orders from each customer?

In order to solve this problem smartly, the concept of *dual problem* is useful. Here, the dual problem is a linear optimization problem associated to the original problem (which in this context is called the *primal* problem). The derivation method and meaning of the dual problem are given in *Margin seminar 2*; here, we will explain how to use information from the dual of the transportation problem with Python/SCIP.

In order to investigate whether or not a factory can be expanded, let us first focus on the capacity constraint

$$\sum_{i \in I} x_{ij} \leq M_j$$

$$\forall j \in J.$$

For such an inequality constraint, a variable representing the difference between the right and the left hand sides, $M_j - \sum_{i \in I} x_{ij}$, is called a *slack variable*. Of course, one can easily calculate slack variables from the optimal solution, but in Python/SCIP we can look at the `getSlack` attribute for constraint objects. Also, the optimal dual variable for a constraint can be obtained with the `getDualsolLinear` attribute. This represents the amount of reduction on costs when increasing the capacity constraint by one unit (see *Margin seminar 2*).

In order to estimate the cost of additional orders from customers, we focus on the demand satisfaction constraint

$$\sum_{j \in J} x_{ij} = d_i \quad \forall i \in I$$

Since this is an equality constraint, all slack variables are 0. The optimal value of the dual variable associated with this constraint represents the increase in costs as demand increases by one unit.

Note: Margin seminar 2

Duality

Duality in linear programming provides a unifying theory that develops the relationships between a given linear program and another related linear program, stated in terms of variables with this shadow-price interpretation. The importance of duality is twofold. First, fully understanding the shadow-price interpretation of the optimal simplex multipliers can prove very useful in understanding the implications of a particular linear-programming model. Second, it is often possible to solve the related linear program with the shadow prices as the variables in place of, or in conjunction with, the original linear program, thereby taking advantage of some computational efficiencies. The importance of duality for computational procedures will become more apparent in later chapters on network-flow problems and large-scale systems.

Let us re-visit the wine production problem considered earlier to discuss some important concepts in linear-optimization models that play vital role in *sensitivity analysis*. Sensitivity analysis is important for finding out how optimal solution and optimal value may change when there is any change to the data used in the model. Since data may not always be considered as totally accurate, such analysis can be very helpful to the decision makers.

Let us assume that an entrepreneur is interested in the wine making company and would like to buy its resources. The entrepreneur then needs to find out how much to pay for each unit of each of the resources, the pure-grape wines of 2010 A, B and C. This can be done by solving the *dual* version of the model that we will discuss next.

Let y_1, y_2 and y_3 be the price paid, per barrel of Alfrocheiro, Baga, and Castelão, respectively. Then, the total price that should be paid is the quantities of each of the wines in inventory times their prices, i.e., $60y_1 + 60y_2 + 30y_3$. Since the entrepreneur would like the purchasing cost to be minimum, this is the objective function for minimization. Now, for each of the resources, constraints in the model must ensure that prices are high enough for the company to sell to the entrepreneur. For instance, with two barrels of A and one barrel of B, the company can prepare blend D worth 15; hence, it must be offered $2y_1 + y_2 \geq 15$. Similarly we obtain $y_1 + 2y_2 \geq 18$ and $y_1 + y_2 + y_3 \geq 30$ for the blends M and S, respectively. Thus we can formulate a dual model, stated as follows (for a more sound derivation,

using Lagrange multipliers, see lagrange).

$$\text{minimize}$$
$$60y_1 +$$
$$60y_2 +$$
$$30y_3$$
$$\text{subject to:}$$
$$2y_1 +$$
$$y_2$$
$$\geq$$
$$15$$

$$y_1 +$$
$$2y_2$$
$$\geq$$
$$18$$

$$y_1 +$$
$$y_2 +$$
$$y_3 \geq$$
$$30$$

$$y_1,$$
$$y_2,$$
$$y_3 \geq$$
$$0$$

The variables used in the linear-optimization model of the production problem are called *primal variables* and their solution values directly solve the optimization problem. The linear-optimization model in this setting is called the *primal model*.

As seen above, associated with every primal model, there is a dual model. The relationships between primal and dual problems can provide significant information regarding the quality of solutions found and sensitivity of the coefficients used. Moreover, they also provide vital economic interpretations. For example, y_1, the price paid for one unit of Alfrocheiro pure-grape wine is called the *shadow price* of that resource, because it is the amount by which the optimal value of the primal model will change for a unit increase in its availability — or, equivalently, the price the company would be willing to pay for an additional unit of that resource.

Gurobi allows us to access the shadow prices (i.e., the optimal values of the dual variables associated with each constraint) by means of the `.Pi` attribute of the constraint class; e.g., in the model for the wine production company of program wblending we are printing these values in line 31.

Another concept important in duality is the *reduced cost*, which is associated with each decision variable. It is defined as the change in objective function value if one unit of some product that is normally not produced is forced into production; it can also be seen as the amount that the coefficient in the objective has to improve, for a variable that is zero in the optimal solution to become non-zero. Therefore, reduced cost is also appropriately called *opportunity cost*. Shadow prices and reduced costs allow *sensitivity analysis* in linear-optimization and help determine how sensitive the solutions are to small changes in the data. Such analysis can tell us how the solution will change if the objective function coefficients change or if the resource availability changes. It can also tell us how the solution may change if a new constraint is brought into the model. Gurobi allows us accessing the reduced costs through the `.RC` attribute of the variable class; e.g., `x.RC` is the reduced cost of variable `x` in the optimal solution.

As we will see later, primal and dual models can be effectively used not only to gain insights into the solution but also to find a bound for the *linear-optimization relaxation* of an integer-optimization model; linear-optimization relaxation is obtained by having the integrality constraints relaxed to non-integer solution values. Typically, an integer-optimization model is much harder to solve than its linear-optimization relaxation. Specialized algorithms have been designed around the relaxation versions of primal as well as dual optimization models for finding optimal solution more efficiently. Optimal solution of a relaxation model gives a bound for the optimal solution value of the underlying integer-optimization model, and that can be exploited in a *branch-and-bound* scheme for solving the integer optimization model.

[source code]

2.6 Multi-product Transportation Problem

In the previous transportation problem, we considered only one kind of goods produced at the production plants. In the real-world, however, that is a very restrictive scenario: A producer typically produces many different kinds of products and the customers typically demand different sets of the products available from the producers. Moreover, some producers may be specialized into producing only certain kinds of products while some others may only supply to certain customers. Therefore, a general instance of the transportation problem needs to be less restrictive and account for many such possibilities.

A more general version of the transportation problem is typically studied as a multi-commodity transportation model. A linear-optimization model can be built using decision variables x_{ijk} where i denotes the customer, j denotes the production plant and k denotes the product type. Customer demand is indexed by i and k to denote the customer and product type. Then the model can be stated as follows.

$$\text{minimize}$$

$$\sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{K}c_{ijk}x_{ijk}$$

$$\text{subject to}$$

$$\sum_{j=1}^{m}x_{ijk} = d_{ik} \text{ for } i=1,\cdots,n, k=1,\cdots,K$$

$$\sum_{i=1}^{n}\sum_{k=1}^{K}x_{ijk} \leq M_j \text{ for } j=1,\cdots,m$$

$$x_{ijk} \geq 0 \text{ for } i=1,\cdots,n, j=1,\cdots,m, k=1,\cdots,K$$

Note that the objective function addresses the minimum total cost for all possible cost combinations involving customers, production plants and product types. The first set of constraints ensure that all demands of the product types from the customers are met exactly while the second set of constraints ensure that capacity at each production plant is not exceeded by taking into account all product types and all customers.

A model for this in Python/Gurobi can be written as follows:

```
1   def mctransp(I, J, K, c, d, M):
2       model = Model("multi-commodity transportation")
3       x = {}
4       for i,j,k in c:
5           x[i,j,k] = model.addVar(vtype="C", name="x[%s,%s,%s]" % (i, j, k))
6       model.update()
```
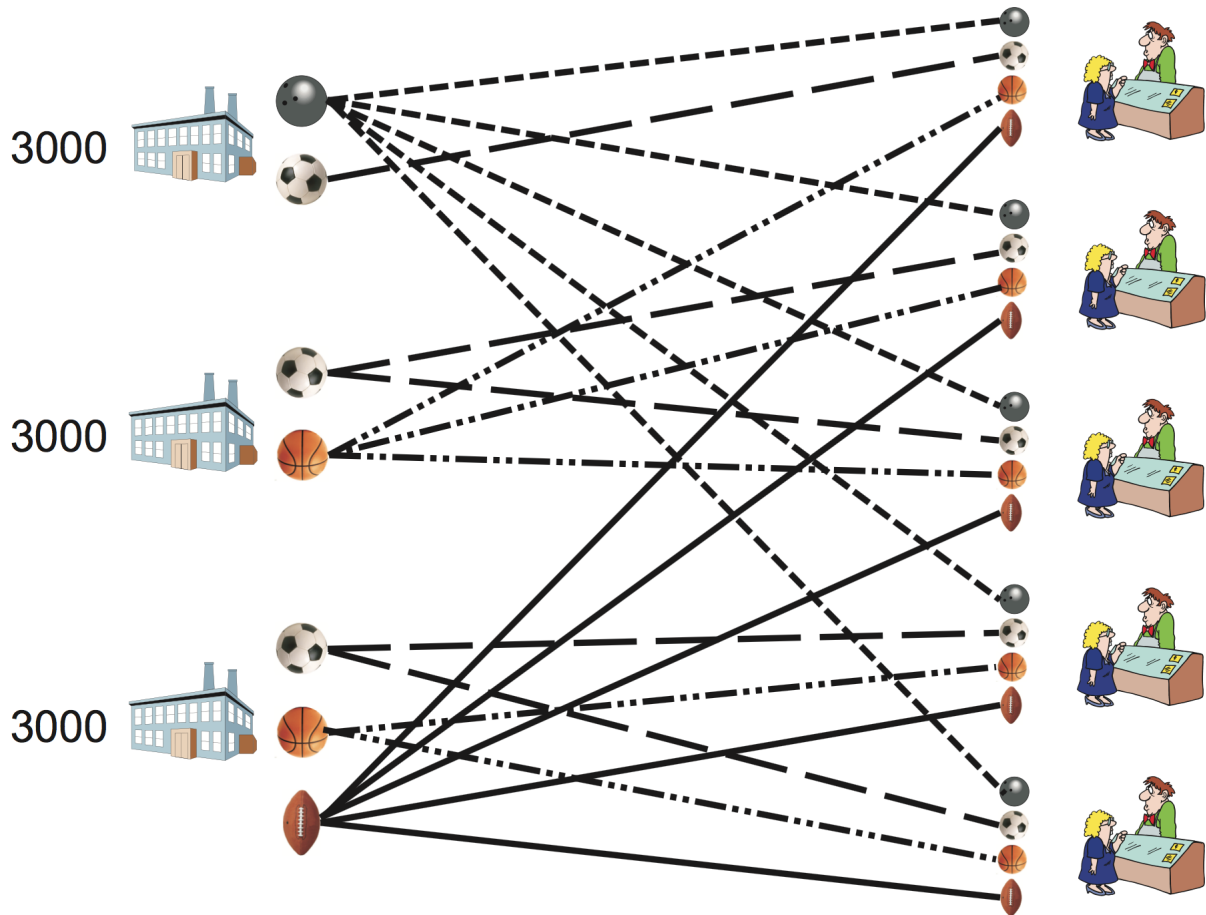
(continues on next page)

Fig. 2: Multicommodity transportation

Graph representation for a multicommodity transportation problem. Suppliers are represented as squares and clients as circles; thick lines represent arcs actually used for transportation in a possible solution, and colors in arcs mean different products.

```
 7          for i in I:
 8              for k in K:
 9                  model.addConstr(quicksum(x[i,j,k] for j in J if (i,j,k) in x) == d[i,k],
    ↪"Demand[%s,%s]" % (i,k))
10          for j in J:
11              model.addConstr(quicksum(x[i,j,k] for (i,j2,k) in x if j2 == j) <= M[j],
    ↪"Capacity[%s]" % j)
12          model.setObjective(quicksum(c[i,j,k]*x[i,j,k]   for i,j,k in x), GRB.MINIMIZE)
13          model.update()
14          model.__data = x
15          return model
```

Variables are created in line 5. In lines 9 and 10 we create a list the variables that appear in each demand-satisfaction constraint, and the corresponding coefficients; these are then used for creating a linear expression, which is used as the left-hand side of a constraint in line 11. Capacity constraints are created in a similar way, in lines 13 to 15. For an example, consider now the same three production plants and five customers as before. Plant 1 produces two products, football and volleyball; it can supply football only to Customer 1 and volleyball to all five customers. Plant 2 produces football and basketball; it can supply football to Customers 2 and 3, basketball to Customers 1, 2 and 3. Plant 3 produces football, basketball and rugby ball; it can supply football and basketball to Customers 4 and 5, rugby ball to all five customers.

Let us specify the data for this problem in a Python program. First of all, we must state what products each of the plants can manufacture; on dictionary `produce` the key is the plant, to which we are associating a list of compatible products. We also create a dictionary `M` with the capacity of each plant (3000 units, in this instance).

```
J,M = multidict({1:3000, 2:3000, 3:3000})
produce = {1:[2,4], 2:[1,2,3], 3:[2,3,4]}
```

The demand for each of the customers can be written as a double dictionary: for each customer, we associate a dictionary of products and quantities demanded.

```
d = {(1,1):80,   (1,2):85,   (1,3):300,  (1,4):6,
     (2,1):270,  (2,2):160,  (2,3):400,  (2,4):7,
     (3,1):250,  (3,2):130,  (3,3):350,  (3,4):4,
     (4,1):160,  (4,2):60,   (4,3):200,  (4,4):3,
     (5,1):180,  (5,2):40,   (5,3):150,  (5,4):5
     }
I = set([i for (i,k) in d])
K = set([k for (i,k) in d])
```

For determining the transportation cost, we may specify the unit weight for each product and the transportation cost per unit of weight; then, we calculate c_{ijk} as their product:

```
weight = {1:5, 2:2, 3:3, 4:4}
cost = {(1,1):4,   (1,2):6, (1,3):9,
        (2,1):5,   (2,2):4, (2,3):7,
        (3,1):6,   (3,2):3, (3,3):4,
        (4,1):8,   (4,2):5, (4,3):3,
        (5,1):10, (5,2):8, (5,3):4
        }
c = {}
for i in I:
    for j in J:
        for k in produce[j]:
            c[i, j, k] = cost[i,j] * weight[k]
```

We are now ready to construct a model using this data, and solving it:

```
model = mctransp(c,d,M)
model.optimize()
print "Optimal value:", model.ObjVal
EPS = 1.e-6
x = model.__data
for i,j,k in x:
    if x[i,j,k].X > EPS:
        print "sending %10g units of %3d from plant %3d to customer %3d" % (x[i,j,k].
→X, k, j, i)
```

If we execute this Python program, the output is the following:

```
1   Optimize a model with 18 rows, 40 columns and 70 nonzeros
2   Presolve removed 18 rows and 40 columns
3   Presolve time: 0.00s
4   Presolve: All rows and columns removed
5   Iteration    Objective       Primal Inf.    Dual Inf.      Time
6           0    1.7400000e+04   0.000000e+00   0.000000e+00     0s
7
8   Solved in 0 iterations and 0.00 seconds
9   Optimal objective  1.740000000e+04
10  Optimal value: 17400.0
11  sending      100.0 units of   2 from plant    3 to customer    4
12  sending      210.0 units of   3 from plant    3 to customer    3
13  sending       40.0 units of   3 from plant    2 to customer    3
14  sending       40.0 units of   1 from plant    2 to customer    1
15  sending       10.0 units of   3 from plant    2 to customer    1
16  sending      100.0 units of   2 from plant    1 to customer    2
17  sending      100.0 units of   3 from plant    2 to customer    2
18  sending       70.0 units of   1 from plant    2 to customer    2
19  sending       60.0 units of   1 from plant    2 to customer    4
20  sending       30.0 units of   2 from plant    1 to customer    1
21  sending      180.0 units of   1 from plant    2 to customer    5
```

Readers may have noticed by now that for these two transportation problems, even though we have used linear-optimization models to solve them, the optimal solutions are integer-valued — as if we have solved integer-optimization models instead. This is because of the special structures of the constraints in the transportation problems that allow this property, commonly referred to as *unimodularity*. This property has enormous significance because, for many integer-optimization problems that can be modeled as transportation problems, we only need to solve their linear-optimization relaxations.

[source code]

2.7 Blending problem

2.8 Fraction optimization problem

2.9 Multi-Constrained Knapsack Problem

Knapsack problems are specially structured optimization problems. The general notion of the *knapsack problem* is to fill up a knapsack of certain capacity with items from a given set such that the collection has maximum value with respect to some special attribute of the items. For instance, given a knapsack of certain volume and several items of different weights, the problem can be that of taking the heaviest collection of the items in the knapsack. Based on

weights, the knapsack can then be appropriately filled by a collection that is optimal in the context of weight as the special attribute.

Suppose we have a knapsack of volume 10,000 cubic-cm that can carry up to 7 Kg weight. We have four items having weights 2, 3, 4 and 5, respectively, and volume 3000, 3500, 5100 and 7200, respectively. Associated with each of the items is its value of 16, 19, 23 and 28, respectively. We would like to fill the knapsack with items such that the total value is maximum.

Fig. 3: Knapsack instance

An integer-optimization model of this problem can be found by defining the decision variables $x_j = 1$ if item j is taken, and $x_j = 0$ otherwise, where $j = 1$ to 4. For constraints, we need to make sure that total weight does not exceed 7 kg and total volume does not exceed 10,000 cubic-cm. Thus, we have an integer-optimization model:

$$\text{maximize}$$
$$16x_1 +$$
$$19x_2 +$$
$$23x_3 +$$
$$28x_4$$
$$\text{subject to:}$$
$$2x_1 +$$
$$3x_2 +$$
$$4x_3 +$$
$$5x_4 \leq$$
$$7$$

$$30x_1 +$$
$$35x_2 +$$
$$51x_3 +$$
$$72x_4 \leq$$
$$100$$

$$x_1,$$
$$x_2,$$
$$x_3,$$
$$x_4 \in$$
$$\{0, 1\}$$

The standard version of the knapsack problem concerns the maximization of the profit subject to a constraint limiting the weight allowed in the knapsack to a constant W; the objective is to maximize $\sum_j v_j x_j$ subject to $\sum_j w_j x_j \leq W$, with $x_j \in \{0, 1\}$, where v_j is the value of item j and w_j is its weight. A more general problem includes constraints in more than one dimension, say, m dimensions (as in the example above); this is called the multi-constrained knapsack problem, or m-dimensional knapsack problem. If we denote the *"weight"* of an object j in dimension i by a_{ij} and the capacity of the knapsack in this dimension by b_i, an integer-optimization model of this problem has the following

structure:

$$\text{maximize}$$

$$\sum_{j=1}^{n} v_j x_j$$

$$\text{subject to}$$

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad \text{for } i = 1, \cdots, m$$

$$x_j \in \{0, 1\} \quad \text{for } j = 1, \cdots, n$$

A Python/Gurobi model for the multi-constrained knapsack problem is:

```
def mkp(I, J, v, a, b):
    model = Model("mkp")
    x = {}
    for j in J:
        x[j] = model.addVar(vtype="B", name="x[%d]"%j)
    model.update()
    for i in I:
        model.addConstr(quicksum(a[i,j]*x[j] for j in J) <= b[i], "Dimension[%d]"%i)
    model.setObjective(quicksum(v[j]*x[j] for j in J), GRB.MAXIMIZE)
    model.update()
    return model
```

This model can be used to solve the example above in the following way:

```
J,v = multidict({1:16, 2:19, 3:23, 4:28})
a = {(1,1):2,    (1,2):3,    (1,3):4,    (1,4):5,
     (2,1):3000, (2,2):3500, (2,3):5100, (2,4):7200,
     }
I,b = multidict({1:7, 2:10000})

model = mkp(I, J, v, a, b)
model.ModelSense = -1
model.optimize()
print "Optimal value=", model.ObjVal
EPS = 1.e-6
for v in model.getVars():
    if v.X > EPS:
    print v.VarName,v.X
```

The solution of this example is found by Gurobi: $x_2 = x_3 = 1, x_1 = x_4 = 0$. We will next briefly sketch how this solution is found.

[source code]

2.9.1 Branch-and-bound

Many optimization problems, such as knapsack problems, require the solutions to have integer values. In particular, variables in the knapsack problem require values of either 1 or 0 for making decision on whether to include an item in the knapsack or not. Simplex method cannot be used directly to solve for such solution values because it cannot be used to capture the integer requirements on the variables. We can write the constraints $0 \leq x_j \leq 1$ for all j for the binary requirements on the variables, but the simplex method may give fractional values for the solution.

Therefore, in general, solving integer-optimization models is much harder. However, we can use a systematic approach called *branch-and-bound* for solving an integer-optimization model, using the simplex method for solving *linear-optimization relaxation* model obtained by "relaxing" any integer requirement on the variables to non-negatives only. The process begins with the linear-optimization relaxation of the integer-optimization model and solves several related linear-optimization models by simplex method for ultimately finding an optimal solution of the integer-optimization model.

Let us use the previous knapsack example to illustrate this procedure. We can transform this integer-optimization model of the knapsack problem to its linear-optimization relaxation by replacing the binary requirements by the constraints $0 \leq x_j \leq 1$ for all j. All feasible solutions of the integer-optimization model are also feasible for this linear-optimization relaxation; i.e., the polyhedron of the integer-optimization model is now contained within the polyhedron of its linear-optimization relaxation.

This linear-optimization relaxation can be solved easily by the simplex method. If the optimal solution found is feasible to the integer-optimization model also — i.e., it satisfies the binary constraints also, then we have found the optimal solution to the integer-optimization model. Otherwise, for this maximization problem, we can use the value of the optimal solution of the linear-optimization relaxation as the upper bound on the maximum value any solution of the integer-optimization model can possibly attain. Thus, optimal solution value of the linear-optimization relaxation provides an upper bound for the optimal solution value of the underlying integer-optimization model; this information can be suitably used for solving integer-optimization model via solving several related linear-optimization models.

The general notion of branch-and-bound scheme is to use bound on the optimal solution value in a tree search, as shown in Figure *Branch-and-bound*. Each leaf of the tree represents some linear-optimization relaxation of the original integer-optimization model. We start at the root of the search tree with the linear-optimization relaxation of the original integer-optimization model. Simplex method, gives the optimal solution $x = (1, 1, 0.5, 0)$ and objective function value 46.5. Since $x_3 = 0.5$ is not integer and for the original integer-optimization model we need the variables to be either 0 or 1, we create two different subproblem children of the root by forcing $x_3 = 1$ and $x_3 = 0$, say $P1$ and $P2$, respectively. Their optimal solutions are $x = (1, 1, 0, 0.4)$ with objective value 46.2 and $x = (1, 0.333, 1, 0)$ with objective value 45.333, respectively. Now these two subproblems can be expanded again by branching on their fractional values just as before. The process will yield a *binary search tree* because x_j can only take values of 0 and 1.
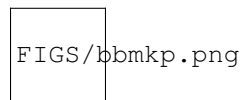

Fig. 4: Branch-and-bound
Branch-and-bound tree for the knapsack example.

Consider the two children of $P1$, $P3$ and $P4$. As found, the optimal solutions for $P3$ and $P4$ are $x = (0, 1, 1, 0)$ with objective function value 42 and $x = (1, 0, 1, 0.2)$ with objective function value 44.6, respectively. Since $P3$ gives us a feasible solution for the integer-optimization model, we have an *incumbent* solution $x = (0, 1, 1, 0)$ with value 42. If no other feasible solution to the integer-optimization model from the tree search produces objective value larger than 42, then the incumbent is the optimal solution.

As can be seen from this small example, exploring the whole solution space can lead to a very large number of computations, as the number of nodes may potentially duplicate from one level to the other. Gurobi uses branch-and-bound in connection to other advanced techniques, such as the *cutting plane* approach, in order to achieve a very good performance on this process. As we will see later (e.g., in Chapter *Graph problems*), there are some limitations to the size of the problems that can be tackled by Gurobi; however, a very large number of interesting, real-world problems can be solved successfully. In other situations, even if Gurobi cannot find the optimal solution, it will find a solution close to the optimum within reasonable time; in many applications, this is enough for practical implementation.

2.10 The Modern Diet Problem

In this section we consider a mathematical model for maximizing diversity of diet intakes, subject to nutritional requirements and calorie restrictions. Let $\mathcal{F}$ be a set of distinct foods and $\mathcal{N}$ be a set of nutrients. Let d_{ij} be the amount of nutrient i in food j. The minimum and maximum intake of each nutrient i is given by a_i and b_i, respectively. An upper bound for the quantity of each food is given by M. Let x_j be the number of dishes to be used for each food j, and let $y_j = 1$ indicate if food j is chosen, $y_j = 0$ if not. Let the cost of the diet be v and the amount of intake of each nutrient be given by z_i. The problem is to minimize the number of distinct foods.

$$\text{minimize}$$

$$\sum_{i \in \mathcal{F}} y_j$$

$$\text{subject to:}$$

$$\sum_{j \in \mathcal{F}} d_{ij} x_j = z_i \forall i \in \mathcal{N}$$

$$y_j \leq x_j \forall j \in \mathcal{F}$$

$$v = \sum_{j \in \mathcal{F}} c_j x_j$$

$$x_j \geq 0 \forall j \in \mathcal{F}$$

$$y_j \in \{0, 1\} \forall j \in \mathcal{F}$$

$$a_i \leq z_i \leq b_i \forall i \in \mathcal{N}$$

The first set of constraints (`Nutr` in the program below) calculate the amount of each nutrient by summing over the selection of foods. Together with the last set of constraints (which is entered as bounds on z, line 8 in the program below), they ensure that nutrient levels z_i are maintained within the maximum and minimum amounts, a_i and b_i, as required. The second set of constraints (`Eat` in the program below) impose that a dish variety y_j will be allowed into the objective (i.e., be non-zero) only if at least one unit of that dish x_j is selected. The third constraint (`Cost`, line 16 in the program) calculates cost v of selecting a diet, while the other two constraints impose non-negativity and binary requirements on the variables x_j and y_j defined earlier.

In Python/Gurobi, this model can be specified as follows.

```
def diet(F, N, a, b, c, d):
    model = Model("modern diet")
    x, y, z = {}, {}, {}
    for j in F:
        x[j] = model.addVar(lb=0, vtype="I", name="x[%s]" % j)
        y[j] = model.addVar(vtype="B", name="y[%s]" % j)
    for i in N:
        z[i] = model.addVar(lb=a[i], ub=b[i], name="z[%s]" % j)
    v = model.addVar(name="v")
    model.update()
    for i in N:
        model.addConstr(quicksum(d[j][i]*x[j] for j in F) == z[i], "Nutr[%s]" % i)
    model.addConstr(quicksum(c[j]*x[j]   for j in F) == v, "Cost")
    for j in F:
```

```
15          model.addConstr(y[j] <= x[j], "Eat[%s]" % j)
16      model.setObjective(quicksum(y[j]  for j in F), GRB.MAXIMIZE)
17      model.__data = x, y, z, v
18      return model
```

We may use the data provided in http://www.ampl.com/EXAMPLES/MCDONALDS/diet2.dat for applying this model to a concrete instance:

```
1   inf = GRB.INFINITY
2   N, a, b = multidict({
3   "Cal"     : [ 2000,  inf ],
4   "Carbo"   : [  350,  375 ],
5   "Protein" : [   55,  inf ],
6   "VitA"    : [  100,  inf ],
7   "VitC"    : [  100,  inf ],
8   "Calc"    : [  100,  inf ],
9   "Iron"    : [  100,  inf ],
10    })
11  F, c, d = multidict({
12  "QPounder":[1.84, {"Cal":510, "Carbo":34, "Protein":28, "VitA":15, "VitC":  6, "Calc
    ↪":30, "Iron":20}],
13  "McLean"  :[2.19, {"Cal":370, "Carbo":35, "Protein":24, "VitA":15, "VitC": 10, "Calc
    ↪":20, "Iron":20}],
14  "Big Mac" :[1.84, {"Cal":500, "Carbo":42, "Protein":25, "VitA": 6, "VitC":  2, "Calc
    ↪":25, "Iron":20}],
15  "FFilet"  :[1.44, {"Cal":370, "Carbo":38, "Protein":14, "VitA": 2, "VitC":  0, "Calc
    ↪":15, "Iron":10}],
16  "Chicken" :[2.29, {"Cal":400, "Carbo":42, "Protein":31, "VitA": 8, "VitC": 15, "Calc
    ↪":15, "Iron": 8}],
17  "Fries"   :[ .77, {"Cal":220, "Carbo":26, "Protein": 3, "VitA": 0, "VitC": 15, "Calc
    ↪": 0, "Iron": 2}],
18  "McMuffin":[1.29, {"Cal":345, "Carbo":27, "Protein":15, "VitA": 4, "VitC":  0, "Calc
    ↪":20, "Iron":15}],
19  "1%LFMilk":[ .60, {"Cal":110, "Carbo":12, "Protein": 9, "VitA":10, "VitC":  4, "Calc
    ↪":30, "Iron": 0}],
20  "OrgJuice":[ .72, {"Cal": 80, "Carbo":20, "Protein": 1, "VitA": 2, "VitC":120, "Calc
    ↪": 2, "Iron": 2}],
21    })
```

In this specification of data we have used a new feature of the `multidict` function: for the same key (e.g., nutrients), we may specify more than one value, and assign it to several Python variables; for example, in line 3 we are specifying both the minimum and the maximum intake amount concerning calories; respectively, `a` and `b`. We are now ready to solve the diet optimization model; let us do it for several possibilities concerning the maximum calorie intake `b["Cal"]`:

```
for b["Cal"] in [inf, 3500, 3000, 2500]:
    print "\n\nDiet for a maximum of %g calories" % b["Cal"]
    model = diet(F, N, a, b, c, d)
    model.Params.OutputFlag = 0
    model.optimize()
    print "Optimal value:", model.ObjVal
    x, y, z, v = model.__data
    for j in x:
        if x[j].X > 0:
            print "%30s: %5g dishes --> %g added to objective" % (j, x[j].X, y[j].X)
    print "amount spent:", v.X
```

```
    print "amount of nutrients:"
    for i in z:
        print "%30s: %5g" % (i, z[i].X)
```

The data is specified in lines 1 through 43. In lines 45 to 58, we solve this problem for different values of the maximum calorie intake, from infinity (i.e., no upper bound on calories) down to 2500. We encourage the reader to use Python/Gurobi to solve this problem, and check that the variety of dishes allowed decreases when the calorie intake is reduced. Interestingly, the amount spent does not vary monotonously: among those values of the calorie intake, the minimum price is for a maximum of calories of 3500 (see also Appendix dietinput).

[source code]

CHAPTER 3

Facility location problems

Todo: Adapt figures, check maths Computational experiment comparing formulations Adapt kmedian — seems to be still gurobi version

---

To import SCIP in python, do:

```python
from pyscipopt import Model, quicksum, multidict
```

We will deal here with facility location, which is a classical optimization problem for determining the sites for factories and warehouses. A typical *facility location problem* consists of choosing the best among potential sites, subject to constraints requiring that demands at several points must be serviced by the established facilities. The objective of the problem is to select facility sites in order to minimize costs; these typically include a part which is proportional to the sum of the distances from the demand points to the servicing facilities, in addition to costs of opening them at the chosen sites. The facilities may or may not have limited capacities for servicing, which classifies the problems into capacited and uncapacited variants. We will analyze several formulations; it is not straightforward to determine which are good and which are bad, but we will provide some tips for helping on this.

The structure of this chapter is the following. In Section *Capacitated facility location problem*, we consider the capacity constrained facility location problem, which will be used to explain the main points of a program in SCIP/Python for solving it. In Section *Weak and strong formulations*, we discuss the quality of different formulations. In Section *The k-Median Problem*, we will present a type of facility location problem that minimizes the sum of the distance to the nearest facility, where the number of facilities is fixed to k: the k-median problem In Section *The k-Center Problem*, we consider a type of facility location problems where the *maximum* value of the distance from a customer to one of the k open facilities is to be minimized. Thus, in this problem we want to find the minimum of maximum value. This is often a tough problem, hard to tackle with a mathematical optimization solver; we will describe some workarounds.

3.1 Capacitated facility location problem

The *capacitated facility location problem* is the basis for many practical optimization problems, where the total demand that each facility may satisfy is limited. Hence, modeling such problem must take into account both demand

satisfaction and capacity constraints.

Let us start with a concrete example. Consider a company with three potential sites for installing its facilities/warehouses and five demand points, as in Section *Transportation Problem*. Each site j has a yearly *activation cost* f_j, i.e., an annual leasing expense that is incurred for using it, independently of the volume it services. This volume is limited to a given maximum amount that may be handled yearly, M_j. Additionally, there is a transportation cost c_{ij} per unit serviced from facility j to the demand point i. These data are shown in Table *Data for the facility location problem: demand, transportation costs, fixed costs, and capacities.*.

Table 1: Data for the facility location problem: demand, transportation costs, fixed costs, and capacities.

Customer i	1	2	3	4	5		
Annual demand d_i	80	270	250	160	180		
Facility j	c_{ij}					f_j	M_j
1	4	5	6	8	10	1000	500
2	6	4	3	5	8	1000	500
3	9	7	4	3	4	1000	500

This situation and its solution are represented in Figure *Facility location*.
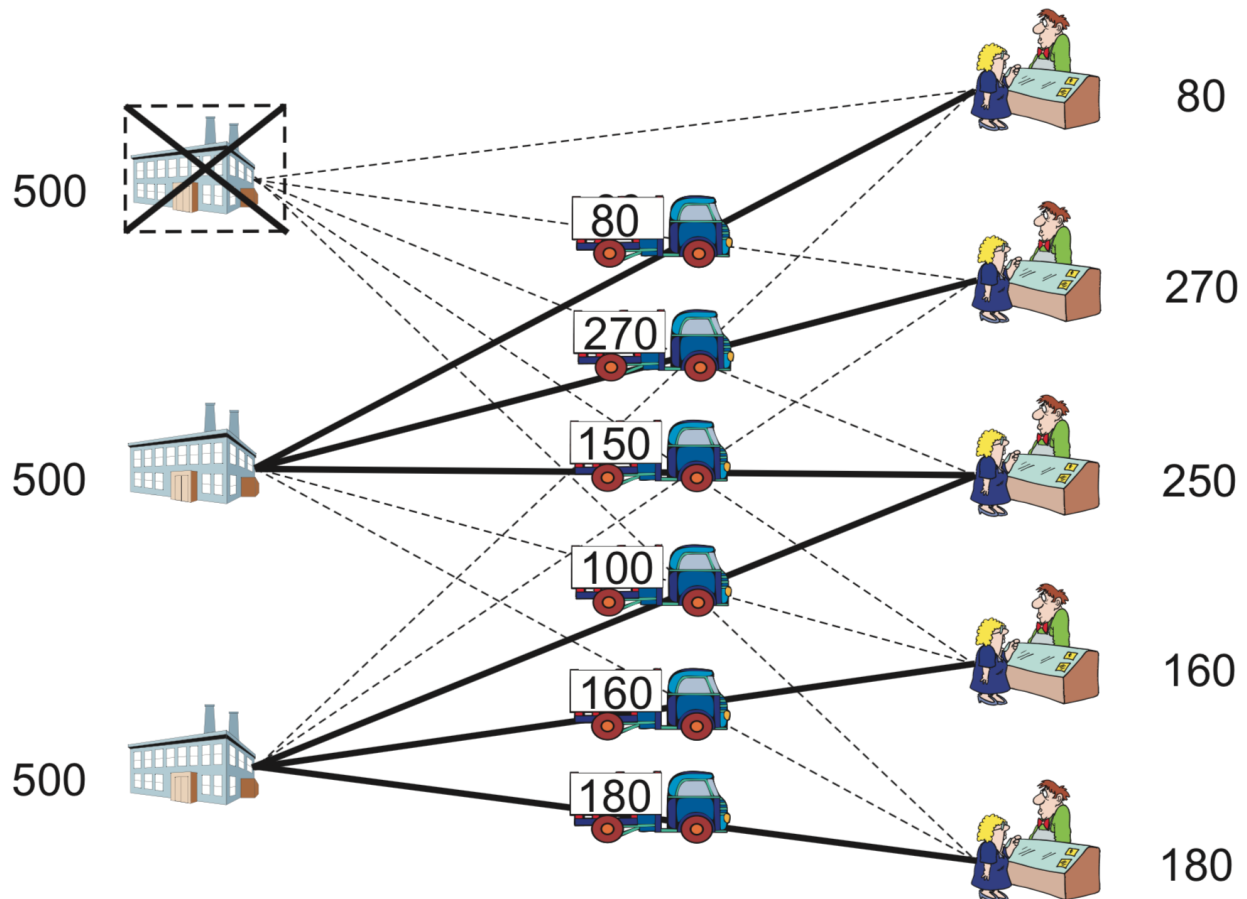
Fig. 1: Facility location

Left: graph representation of an instance of the facility location problem. Suppliers are represented as squares and clients as circles. Right: possible solution, with thick lines representing selected facilities and arcs actually used for transportation.

Let us formulate the above problem as a mathematical optimization model. Consider n customers $i = 1, 2, \ldots, n$

and m sites for facilities $j = 1, 2, \ldots, m$. Define continuous variables $x_{ij} \geq 0$ as the amount serviced from facility j to demand point i, and binary variables $y_j = 1$ if a facility is established at location j, $y_j = 0$ otherwise. An integer-optimization model for the capacitated facility location problem can now be specified as follows:

$$\text{minimize}$$

$$\sum_{j=1}^{m} f_j y_j + \sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij}$$

$$\text{subject to:}$$

$$\sum_{j=1}^{m} x_{ij} = d_i \text{ for } i = 1, \cdots, n$$

$$\sum_{i=1}^{n} x_{ij} \leq M_j y_j \text{ for } j = 1, \cdots, m$$

$$x_{ij} \leq d_i y_j \text{ for } i = 1, \cdots, n; j = 1, \cdots, m$$

$$x_{ij} \geq 0 \text{ for } i = 1, \cdots, n; j = 1, \cdots, m$$

$$y_j \in \{0, 1\} \text{ for } j = 1, \cdots, m$$

The objective of the problem is to minimize the sum of facility activation costs and transportation costs. The first constraints require that each customer's demand must be satisfied. The capacity of each facility j is limited by the second constraints: if facility j is activated, its capacity restriction is observed; if it is not activated, the demand satisfied by j is zero. Third constraints provide variable upper bounds; even though they are redundant, they yield a much tighter linear programming relaxation than the equivalent, weaker formulation without them, as will be discussed in the next section.

The translation of this model to SCIP/Python is straightforward; it is done in the program that follows.

```
def flp(I,J,d,M,f,c):
    model = Model("flp")
    x,y = {},{}
    for j in J:
        y[j] = model.addVar(vtype="B", name="y(%s)"%j)
        for i in I:
            x[i,j] = model.addVar(vtype="C", name="x(%s,%s)"%(i,j))
    for i in I:
        model.addCons(quicksum(x[i,j] for j in J) == d[i], "Demand(%s)"%i)
    for j in M:
        model.addCons(quicksum(x[i,j] for i in I) <= M[j]*y[j], "Capacity(%s)"%i)
    for (i,j) in x:
        model.addCons(x[i,j] <= d[i]*y[j], "Strong(%s,%s)"%(i,j))
    model.setObjective(
        quicksum(f[j]*y[j] for j in J) +
        quicksum(c[i,j]*x[i,j] for i in I for j in J),
        "minimize")
    model.data = x,y
    return model
```

Data for this problem may be specified in Python as follows:

```
1  I, d = multidict({1:80, 2:270, 3:250, 4:160, 5:180})
2  J, M, f = multidict({1:[500,1000], 2:[500,1000], 3:[500,1000]})
3  c = {(1,1):4,   (1,2):6,   (1,3):9,
4       (2,1):5,   (2,2):4,   (2,3):7,
5       (3,1):6,   (3,2):3,   (3,3):4,
6       (4,1):8,   (4,2):5,   (4,3):3,
7       (5,1):10,  (5,2):8,   (5,3):4,
8       }
```

We can now solve the problem:

```
1  model = flp(I, J, d, M, f, c)
2  model.optimize()
3  EPS = 1.e-6
4  x,y = model.__data
5  edges = [(i,j) for (i,j) in x if model.GetVal(x[i,j]) > EPS]
6  facilities = [j for j in y if model.GetVal(y[j]) > EPS]
7  print "Optimal value=", model.GetObjVal()
8  print "Facilities at nodes:", facilities
9  print "Edges:", edges
```

The optimal solution obtained suggests establishing the facilities at Sites 2 and 3 only, as shown in Table *Optimum solution for the facility location problem example.*. This solution incurs minimum total cost of 5610 for servicing all the demands.

Table 2: Optimum solution for the facility location problem example.

Customer	1	2	3	4	5	
Facility	Volume transported					Status
1	0	0	0	0	0	closed
2	80	270	150	0	0	open
3	0	0	100	160	180	open

```
1  Optimal value= 5610.0
2  Facilities at nodes: [2, 3]
3  Edges: [(1, 2), (3, 2), (3, 3), (4, 3), (2, 2), (5, 3)]
```

[source code]

3.2 Weak and strong formulations

Let us consider the facility location problem of the previous section, in a situation where the capacity constraint is not important (any quantity may can be produced at each site). This is referred to as the *uncapacitated facility location problem*. One way of modeling this situation is to set the value of M in the constraint

$$\sum_{i=1}^{n} x_{ij} \leq M_j y_j \quad \text{for } j = 1, \cdots, m$$

as a very large number. Notice that the formulation is correct even if we omit constraints $x_{ij} \leq d_j y_j$, for $i = 1, \cdots, n; j = 1, \cdots, m$. Removing that constraint, the problem may suddenly become very difficult to solve, especially as its size increases; the reason is the *big :math:'M' pitfall*.

Parameter M represents a *large enough* number, usually called **Big M**; it is associated with one of the biggest pitfalls for beginners in mathematical optimization. The idea behind the constraint is to model the fact that "if we do not

activate a warehouse, we cannot transport from there". However, large values for M do disturb the model in practice. Constraints with a "Big M" may be a burden to the mathematical optimization solver, making the model extremely difficult to solve.

Tip: Modeling tip 2

A large number M must be set to a value *as small as possible*

Whenever possible, it is better not to use a large number. If its use is necessary, choose a number that is as small as possible, as long as the formulation is correct. Using large numbers, as $M = 9999999$, is unthinkable, except for very small instances.

In the uncapacitated facility location problem, a correct formulation is to set the capacity M equal to the total amount demanded. However, it is possible to improve the formulation by adding the contraints $x_{ij} \leq d_i y_j$. The natural question here is "what formulation should we use"? Of course, the answer depends on the particular case; but in general *stronger formulations* are recommended. Here, the *strength* of a formulation is not ambiguous: it can be defined in terms of the linear optimization relaxation as follows.

> Definition: Strong and Weak Formulations
>
> Suppose that there are two formulations A and B for the same problem. By excluding the integrality constraints (which force variables to take an integer value), we obtain the *linear optimization relaxation*. Let the feasible region of formulations be P_A and P_B. When the region P_B contains P_A, i.e., $P_A \subset P_B$, formulation A is *stronger* than formulation B (analogously, B is *weaker* than A).

Intuitively, as P_A is tighter than P_B, the upper bound obtained by the relaxation in a maximization problem (or the lower bound in minimization) is closer to the optimum of the integer problem.

$$x_{ij} \leq d_i y_j$$

is stronger than using only constraints

$$\sum_{j=1}^{m} x_{ij} \leq \left(\sum_{i=1}^{n} d_i \right) y_j.$$

To verify it, let P_A be the feasible region using the former constraints and P_B the feasible region when using the latter; observe that the latter constraints are obtained by adding the former, hence $P_A \subseteq P_B$

A truly stronger formulation is either indicated by having $P_A \subset P_B$, or by verifying that solution of the linear optimization relaxation of B is not included in P_A.

As for the question "is it always preferable to use a stronger formulation?", there is not a theoretical answer; distinguishing each case is part of the mathematical modeling art.

Let us try to give some guidance on this. Often, stronger formulations require many more constraints or variables than weaker formulations. In the previous example, the strong formulation requires nm constraints, while the weak requires only n. The time for solving the linear relaxation, which depends on the number of constraints and variables, is likely to be longer in the case of the stronger formulation. Hence, there is a trade-off between shorter times for solving linear optimization relaxations in weaker formulations, and longer computational times for branch-and-bound; indeed, the enumerating tree is likely smaller for stronger formulations. As a guideline, as the size of the enumeration tree grows very rapidly when the scale of the problem increases, stronger formulations are considered more desirable (even if the number of constraints and variables becomes larger).

3.3 The k-Median Problem

There are many variants of the facility location problem; here we will consider the following classic problem.

Median problem

Select a given number of facilities from possible points in a graph, in such a way that the **sum** of the distances from each customer to the closest facility is minimized.

Often, the number of facilities to be selected is predetermined in advance; this number is commonly denoted k (or p), and the corresponding problem is given this symbol as a suffix.

The *k-median problem* is hence a variant of the *uncapacitated facility location problem* and specifically seeks to establish k facilities, without considering fixed costs.

The distance from the customer i to facility j is denoted c_{ij}, the set of customers by $\{1, n\}$, and the set of potential places for facilities by $\{1, m\}$. In the most common situation, it is assumed that facilities and customers share the same set of points. Let us define the following variables:

$$x_{ij} = \begin{cases} 1 & \text{when the demand of customer } i \text{ is met by facility } j \\ 0 & \text{otherwise} \end{cases}$$

$$y_j = \begin{cases} 1 & \text{when facility } j \text{ is open} \\ 0 & \text{otherwise} \end{cases}$$

Using the symbols and variables defined above, the k-median problem can be formulated as an integer-optimization model.

$$\text{minimize}$$

$$\sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij}$$

$$\text{subject to:}$$

$$\sum_{j=1}^{m} x_{ij} = 1 \text{ for } i = 1, \cdots, n$$

$$\sum_{j=1}^{m} y_j = k$$

$$x_{ij} \leq y_j \text{ for } i = 1, \cdots, n; j = 1, \cdots, m$$

$$x_{ij} \in \{0, 1\} \text{ for } i = 1, \cdots, n; j = 1, \cdots, m$$

$$y_j \in \{0, 1\} \text{ for } j = 1, \cdots, m$$

$$\sum_{i=1}^{n} x_{ij} \leq y_j, \text{ for } j = 1, \cdots, m.$$

However, as explained in the previous section, this constraints will lead to much worse values in the linear relaxation, and should be avoided.

The objective function minimizes the total cost of servicing all demand points.

As an illustration of this problem, consider the solution obtained for a graph with 200 vertices placed randomly in the two-dimensional unit box, represented in Figure *k-median*. Costs are given by the Euclidean distance between the points, and each of the vertices is a potential location for a facility.

The following Python program shows how to create a model for this problem, implemented as a function that takes the problem's data as parameters, and returns variable objects x and y as attribute data of the model object.
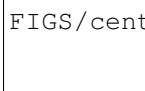
FIGS/median.pdf

Fig. 2: k-median

Solution of the k-median model to a instance with 200 random vertices on the plane, and $k = 20$.

3.4 The k-Center Problem

The k-median problem, considered above, has an interesting variant called the *k-center problem*.

Center problem

Select a given number of facilities from possible points in a graph, in such a way that the **maximum value** of a distance from a customer to the closest facility is minimized.

FIGS/center.pdf

Fig. 3: k-center

Solution of k-center to a random instance with 100 nodes and $k = 10$.

Essentially, the problem seeks an assignment of facilities to a subset of vertices in the graph so that each customer's vertex is "close" to some facility. As in the k-median problem, the number of facilities to be selected is predetermined in advance, and fixed to a value k. Input data for the k-center problem is also the distance c_{ij} from the customer i to facility j, the set of customers by $\{1, n\}$, and the set of potential places for facilities by $\{1, m\}$; often, again, it is assumed that facilities and customers share the same set of points. Variables have the same meaning:

$$x_{ij} = \begin{cases} 1 & \text{when the demand of customer } i \text{ is met by facility } j \\ 0 & \text{otherwise} \end{cases}$$

$$y_j = \begin{cases} 1 & \text{when facility } j \text{ is open} \\ 0 & \text{otherwise} \end{cases}$$

In addition, we introduce a continuous variable z to represent the distance/cost for the customer which is most distant from an established facility. Using these symbols and variables, the k-center problem can be formulated as the

following mixed-integer optimization problem.

$$\text{minimize}$$
$$z$$
$$\text{subject to:}$$

$$\sum_{j=1}^{m} x_{ij} = 1 \text{ for } i = 1, \cdots, n$$

$$\sum_{j=1}^{m} y_j = k$$

$$x_{ij} \leq y_j \text{ for } i = 1, \cdots, n; j = 1, \cdots, m$$

$$\sum_{j=1}^{m} c_{ij} x_{ij} \leq z \text{ for } i = 1, \cdots, n$$

$$x_{ij} \in \{0, 1\} \text{ for } i = 1, \cdots, n; j = 1, \cdots, m$$

$$y_j \in \{0, 1\} \text{ for } j = 1, \cdots, m$$

The first constraints require that each customer i is assigned to exactly one facility. The second constraints ensure that exactly k facilities are opened. The third constraints force facility j to be activated some customer i is assigned to j. The fourth constraints determine z to take on at least the value weight c_{ij}, for all facilities j and customers i assigned to j. A version of these constraints which may be more natural, but which is much weaker, is to specify instead $c_{ij} x_{ij} \leq z$, for $i = 1, \cdots, n; j = 1, \cdots, m$. Intuitively, we can reason that in the strong formulation, as we are adding more terms in the left-hand side, the corresponding feasible region is tighter. The objective function represents the distance that the customer which is served by the most distant facility, as calculated by the third constraints, must travel.

The objective of the k-center problem is a classic case of minimizing a maximum value, also called a *min-max* objective; this is a type of problems for which mathematical optimization solvers are typically weak.

The following Python program shows how to create a model for the k-center problem; it is very similar to the program used to solve the k-median problem.

Note: Margin seminar 4

Techniques in linear optimization

As illustrated in the text, the problem of "minimization of the maximum value" can be reduced to a standard linear optimization, by adding a new variable and a few modifications to the model. Here, we will describe with more detail these techniques in linear optimization. Let us give a simple example. Assume that we want to minimize the maximum of two linear expressions, $3x_1 + 4x_2$ and $2x_1 + 7x_2$. For this, we introduce a new variable z and the constraints:

$$3x_1 + 4x_2 \leq z$$
$$2x_1 + 7x_2 \leq z$$

With the addition of these linear constraints, minimizing z will correctly model the minimization of the maximum of those two expressions.

A related topic that often arises in practice is the minimization of the absolute value $|x|$ of a real variable x, which is a nonlinear expression. We can linearize it by means of two non-negative variables y and z. Firstly, we compute the

---

value of x in terms of y and z, though the constraint $x = y - z$. Now, since $y \geq 0$ and $z \geq 0$, we can represent a positive x with $z = 0$ and $y > 0$, and a negative x with $y = 0$ and $z > 0$. Then, $|x|$ can be written as $x + y$. In other words, all occurrences of x in the formulation are replaced by $y - z$, and $|x|$ in the objective function is replaced by $y + z$.

It is also possible to handle an absolute value by simply adding one variable z and then imposing $z \geq x$ and $z \geq -x$. When minimizing z, if x is non-negative, then the contraint $z \geq x$ is active; otherwise, $z \geq -x$ is binding. Variable z replaces $|x|$ in the objective function.

Tip: Modeling tip 3

An objective function that minimizes a maximum value should be avoided, if possible.

When an integer optimization problem is being solved by the branch-and-bound method, if the objective function minimizes the maximum value of a set of variables (or maximizes their minimum value), there is a tendency to have large values for the difference between the lower bound and the upper bound (the so-called *duality gap*). In this situation, either the time for solving the problem becomes very large, or, if branch-and-bound is interrupted, the solution obtained (the *incumbent* solution) is rather poor. Therefore, when modeling real problems, it is preferable to avoid such formulations, if possible.

3.4.1 The k-Cover Problem

The k-center problem, considered above, has an interesting variant which allows us to avoid the min-max objective, based on the so-called the k-*cover problem*. In the following, we utilize the structure of k-center in a process for solving it making use of binary search.

Consider the graph $G_\theta = (V, E_\theta)$ consisting of the set of edges whose distances from a customer to a facility which do not exceed a threshold value θ, i.e., edges $E_\theta = \{\{i, j\} \in E : c_{ij} \leq \theta\}$. Given a subset $S \subseteq V$ of the vertex set, S is called a *cover* if every vertex $i \in V$ is adjacent to at least one of the vertices in S. We will use the fact that the optimum value of the k-center problem is less than or equal to θ if there exists a cover with cardinality $|S| = k$ on graph G_θ.

Define $y_j = 1$ if a facility is opened at j (meaning that the vertex j is in the subset S), and $y_j = 0$ otherwise. Furthermore, we introduce another variable:

$$z_i = \begin{cases} 1 & \text{vertex } i \text{ is adjacent to no vertex in } S \text{ (it is not covered)}, \\ 0 & \text{otherwise.} \end{cases}$$

Let us denote by $[a_{ij}]$ the incidence matrix of G_θ, whose element a_{ij} is equal to 1 if vertices i and j are adjacent, and is equal to 0 otherwise. Now we need to determine whether the graph G_θ has a cover $|S| = k$; we can do that by

solving the following integer-optimization model, called the k-cover problem on G_θ:

$$\text{minimize } \sum_{i=1}^{n} z_i$$

$$\text{subject to } \sum_{j=1}^{m} a_{ij} y_j + z_i \geq 1$$

$$\text{for } i = 1, \cdots, n$$

$$\sum_{j=1}^{m} y_j = k$$

$$z_i \in \{0, 1\}$$

$$\text{for } i = 1, \cdots, n$$

$$y_j \in \{0, 1\}$$

$$\text{for } j = 1, \cdots, m.$$

Notice that the adjacency matrix is built upon a given value of distance θ, based on which is computed the set of facilities that may service each of the customers within that distance. For a given value of θ there are two possibilities: either the optimal objective value of the previous optimization problem is zero (meaning that k facilities were indeed enough for covering all the customers withing distance θ), of it may be greater that zero (meaning that there is at least one $z_i > 0$, and thus a customer could not be serviced from any of the k open facilities). In the former case, we attempt to reduce θ, and check if all customers remain covered; in the latter, θ is increased. This process is repeated until the bounds for θ are close enough, in a process called *binary search*.

$\text{LB} \leftarrow 0$
$\text{UB} \leftarrow \text{maximum value of the distance}$
while $UB - LB > \epsilon$ **do**
  $\theta \leftarrow (UB + LB)/2$
  solve k-cover problem on G_θ
  if *optimum is 0* **then**
    | $\text{UB} \leftarrow \theta$
  else
    $\lfloor \text{LB} \leftarrow \theta$

Fig. 4: Binary search method for solving the k-center problem.

An illustration is provided in Figure *Search for optimum \theta with binary search.*.

Using these ideas, we can get the optimal value of the k-center problem by the following binary search method.

In Table *Solution and runtimes for a random instance.* we provide some numbers, for having just an idea of the order of magnitude of the computational times involved in these problems. They concern an instance with 200 vertices randomly distributed in the plane and $k = 5$.
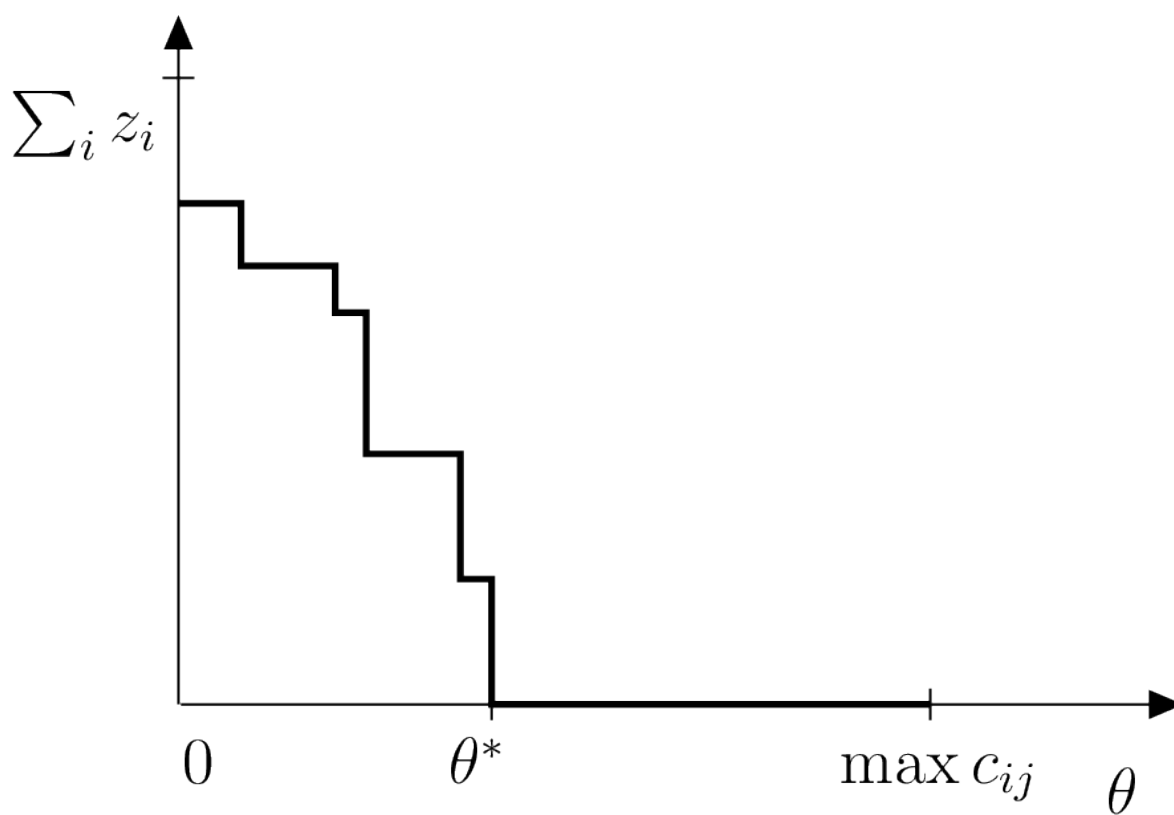
Fig. 5: Search for optimum θ with binary search.

Table 3: Solution and runtimes for a random instance.

	max distance	CPU time (s)
k-median	0.3854	3.5
k-center	0.3062	645.5
k-cover + binary search	0.3062	2.5

As we can see, k-cover within binary search method allows the computation of the k-center solution in a time comparable to that required for the k-median solution.

From a practical point of view, the k-center solution is usually preferable to that of k-median. Indeed, the longest time required for servicing a customer is frequently an important criterion to be considered by a company, and this may be prohibitively large on the k-median solution.

Bin packing and cutting stock problems

Todo: Adapt column generation to work with SCIP

This chapter deals with two classic problem: the bin packing problem and the cutting stock problem. Let us start with some definitions and examples.

You are the person in charge of packing in a large company. Your job is to skillfully pack items of various weights in a box with a predetermined capacity; your aim is to use as few boxes as possible. Each of the items has a known weights, and the upper limit of the contents that can be packed in a box is 9 kg. The weight list of items to pack is given in Table *Weights of items to be packed in bins of size 9.*. In addition, the items you are dealing with your company are heavy; there is no concern with the volume they occupy. So, how should these items be packed?

Table 1: Weights of items to be packed in bins of size 9.

Weights of items to be packed
6, 6, 5, 5, 5, 4, 4, 4, 4, 2, 2, 2, 2, 3, 3, 7, 7, 5, 5, 8, 8, 4, 4, 5

This is an example of a problem called the *bin packing problem*. It can be described mathematically as follows.

> Bin packing problem
>
> There are n items to be packed and an infinite number of available bins of size B. The sizes $0 \leq s_i \leq B$ of individual items are assumed to be known. The problem is to determine how to pack these n items in bins of size B so that the number of required bins is minimum.

A related problem is the *cutting stock problem*, which is defined as follows.

You are the person in charge of cutting in a large company producing rolls of paper. Your job is to skillfully cut the large rolls produced in a standard size into smaller rolls, with sizes demanded by the customers. It is not always possible to fully use every roll; sometimes, it is necessary to create leftovers, called *trim loss*. In this case, your aim is to use as few rolls as possible; in other words, to minimize the trim loss created. The width of the large rolls is 9 meters, and there are customers' orders for seven different sizes, as detailed in Table *Orders for different roll lengths.*. So, how should the large rolls be cut?
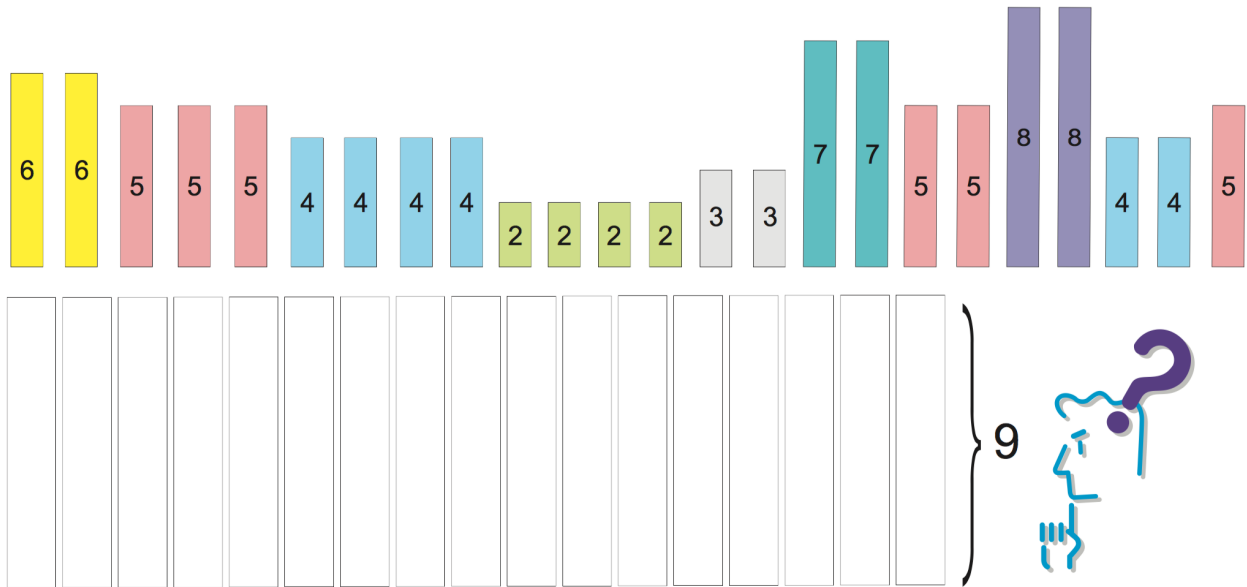
Fig. 1: Bin packing instance
Item weights and bin capacity for an instance of the bin packing problem.

Table 2: Orders for different roll lengths.

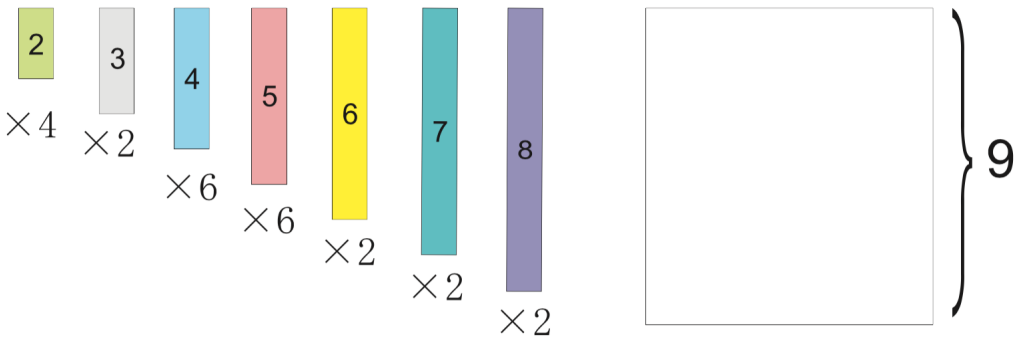| Length	Number of rolls
2 m	4
3 m	2
4 m	6
5 m	6
6 m	2
7 m	2
8 m	2

Fig. 2: Cutting stock instance
Item lengths and roll size for an instance of the cutting stock problem.

The cutting stock problem can be described mathematically as follows.

Cutting stock problem

There are orders for $i = 1, \ldots, m$ different widths, with quantity q_i ordered for width $0 \leq w_i \leq B$, to be cut from standard rolls with width B. The problem is to find a way to fulfill the orders while using the

minimum number of rolls.

The bin packing and the cutting stock problems may at first glance appear to be different, but in fact it is the same problem. This can be seen with the examples above, which actually refer to the same situation. If find a the solution using a formulation for one of the problems, it will also be a solution for the other case. As the problems are equivalent, deciding which to solve depends on the situation.

This chapter is structured as follows. Section *The Bin Packing Problem* presents a straightforward formulation for the bin packing problem. Section *Column generation method for the cutting stock problem* describes the column generation method for the cutting stock problem. For both cases, we show how to obtain a solution with SCIP/Python.

4.1 The Bin Packing Problem

In the bin packing problem, it is assumed that an upper bound U of the number of bins is given. In a simple formulation, a variable X indicates whether an item is packed in a given bin, and a variable Y specifies if a bin is used in the solution or not.

$$X_{ij} = \begin{cases} 1 & \text{if item } i \text{ is packed in bin } j \\ 0 & \text{otherwise} \end{cases}$$

$$Y_j = \begin{cases} 1 & \text{if bin } j \text{ is used} \\ 0 & \text{otherwise} \end{cases}$$

Using these variables, the bin packing problem can be described as an integer optimization problem.

$$\text{minimize}$$

$$\sum_{j=1}^{U} Y_j$$

$$\text{subject to:}$$

$$\sum_{j=1}^{U} X_{ij} = 1 \text{ for } i = 1, \cdots, n$$

$$\sum_{i=1}^{n} s_i X_{ij} \leq B Y_j \text{ for } j = 1, \cdots, U$$

$$X_{ij} \leq Y_j \text{ for } i = 1, \cdots, n; j = 1, \cdots, U$$

$$X_{ij} \in \{0, 1\} \text{ for } i = 1, \cdots, n; j = 1, \cdots, U$$

$$Y_j \in \{0, 1\} \text{ for } j = 1, \cdots, U$$

The objective function is the minimization of the number of bins used. The first constraints force the placement of each item in one bin. The second constraints represent the upper limit on the bins contents, as well as the fact that items cannot be packed in a bin that is not in use. The third constraints provide an enhanced formulation, indicating that if a bottle is not used ($Y_j = 0$), items cannot be placed there ($X_{ij} = 0$). Without these inequalities it is possible to find an optimum solution; however, as mentioned in Section *Weak and strong formulations*, a speedup can be expected by the addition of these stronger constraints.

Let us see how this formulation can be written in SCIP/Python. First of all, we will prepare a function to generate the example's data.

```
def BinPackingExample():
    B = 9
    w = [2,3,4,5,6,7,8]
    q = [4,2,6,6,2,2,2]
    s=[]
    for j in range(len(w)):
        for i in range(q[j]):
            s.append(w[j])
    return s,B
```

Here, the data is prepared as for a cutting stock problem (width of rolls B, number of orders q and width orders w) and is converted to the bin packing data (list s of sizes of items, bin size B).

Next, we need to calculate the upper limit U of the number of bins. The bin packing problem has been for a long time a field for the development of heuristics. *Heuristics* are procedures for obtaining a solution based on rules that do not guarantee that the optimum will be reached. A well-known heuristics for this problem is *first-fit decreasing (FFD)*, which consists of arranging the items in non-increasing order of their size, and then for each item try inserting it in the first open bin where it fits; if no such bin exists, then open a new bin and insert the item there. Here is a simple implementation in Python.

```
def FFD(s, B):
    remain = [B]
    sol = [[]]
    for item in sorted(s, reverse=True):
        for j,free in enumerate(remain):
            if free >= item:
                remain[j] -= item
                sol[j].append(item)
                break
        else:
            sol.append([item])
            remain.append(B-item)
    return sol
```

In line 2, `remain` is a list to store the space remaining in bins currently in use, which is initialized to have only one bin of size B. The solution is stored in a list of lists, initialized in line 3 as a list containing an empty list; this represents a solution consisting of an empty bin. Line 4 starts a `for` loop, where items are taken out in descending order of their size. Here, `sorted` is a Python function for generating the contents of a list in order; with the optional parameter `reverse=True`, the order is reversed. Line 5 starts an iteration over the bins currenlty in use, where `free` is assigned to the space available in `j` th bin; if there is space available the current item is packed in `j`. If the current items doesn't fit in any bin, a new bin is created and the item is packed there. Here, `enumerate` is a Python function returning tuples (`index,value`) with the index for each element (`value`) in a sequence. The return value is list `sol` representing the solution found, and hence its length is an upper bound U of the number of bins.

We now have the tools for implementing a function for solving the bin packing problem.

```
def bpp(s,B):
    n = len(s)
    U = len(FFD(s,B))
    model = Model("bpp")
    x,y = {},{}
    for i in range(n):
        for j in range(U):
            x[i,j] = model.addVar(vtype="B", name="x(%s,%s)"%(i,j))
    for j in range(U):
        y[j] = model.addVar(vtype="B", name="y(%s)"%j)
    for i in range(n):
```

```
        model.addCons(quicksum(x[i,j] for j in range(U)) == 1, "Assign(%s)"%i)
    for j in range(U):
        model.addCons(quicksum(s[i]*x[i,j] for i in range(n)) <= B*y[j], "Capac(%s)"
→%j)
    for j in range(U):
        for i in range(n):
            model.addCons(x[i,j] <= y[j], "Strong(%s,%s)"%(i,j))
    model.setObjective(quicksum(y[j] for j in range(U)), "minimize")
    model.data = x,y
    return model
```

This model can be used to compute a list with the items that should be placed in each bin, as follows:

```
def solveBinPacking(s,B):
    n = len(s)
    U = len(FFD(s,B))
    model = bpp(s,B)
    x,y = model.data
    model.optimize()
    bins = [[] for i in range(U)]
    for (i,j) in x:
        if model.getVal(x[i,j]) > .5:
            bins[j].append(s[i])
    for i in range(bins.count([])):
        bins.remove([])
    for b in bins:
        b.sort()
    bins.sort()
    return bins
```

Using the program above it is possible to obtain a solution with objective value (the number of bins) 13. This solution is shown in Figure *Solution*.
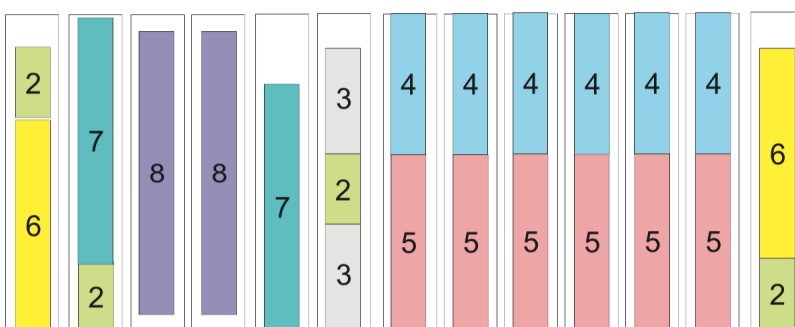
Fig. 3: Solution
Solution obtained for the bin packing example.

4.2 Column generation method for the cutting stock problem

Here, we will introduce the column generation method for the cutting stock problem proposed by Gilmore-Gomory [GG61] [GG63].

When representing a linear optimization problem by means of a matrix, the left-hand side of the constraints' coefficients, there is a correspondence of each row of the matrix to a constraint, and a correspondence between each column

of the matrix and a variable. Hence, constraints are often referred to as *rows*, and variables are also called *columns*.

In the *column generation* method only a (usually small) subset of the variables is used initially. The method sequentially addes columns (i.e., variables), using information given by the dual variables for finding the approriate variable to add.

Let us try to explain how it works by means of the example provided in *Orders for different roll lengths.*. There are many was of cutting the base roll into width requested in the order; let us consider a valid *cutting pattern* a set of widths whose sum does not exceed the roll's length ($B = 9$ meters). First, we will generate simple patterns, each composed only of one ordered width repeated as many times as it fits in roll length. For order j of width w_j, the number of times it can be cut from the base roll is B divided by w_j rounded down. Let us represent a pattern as a vector (in the programs, as a list) with the number of times each width is cut. For example, the width $w_1 = 2$ of order 1 was 2 meters, and will be cut $\lfloor B/w_1 \rfloor = \lfloor 9/2 \rfloor = 4$ times in case of cutting only the width of order 1; this cutting pattern can be represented as $(4, 0, 0, 0, 0, 0, 0)$. Repeating this for the other orders allows us to generate an initial set of cutting patterns. A Python program for generating a list *t* of all the initial cutting patterns can be written as follows.

```
t = []
m = len(w)
for i in range(m):
    pat = [0]*m
    pat[i] = int(B/w[i])
    t.append(pat)
```

The initial set of cutting patterns is the following (also represented on the left side of Figure *Solution*).

```
[4,0,0,0,0,0,0]
[0,3,0,0,0,0,0]
[0,0,2,0,0,0,0]
[0,0,0,1,0,0,0]
[0,0,0,0,1,0,0]
[0,0,0,0,0,1,0]
[0,0,0,0,0,0,1]
```
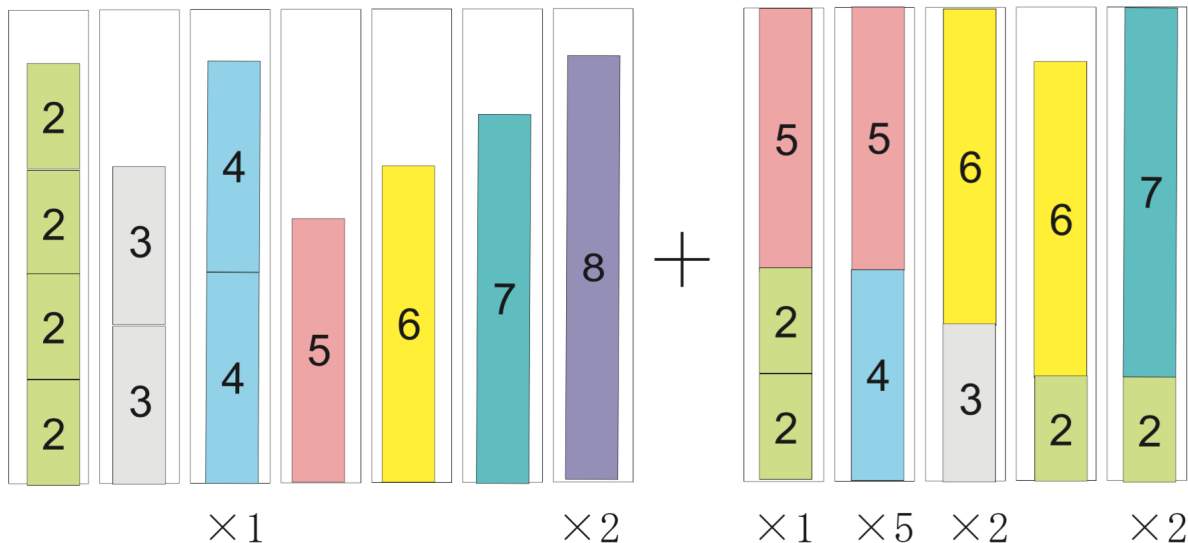

Fig. 4: Solution
Solution obtained for the cutting stock example.

If we define integer variable x_i for representing the number of times to use cutting pattern i, considering only the initial cutting patterns, an integer optimization problem problem for finding the minimum number of rolls to meet all

the orders is the following.

$$\text{minimize}$$

$$x_1 + x_2 +$$
$$x_3 + x_4 +$$
$$x_5 + x_6 +$$
$$x_7$$

$$4x_1$$

$$\geq 4$$

$$3x_2$$

$$\geq 2$$

$$2x_3$$

$$\geq 6$$

$$x_4$$

$$\geq 6$$

$$x_5$$
$$\geq 2$$

$$x_6$$
$$\geq 2$$

$$x_7 \geq 2$$

$$x_1, x_2,$$
$$x_3, x_4,$$
$$x_5, x_6,$$
$$x_7 \geq$$

$$0, \text{integer}$$

If we now solve this linear optimization problem by relaxing the integer constraints, the optimum is $16\frac{2}{3}$ and an optimal solution is $x = (1, 2/3, 3, 6, 2, 2, 2)$. We can also obtain, for each constraint, the corresponding optimal dual variable: $\lambda = (1/4, 1/3, 1/2, 1, 1, 1, 1)$. These quantities can be interpreted as the *value* of each order in terms of the base roll; for example, $\lambda_1 = 1/4$ can be interpreted as "order 1 is worthy 1/4 of a roll". (See margin seminar *Duality* for an interpretation of the optimal values of dual variables.)

We can observe that in the first cutting pattern a lot of waste has been generated. In order to obtain a more efficient cutting strategy, the base roll must be cut with different, high-value orders such that its width is not exceeded.

Using integer variables y_j to represent how many pieces of order j should be cut, finding the cutting pattern with the largest value can be formulated as the following integer optimization problem.

$$\text{minimize}$$
$$\frac{1}{4}y_1 + \frac{1}{3}y_2 +$$
$$\frac{1}{2}y_3 + y_4 +$$
$$y_5 + y_6 +$$
$$y_7$$

$$2y_1 + 3y_2 +$$
$$4y_3 + 5y_4 +$$
$$6y_5 + 7y_6 +$$
$$8y_7 \leq 9$$

$$y_1, y_2,$$
$$y_3, y_4,$$
$$y_5, y_6,$$
$$y_7 \geq$$
$$0, \text{integer}$$

This is called the *integer knapsack problem*, a variant of the problem presented in Section knapsack where the variables are non-negative integers. Even though the integer knapsack problem is known to be NP-hard, optimal solutions can be obtained relatively easily with SCIP. For the instance above, the optimum is 1.5, and the corresponding solution is $y = (2, 0, 0, 1, 0, 0, 0)$. This indicates that a pattern with the value of 1.5 units of the base roll can be obtained by cutting a roll in two pieces of order 1 and one piece of order 4.

The reduced cost of this new column is $1(2\lambda_1 + \lambda_4) = 0.5$; this indicates that by adding a column with this cutting pattern it is possible to obtain a benefit of 0.5 base rolls. (See Margin Seminar reducedcosts for the definition of reduced costs.)

We will now add this column and solve the linear relaxation problem again. Let the variable x_8 indicate the number of times to use the new cutting pattern; the linear relaxation of the problem of finding a minimum number of rolls so

as to satisfy the orders is as follows.

$$\text{minimize}$$

$$x_1 + x_2$$
$$+x_3 + x_4$$
$$+x_5 + x_6$$
$$+x_7 + x_8$$

$$4x_1$$

$$+ 2x_8$$
$$\geq 4$$

$$3x_2$$

$$\geq 2$$

$$2x_3$$

$$\geq 6$$

$$x_4$$

$$+ x_8$$
$$\geq 6$$

$$x_5$$

$$\geq 2$$

$$x_6$$

$$\geq 2$$

$$x_7$$
$$\geq 2$$

In this example, after adding five new patterns the reduced cost of the new column found by solving the knapsack problem is not negative, and the column generation procedure stops. As at the end we want an integer solution, we add the integrality constraints to the last linear optimization problem created in the procedure. Solving this problem, we determine a solution using 13 rolls; the final set of patterns, as well as the count of each of them in the final solution, are shown in Figure *Solution*. Notice that, in general, there is no guarantee that all the relevant patterns had been added, and hence this solution may not be optimal for the original problem (though in this particular example we can show that the solution is optimal, as the minimum number of bins required is $\lceil \sum_{i=1}^{m} q_i w_1 / B \rceil = \lceil 12 \frac{2}{9} \rceil = 13$).

When the number of variable in a model is huge, the method of column generation is effective. It is summarized below.

Tip: Modeling tip 4

Use the column generation method when the number of variables is extremely large.

For many practical problems (as the cutting stock problem above), a solution approach is to generate possible patterns and let an optimization model select the relevant patterns.

The number of possible patterns may be enormous. Rather than enumerating all the possibilities, it is effective to solve an appropriate subproblem (a knapsack problem, in the case of the cutting stock problem) to generate only relevant patterns.

After defining the subproblem, the complicated part is the exchange of information between these two problems, in particular dual information. However (as shown below), this is relatively simple to program with SCIP/Python.

Before describing the program used for solving the cutting stock problem, let us introduce a formulation and the column generation method in a general form. Let the k-th cutting pattern of base roll width B into some of the m width ordered be denoted as a vector $(t_1^k, t_2^k, \ldots, t_m^k)$. Here, t_i^k represents the number of times the width of order i is cut out in the k-th cutting pattern. For a pattern $(t_1^k, t_2^k, \ldots, t_m^k)$ (which is a packing in the bin packing problem) to be feasible, it must satisfy:

$$\sum_{k=1}^{m} t_i^k \leq B$$

Let us denote by K the current number of cutting patterns. The cutting stock problem is to decide how to cut a total number of ordered width j at least q_j times, from all the available cutting patterns, so that the total number of base rolls used is minimized.

$$\text{minimize}$$

$$\sum_{k=1}^{K} x_k$$

$$\text{subject to:}$$

$$\sum_{k=1}^{K} t_i^k x_k \geq q_i \quad \text{for } i = 1, \ldots, m$$

$$x_k \geq 0, \text{integer} \quad \text{for } k = 1, \ldots, K.$$

This is called the *master problem*. Consider the linear optimization relaxation of the master problem, and the optimal dual variable vector λ. Using λ as the value assigned to each width i, the next problem is to find a feasible pattern $(y_1, y_2, \ldots, y_m)$ that maximizes the value of the selected widths. This is an integer knapsack problem; its solution

will be used as an additional pattern in the master problem.

$$\text{maximize}$$

$$\sum_{i=1}^{m} \lambda_i y_i$$

$$\text{subject to:}$$

$$\sum_{i=1}^{m} w_i y_i \leq B$$

$$y_i \geq 0, \text{integer} \quad \text{for } i = 1, \ldots, m.$$

Based on the notation introduced above, we will describe how to implement column generation for the cutting stock problem using SCIP/Python. At first, we will create the model for the master problem, i.e., an integer optimization model for finding the minimum number of base rolls, with the currently available patterns, such that all the orders are satisfied. Generated patterns are stored in the list `t`, where `t[k][[i]` holds the number of times width `i` is used in pattern `k`; that number multiplied by the number of times pattern `k` is used `x[k]` must satisfy the ordered number of width `i`, `q[i]`. The objective is to minimize the number of base rolls needed, which is given by the sum of `x[k]` for all patterns `k`.

```
K = len(t)
master = Model("master LP")
x = {}
for k in range(K):
    x[k] = master.addVar(vtype="I", name="x(%s)"%k)
orders = {}
for i in range(m):
    orders[i] = master.addCons(
        quicksum(t[k][i]*x[k] for k in range(K)) >= q[i])
master.setObjective(quicksum(x[k] for k in range(K)), "minimize")
```

After generating an initial set of `K` patterns, the master problem is defined and an variable for each pattern is added to the model. The main loop of the column generation method starts by solving the relaxation of the master problem, and assigning its dual variables to list `lambda_`. Then, the knapsack subproblem is defined. The coefficients at the objective are the values of the dual variables, and the knapsack constraint is the width `w[i]` of ordered width `i` multiplied by the number of times that width is used in the pattern, `y[i]`. If the optimum for the subproblem less than 1, then the reduced costs have become all non-negative, and no more patterns are generated. Otherwise, the new pattern is added to list `t`, and a new column for this pattern is added to the master problem.

```
while True:
    relax = master.relax()
    relax.optimize()
    pi = [c.Pi for c in relax.getConstrs()]
    knapsack = Model("KP")
    knapsack.ModelSense=-1
    y = {}
    for i in range(m):
        y[i] = knapsack.addVar(ub=q[i], vtype="I", name="y[%d]"%i)
    knapsack.update()
    knapsack.addConstr(quicksum(w[i]*y[i] for i in range(m)) <= B, "width")
    knapsack.setObjective(quicksum(pi[i]*y[i] for i in range(m)), GRB.MAXIMIZE)
    knapsack.optimize()
    if knapsack.ObjVal < 1+EPS:
        break
    pat = [int(y[i].X+0.5) for i in y]
```

(continues on next page)

```python
        t.append(pat)
        col = Column()
        for i in range(m):
            if t[K][i] > 0:
                col.addTerms(t[K][i], orders[i])
        x[K] = master.addVar(obj=1, vtype="I", name="x[%d]"%K, column=col)
        master.update()
        K += 1


master.optimize()
rolls = []
for k in x:
    for j in range(int(x[k].X + .5)):
        rolls.append(sorted([w[i] for i in range(m) if t[k][i]>0 for j in
→range(t[k][i])]))
rolls.sort()
return rolls
```

After finishing the column generation cycle, the (integer) model with all added patterns is solved.

Graph problems

Todo: Adapt everything: figures, maths, . . .

In this chapter we will present models for three optimization problems with a combinatorial structure (graph partitioning problem, maximum stable set problem, graph coloring problem) and try to solve them with SCIP/Python. All the models dealt with here are based on the definition of a graph. A graph is an abstract concept, a construction derived from vertices and edges linking two vertices, but many of the practical optimization problem can be defined naturally by means of graphs.

The roadmap for this chapter is the following. Section gpp deals with the basic notions of graph theory and with the graph partitioning problem, describing a method for dealing with a quadratic objective function by linearizing it. Section ssp presents the maximum stable set problem. Section gcp describes the graph coloring problem, proposing an improved model for avoiding symmetry in the solution space.

5.1 Graph partitioning problem

Consider the following scenario.

Six friends are deciding how to split for forming two teams of mini-soccer (Figure *Graph partitioning problem*). Of course, in order to be fair, each team must have the same number of persons — three, in this case. However, having good friends in separate teams should be avoided as much as possible. So, how should the group of persons be divided into two teams?

The case above is an example of a combinatorial optimization problem called the *graph partitioning problem*. Actually, rather than creating football teams, this NP-hard problem has a number of serious applications, including VLSI (very-large-scale integration) design.

This real problem is easy to understand using the concept of "graph". A graph is an abstract object composed of *vertices* and *edges*; an edge is a link between two vertices. Graphs are very useful tools to unambiguously represent many real problems. As an example, let us represent a friendship relationship with a graph.

You have six friends. First of all, represent each of these friends by a circle; in graph theory, these circles are called *vertices* (also called *nodes* or *points*). As always in life, some of these fellows have a good relationship between them,
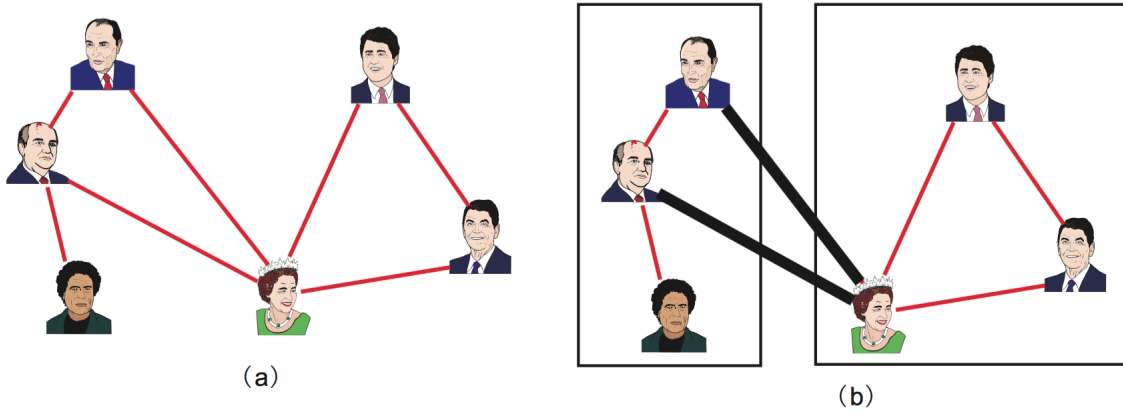
Fig. 1: Graph partitioning problem

(a) Graph where an edge between two persons indicates that they are good friends.

(b) Solution where the number of good friends in (equally divided) different teams is minimum. Pairs of good friends belonging to different teams are represented by a thick line — there are two, in this case. Hence, the objective value for equal division is 2.

whereas others have a bad relationship. In order to organize these complicated relashionships, you connect with a line each pair of your friends which are in good terms with each other. In graph theory, such a line is called an *edge* (also called *arc* or *line*). When represented in this way, the friendship scenario becomes very easy to grasp. This representation is a graph.

More precisely, the graphs dealt with in this chapter are called *undirected graphs*, because the lines connecting two vertices have no implied direction.

The set of vertices, here representing the set of friends, is usually referred to as V. The set of edges, here representing friendship connections, is usually referred to as E. Since a graph is composed of a set of vertices and a set of edges, it is commonly denoted as $G = (V, E)$. Vertices which are endpoints of an edge are said to be *adjacent* to each other. Besides, an edge is said to be *incident* to the vertices at both ends. The number of edges connected to a vertex defines its *degree*.

The graph partitioning problem can be neatly described using this terminology.

Graph partitioning problem

Given an undirected graph $G = (V, E)$ with an even number of vertices $n = |V|$[1], divide V into two subsets L and R with the same number of vertices (*uniform partition* or *equipartition*) satisfying $L \cap R = \emptyset$, $L \cup R = V$, $|L| = |R| = n/2$, so as to minimize the number of edges across L and R (more precisely, the number of edges $\{i, j\}$ such that either $i \in L$ and $j \in R$, or $i \in R$ and $j \in L$).

In order to define the graph partitioning problem more precisely, we will formulate it as an integer optimization problem. Given an undirected graph $G = (V, E)$, the pair (L, R) is a partition the set of vertices into two subsets L and R (i.e., a bipartition) if it satisfies $L \cap R = \emptyset$ (no intersection) and $L \cup R = V$ (the union is the whole set of vertices). Even though L stands for left and R for right, nothing changes if their roles are swapped; hence, (L, R) is a non-ordered pair. Introducing binary variables x_i which will take the value 1 when vertex i is included in subset L, and the value 0 otherwise (i.e., i is included in subset R), for having vertices equally divided the sum of x_i must be equal to $n/2$. When an edge $\{i, j\}$ is across L and R, either $x_i(1 - x_j)$ or $(1 - x_i)x_j$ become 1, allowing us to write

---

[1] The number of elements included in a set V is called the *cardinality* of the set, and is represented by $|V|$.

the following formulation.

$$\text{minimize}$$

$$\sum_{\{i,j\}\in E} \left(x_i(1-x_j) + (1-x_i)x_j \right)$$

$$\text{subject to}$$

$$\sum_{i\in V} x_i = n/2$$

$$x_i \in \{0,1\} \forall i \in V$$

Many of the available mathematical optimization solvers do not support minimization problem whose objective function is not convex (for the definition of convex function refer to Chapter piecewiselinear). The above quadratic terms are not convex. Even though SCIP does provide support for these cases, it is much more efficient for solving linear problems. Therefore, in cases where we can find an equivalent linear formulation is it advisable to use it. We will see that for the graph partition problem this is possible.

Let binary variables y_{ij} model the case where edges are incident to different subsets, i.e., $y_{ij} = 1$ if the endpoints of edge $\{i,j\}$ are across L and R, $y_{ij} = 0$ otherwise. Variables $x_i, i \in V$ have the same meaning as above. With these variables, the graph partitioning problem can be modeled with linear integer optimization as follows.

$$\text{minimize} \quad \sum_{\{i,j\}\in E} y_{ij}$$

$$\text{subject to} \quad \sum_{i\in V} x_i = n/2$$

$$x_i - x_j \le y_{ij}$$
$$\forall \{i,j\} \in E$$
$$x_j - x_i \le y_{ij}$$
$$\forall \{i,j\} \in E$$
$$x_i \in \{0,1\}$$
$$\forall i \in V$$
$$y_{ij} \in \{0,1\}$$
$$\forall \{i,j\} \in E$$

As the objective is to minimize the sum of variables y_{ij}, their value will be as much as possible zero, but constraints force some of them to be one. The first constraint defines an equal division of the set of vertices. The second constraint implies that if $i \in L$ and $j \notin L$ (i.e., edge $\{i,j\}$ is across subsets L and R), then $y_{ij} = 1$ The third constraint implies that if $j \in L$ and $i \notin L$, then $y_{ij} = 1$.

A model for this in Python/SCIP can be written as follows:

The function `gpp` requires as parameters a set of vertices `V` and a set of edges `E`. An example of a function for generating such data randomly given below.

With these functions, the main program can be written as follows.

> **if __name__ == "__main__":** V,E = make_data(4,.5) model = gpp(V,E) model.optimize()
> print("Optimal value:", model.getObjVal())

Note: **Margin seminar 5**

5.1. Graph partitioning problem

Mathematical optimization and constraint programming

Although the central paradigm used in this document for solving optimization problems is *mathematical optimization* (previously known as *mathematical programming*), another framework for solving similar problems is *constraint programming*. These two technologies, more than competing, complement each other as powerful optimization tools. Depending on the problem it may be advisable to use tools from mathematical optimization, from constraint programming, or to combine the two technologies.

In mathematical optimization, variables must be defined as real or integer numbers. In constraint programming, variables typically take one value from a given discrete set, called the *domain*. Constraint programming is good at solving problems with a combinatorial structure; it is weak for handling continuous (real) variables, for which mathematical optimization is very powerful. On the other hand, problems containing non-convex expressions, such as the graph partitioning problem, can often be easily solved in constraint programming. In addition, it is also good for problems for which it is difficult to find a feasible solution, such as puzzles or the staff scheduling problem described in Section 9.3.

SCIP is specialized in *constraint integer optimization*, combining techniques for constraint programming, mixed-integer optimization, and satisfiability problems.

5.2 Maximum stable set problem

You are choosing, from a group of six friends, with whom to go for a picnic. However, persons linked with an edge in Figure *Maximum stable set problem* are on very unfriendly terms with each other, so if both of them go to the picnic, it will be spoiled. To have as many friends as possible in the picnic, who should be invited?

(a)   (b)
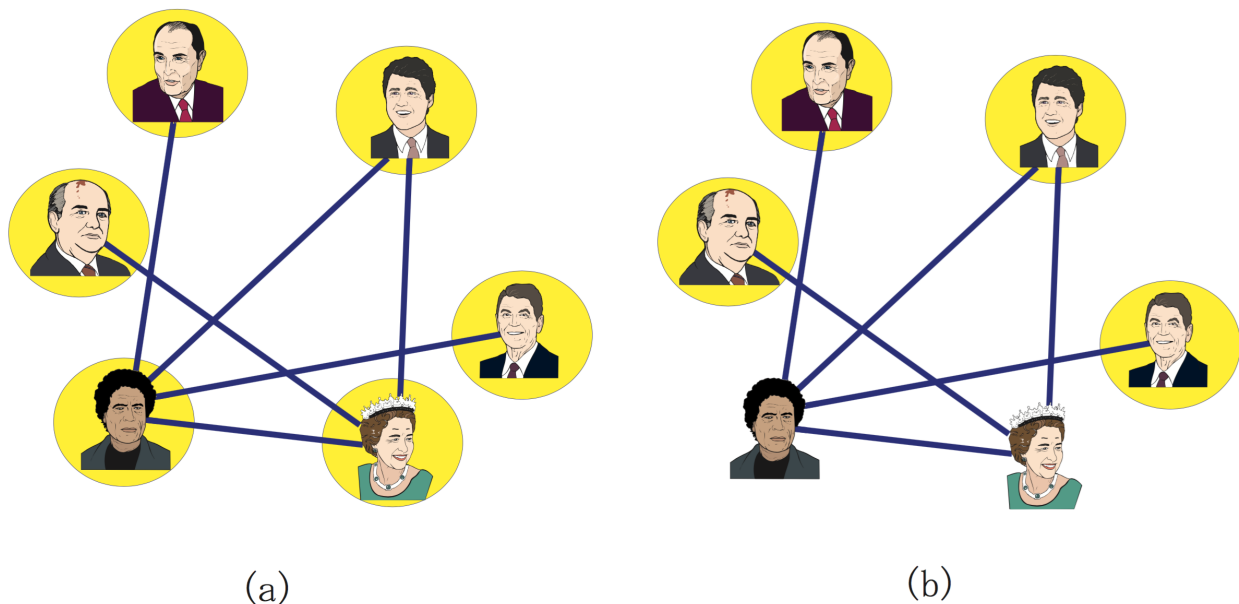
Fig. 2: Maximum stable set problem

(a) Graph where an edge between two persons indicates that they are on unfriendly terms.

(b) Maximum number of persons that can go to a picnic such that all the invitees are in good terms. The four persons encircled can all be at the picnic without spoiling it; this is the optimal solution.

This is an example of the so-called maximum stable set problem, a fundamental problem in graph theory. The maximum stable sets problem can be defined as follows.

Maximum stable set problem

Given an undirected graph $G = (V, E)$, a subset $S \subseteq V$ is called a *stable set* when there isn't any edge among vertices of S. The problem is to find a stable set S such that its cardinality (i.e., $|S|$, the number of vertices it contains) is maximum.

Considering the complementary graph this problem—the complementary graph inverts the edges, i.e., contains edges only between pairs of vertices for which there is *no edge* in the original graph—the *maximum clique problem* is defined below. These two problems are equivalent, in the sense that they can be converted through a simple transformation, and the solution is the same (see Figure *Maximum stable set and maximum clique*).
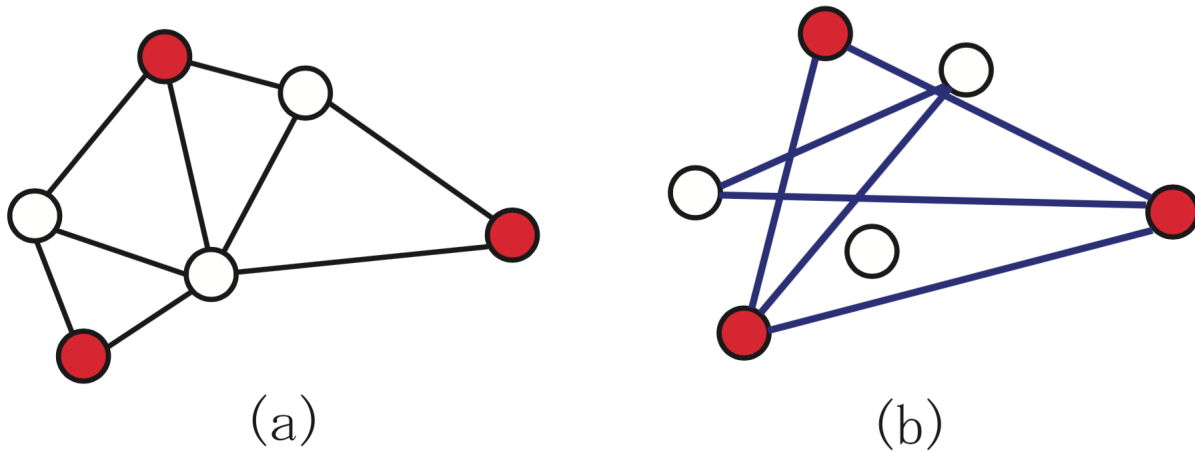
Fig. 3: Maximum stable set and maximum clique

(a) Maximum stable set.

(b) Maximum clique on the complementary graph.

Maximum clique problem

Given an undirected graph $G = (V, E)$, a subset $C \subseteq V$ is called a *clique* when the subgraph induced by C is complete (in a *complete graph* there is edge connecting all pairs of vertices; the subgraph *induced* by a subset of vertices contains all the edges of the original graph with both ends in that subset). The problem is to find a clique C which maximizes cardinality $|C|$.

These problems have applications in coding theory, reliability, genetics, archeology and VLSI design, among others. Using a variable for each vertex i, which take on the value 1 when vertex i is included in the stable set, this problem can be formulated as follows.

$$
\begin{aligned}
\text{maximize} \quad & \sum_{i \in V} x_i \\
\text{subject to} \quad & x_i + x_j \leq 1 \\
& \forall \{i, j\} \in E \\
& x_i \in \{0, 1\} \\
& \forall i \in V
\end{aligned}
$$

This formulation can be written as a Python/SCIP program in the following manner.

This function can be used similarly to the one described above for the graph partitioning problem.

5.3 Graph coloring problem

You are concerned about how to assign a class to each of your friends. Those which are on unfriendly terms with each other are linked with an edge in Figure *Graph coloring problem*. If put on the same class, persons on unfriendly terms will start a fight. To divide your friends into as few classes as possible, how should you proceed?
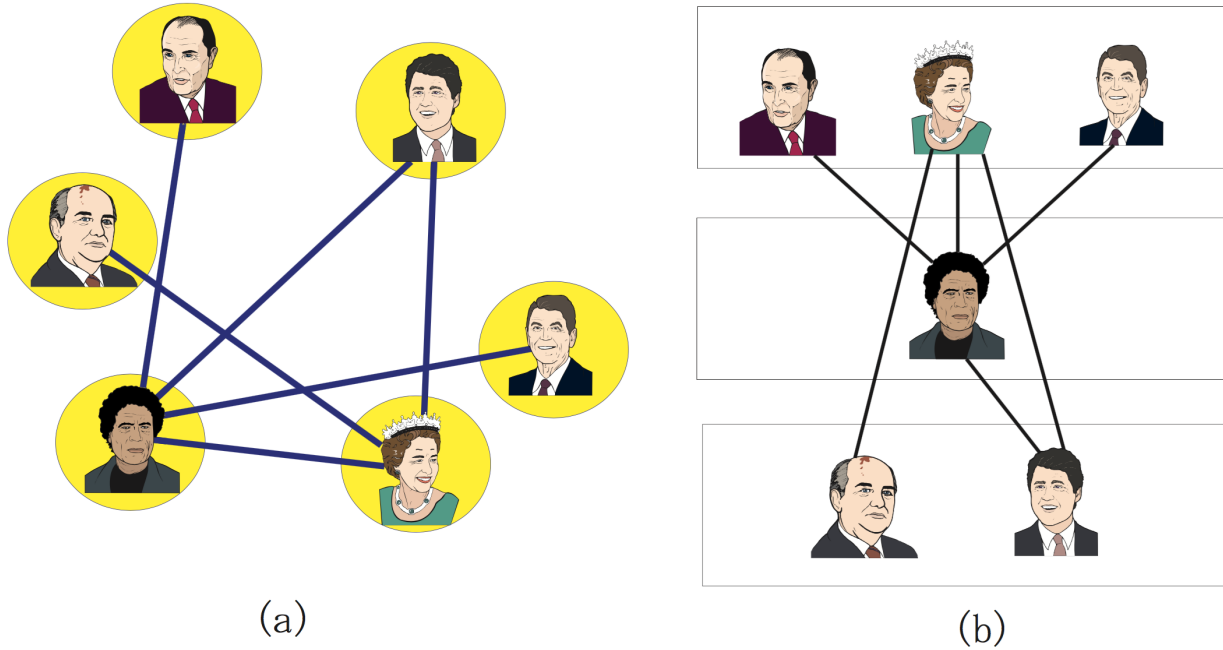

(a)   (b)

Fig. 4: Graph coloring problem

(a) Graph where an edge between two persons indicates that they are on unfriendly terms.

(b) Dividing into three classes keeps persons on unfriendly terms in different classes. The value of the objective function (the number of classes) being 3, this is an optimal solution.

This is an example of the classical optimization problem called *graph coloring problem*, which can be defined as follows.

> Graph coloring problem
>
> Given an undirected graph $G = (V, E)$, a $K-$partition is a division of the vertices V into K subsets $V_1, V_2, \ldots, V_K$ such that $V_i \cap V_j = \emptyset, \forall i \neq j$ (there is no overlap), and $\bigcup_{j=1}^{K} V_j = V$ (the union of subsets is the full set of vertices). Each $V_i (i = 1, 2, \ldots K)$ is called a *color class*. In a $K-$partition, if all the vertices in a color class V_i form a stable set (i.e., there is no edge among two vertices in that class), it is called $K-$*coloring*.

For a given undirected graph, the graph coloring problem consists of finding the minimum K for which there is a $K-$coloring; this is called the graph's *chromatic number*.

The graph coloring problem has a variety of applications, such as timetabling and frequency allocation.

For writing a mathematical formulation for the graph coloring problem, an upper bound $K_{\max}$ of the number of colors is required. In other words, the optimal number of colors K determined as an integer $1 \leq K \leq K_{\max}$.

Let us define binary variables x_{ik} such that when a vertex i is assigned a color k, x_{ik} takes the value 1; otherwise, x_{ik} takes the value 0. Besides, binary variable $y_k = 1$ indicates that color k has been used, i.e., set V_i contains at least one

vertex; otherwise, V_i is empty and $y_k = 0$, indicating that color k was not required.

$$\text{minimize} \quad \sum_{k=1}^{K_{\max}} y_k$$

$$\text{subject to} \quad \sum_{k=1}^{K_{\max}} x_{ik} = 1$$
$$\forall i \in V$$
$$x_{ik} + x_{jk} \le y_k$$
$$\forall \{i, j\} \in E; k = 1, \ldots, K_{\max}$$
$$x_{ik} \in \{0, 1\}$$
$$\forall i \in V; k = 1, \ldots, K_{\max}$$
$$y_k \in \{0, 1\}$$
$$k = 1, \ldots, K_{\max}$$

The first constraint in this formulation indicates that exactly one color is assigned to each vertex. The second constraint connects variables x and y, allowing coloring with color k only if $y_k = 1$, and forbids the endpoits of any edge $\{i, j\}$, vertices i and j, from having the same color simultaneously.

Many of the mathematical optimization solvers, including SCIP, use the branch-and-bound method (see Margin Seminar branch-and-bound). Since all color classes in the formulation above are treated indifferently, the solution space has a great deal of symmetry. Symmetry causes troubles to branch-and-bound, increasing enormously the size of the tree that needs to be explored. For example, the solutions $V_1 = 1, 2, 3, V2 = 4, 5$ and $V_1 = 4, 5, V2 = 1, 2, 3$ are equivalent, but are represented by different vectors x and y. In this case, there occurs a phenomenon where branching on any of the variables x, y leads to no improvements in the lower bound. When solving the graph coloring problem with a mathematical optimization solver, to avoid some symmetry in the solution space, it is recommended to add the following constraints.

$$y_k \ge y_{k+1} \quad k = 1, \ldots, K_{\max} - 1$$

Adding the above constraint forces to use preferentially color classes with low subscripts. Simply adding this constraint may considerably improve the solving time.

Tip: Modeling tip 5

When there is symmetry in a formulation, add constraints for removing it.

When formulations for integer optimization problems have a large amount of symmetry, the branch-and-bound method is weak. In such a case, by adding constraints for explicitly breaking symmetry in the formulation, the solving time may be dramatically improved. However, deciding what constraints should be added is still a matter of craftsmanship, there are no uniform guidelines. In the authors' experience, adding simple constraints using the 0-1 variables such as those added in the graph coloring problem often works well. However, in some cases adding elaborate constraints will break the structure of the problem, and in these cases the solver is likely to become slower; hence, one often needs careful experimentation for deciding if such constraints are useful.

A program in Python/SCIP implementing a formulation for the graph coloring problem, including the a constraint for removing symmetry, is as follows.

In some cases, by adding SOS (special ordered set) constraints this formulation can be improved.

Tip: Modeling tip 6

When in a group of binary variables only one (or two consecutive) takes a positive value, use special ordered sets.

A *special ordered set (SOS)* is a constraint applied to a set of variables. There are SOS constraints of types 1 and 2. For special ordered set constraints of type 1, at most one variable in the set may take non-zero values. For special ordered sets of type 2, at most two consecutive variables (in the specified order) may be non-zero.

In the graph coloring problem, since each vertex may colored in any color, we may declare a special ordered set of type 1 for each vertex, meaning that it takes a value, but at most one may be non-zero.

Especially when the solutions contain symmetry, providing information concerning these special ordered sets often improves efficiency during the search for a solution. (Even though the improvements are not guaranteed, it is worth trying.) In addition, special ordered sets of type 2 play an effective role in the approximation of nonlinear functions by piecewise linear functions. This is described in Section 8.2.1

In the approach shown above, it was intended to minimize the number of colors used, and thus determine the chromatic number K. Let us now turn to a different approach which will allow us to solve larger instances, where the number of colors used is fixed.

If number of colors to be used is fixed and limited, there is no guarantee that we can assign a different color to each endpoint of all edges in the graph. Let a new variable z_{ij} be 1 if the endpoints of edge $\{i, j\}$ have been assigned the same color (i.e., $\{i, j\}$ is a *bad edge*), 0 otherwise. The objective is now to minimize the number of bad edges; if the optimum is 0, it means that the colors assigned are feasible, and hence that the number of colors used is an upper bound to the chromatic number K. On the other hand, if there are bad edges in the optimum, then the value that had been fixed for the number of colors is less than the chromatic number.

$$\text{minimize} \quad \sum_{\{i,j\}\in E} z_{ij}$$

$$\text{subject to} \quad \sum_{k=1}^{K} x_{ik} = 1$$
$$\forall i \in V$$
$$x_{ik} + x_{jk} \leq 1 + z_{ij}$$
$$\forall \{i, j\} \in E; k = 1, \ldots, K$$
$$x_{ik} \in \{0, 1\}$$
$$\forall i \in V; k = 1, \ldots, K$$
$$z_{ij} \in \{0, 1\}$$
$$\forall \{i, j\} \in E$$

Here, the objective is to minimize the number of bad edges. The first constraint indicates that the exactly one color is assigned to each vertex. The second constraint determines that edges $\{i, j\}$ whose endpoints i and j are assigned the same color class are bad edges (i.e., $z_{ij} = 1$).

Follows a program in Python/SCIP implementing this formulation for the graph coloring problem.

The optimum K (i.e., the smallest value such that the optimum of the above problem is 0) may be determined through binary search. Given an upper and a lower bound to the chromatic number (e.g., the number of vertices n and 1, respectively), the binary search algorithm can be written as follows.
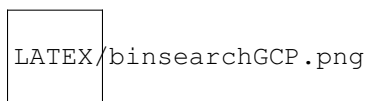

Fig. 5: Binary search method for solving the graph coloring problem.

Next we present code in Python for the same purpose.

The approach for solving the graph coloring problem using binary search and a formulation with fixed K can solve larger problems that the initial, standalone formulation.

CHAPTER 6

Routing problems

Todo: Adapt everything: figures, maths, …

In this chapter we will consider several problems related to routing, discussing and characterizing different mathematical optimization formulations. The roadmap is the following. Section *Traveling Salesman Problem* presents several mathematical formulations for the traveling salesman problem (TSP), one of the most extensively studied optimization problems in operations research. In section *Traveling Salesman Problem with Time Windows* we extend one of the formulations for the TSP for dealing with the case where there is a time interval within which each vertex must be visited. Section *Capacitated Vehicle Routing Problem* describes the capacity-constrained delivery planning problem, showing a solution based on the cutting plane method.

6.1 Traveling Salesman Problem

Here we consider the traveling salesman problem, which is a typical example of a combinatorial optimization problem in routing. Let us start with an example of the traveling salesman problem.

You are thinking about taking a vacation and taking a tour of Europe. You are currently in Zurich, Switzerland, and your aim is to watch a bullfight in Madrid, Spain, to see the Big Ben in London, U.K., to visit the Colosseum in Rome, Italy, and to drink authentic beers in Berlin, Germany. You decide to borrow a rental helicopter, but you have to pay a high rental fee proportional to the distance traveled. Therefore, after leaving, you wish to return to Zurich again, after visiting the other four cities (Madrid, London, Rome, Berlin) by traveling a distance as short as possible. Checking the travel distance between cities, you found that it is as shown in Figure *Traveling salesman problem*. Now, in what order should you travel so that distance is minimized?

Let us define the problem without ambiguity. This definition is based on the concept of graph, introduced in Section *Graph problems*.

Traveling salesman problem (TSP)

Given an undirected graph $G = (V, E)$ consisting of n vertices (cities), a function $c : E \to \mathbb{R}$ associating a distance (weight, cost, travel time) to each edge, find a tour which passes exactly once in each city and minimizes the total distance (i.e., the length of the tour).
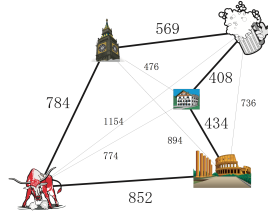
Fig. 1: Traveling salesman problem
Graph representation of cities in Europe (numerical value on edirs is the distance in miles) and optimum solution (thick line).

When the problem is defined on a non-oriented graph (called an *undirected graph*), as in the above example, we call it a *symmetric traveling salesman problem*. Symmetric means that the distance from a given point a to another point b is the same as the distance from b to a. Also, the problem defined on a graph with orientation (called a *directed graph* or *digraph*)) is called an asymmetric traveling salesman problem; in this case, the distance for going from a point to another may be different of the returning distance. Of course, since the symmetric traveling salesman problem is a special form of the asymmetric case, a formulation for the latter can be applied as it is to symmetric problems (independently from whether it can be solved efficiently or not).

In this section we will see several formulations for the traveling salesman problem (symmetric and asymmetric) and compare them experimentally. Section *Subtour elimination formulation* presents the subtour elimination formulation for the symmetric problem proposed by Dantzig-Fulkerson-Johnson [DFJ54]. Section *Miller-Tucker-Zemlin (potential) formulation* presents an enhanced formulation based on the notion of *potential* for the asymmetric traveling salesman problem proposed by Miller-Tucker-Zemlin [MTZ69]. Sections *Single-commodity flow formulation* and *Multi-commodity flow formulation* propose formulations using the concept of flow in a graph. In *Single-commodity flow formulation* we present a single-commodity flow formulation, and in *Multi-commodity flow formulation* we develop a multi-product flow formulation.

6.1.1 Subtour elimination formulation

There are several ways to formulate the traveling salesman problem. We will start with a formulation for the symmetric case. Let variables x_e represent the edges selected for the tour, i.e., let x_e be 1 when edge $e \in E$ is in the tour, 0 otherwise.

For a subset S of vertices, we denote $E(S)$ as the set of edges whose endpoints are both included in S, and $\delta(S)$ as the set of edges such that one of the endpoints is included in S and the other is not. In order to have a traveling route, the number of selected edges connected to each vertex must be two. Besides, the salesman must pass through all the cities; this means that any tour which does not visit all the vertices in set V must be prohibited. One possibility for ensuring this is to require that for any proper subset $S \subset V$ with cardinality $|S| \geq 2$, the number of selected edges whose endpoints are both in S is, at most, equal to the number of vertices $|S|$ minus one.

From the above discussion, we can derive the following formulation.

$$\text{minimize}$$

$$\sum_{e \in E} c_e x_e$$

$$\text{subject to}$$

$$\sum_{e \in \delta(\{i\})} x_e = 2 \qquad \forall i \in V,$$

$$\sum_{e \in E(S)} x_e \leq |S| - 1 \qquad \forall S \subset V, 2 \leq |S| \leq |V| - 2,$$

$$X_e \in \{0, 1\} \forall e \in E.$$

Since the number of edges connected to a vertex is called its degree, the first constraint is called *degree constraint*. The second constraint is called the *subtour elimination inequality* because it excludes partial tours (i.e., cycles which pass through a proper subset of vertices; valid cycles must pass through all the vertices).

For a given subset S of vertices, if we double both sides of the subtour elimination inequality and then subtract the degree constraint

$$\sum_{e \in \delta(\{i\})} x_e = 2$$

for each vertex $i \in S$, we obtain the following inequality:

$$\sum_{e \in \delta(S)} x_e \geq 2, \qquad \forall S \subset V, |S| \geq 2.$$

This constraint is called a *cutset inequality*, and in the case of the traveling salesman problem it has the same strength as the subtour elimination inequality. In the remainder of this chapter, we consider only the cutset inequality.

The number of subsets of a set increases exponentially with the size of the set. Similarly, the number of subtour elimination constraints (cutset constraints) for any moderate size instance is extremely large. Therefore, we cannot afford solving the complete model; we have to resort to the so-called *cutting plane method*, where constraints are added as necessary.

Assuming that the solution of the linear relaxation of the problem using only a subset of constraints is $\bar{x}$, the problem of finding a constraint that is not satisfied for this solution is usually called the *separation problem* (notice that components of $\bar{x}$ can be fractional values, not necessarily 0 or 1). In order to design a cutting plane method, it is necessary to have an efficient algorithm for the separation problem. In the case of the symmetric traveling salesman problem, we can obtain a violated cutset constraint (a subtour elimination inequality) by solving a maximum flow problem for a network having $\bar{x}_e$ as the capacity, where $\bar{x}_e$ is the solution of the linear relaxation with a (possibly empty) subset of subtour elimination constraints. Notice that if this solution has, e.g., two subtours, the maximum flow from any vertex in the first subtour to any vertex in the second is zero. By solving finding the maximum flow problem, we also obtain the solution of the *minimum cut problem*, i.e., a partition of the set of vertices V into two subsets $(S, V \setminus S)$ such that the capacity of the edges between S and $V \setminus S$ is minimum [FF56]. A minimum cut is obtained by solving

the following max-flow problem, for sink vertex $k = 2, 3, \ldots, n$:

$$\text{maximize} \quad \sum_{j:j>1} f_{1j}$$

$$\text{subject to} \quad \sum_{j:i<j} f_{ij} - \sum_{j:i>j} f_{ji} = 0$$

$$\forall i : i \neq 1, i \neq k$$

$$-\bar{x}_{ij} \leq f_{ij} \leq \bar{x}_{ij}$$

$$\forall i < j$$

The objective represents the total flow out of node 1. The first constraint concerns flow preservation at each vertex other than the source vertex 1 and the target vertex k, and the second constraint limits flow capacity on each arc. In this model, in order to solve a problem defined on an undirected graph into a directed graph, a negative flow represents a flow in the opposite direction.

As we have seen, if the optimum value of this problem is less than 2, then a cutset constraint (eliminating a subtour) which is not satisfied by the solution of the previous relaxation has been found. We can determine the corresponding cut $(S, V \setminus S)$ by setting $S = \{i \in V : \pi_i \neq 0\}$, where π is the optimal dual variable for the flow conservation constraint.

Here, instead of solving the maximum flow problem, we will use a convenient method to find *connected components* for the graph consisting of the edges for which x_e is positive in the previous formulation, when relaxing part of the subtour elimination constraints. A graph is said to be *connected* if there is a path between any pair of its vertices. A *connected component* is a maximal connected subgraph, i.e., a connected subgraph such that no other connected subgraph strictly contains it. To decompose the graph into connected components, we use a Python module called *networkX*[1].

The following function *addcut* takes as argument a set of edges, and can be used to add a subtour elimination constraint corresponding to a connected component $S(\neq V)$.

```
def addcut(cut_edges):
    G = networkx.Graph()
    G.add_edges_from(cut_edges)
    Components = list(networkx.connected_components(G))
    if len(Components) == 1:
        return False
    model.freeTransform()
    for S in Components:
        model.addCons(quicksum(x[i,j] for i in S for j in S if j>i) <= len(S)-1)
    return True
```

In the second line of the above program, we create an empty undirected graph object G by using the *networkx* module and construct the graph, by adding vertices and edges in the current solution *cut_edges*, in line 3. Next, in line 4, connected components are found by using function *connected_components*. If there is one connected component (meaning that there are no subtours), *False* is returned. Otherwise, the subtour elimination constraint is added to the model (lines 7 to 10).

Using the *addcut* function created above, an algorithm implementing the cutting plane method for the symmetric travelling salesman problem is described as follows.

```
def solve_tsp(V,c):
    model = Model("tsp")
    model.hideOutput()
```

(continues on next page)

[1] *networkX* is a Python module containing various algorithms for graphs, and can be downloaded from https://networkx.github.io

(continued from previous page)

```
4        x = {}
5        for i in V:
6            for j in V:
7                if j > i:
8                    x[i,j] = model.addVar(ub=1, name="x(%s,%s)"%(i,j))
9        for i in V:
10           model.addCons(quicksum(x[j,i] for j in V if j < i) + \
11                       quicksum(x[i,j] for j in V if j > i) == 2, "Degree(%s)"%i)
12       model.setObjective(quicksum(c[i,j]*x[i,j] for i in V for j in V if j > i),
     ↪"minimize")
13       EPS = 1.e-6
14       isMIP = False
15       while True:
16           model.optimize()
17           edges = []
18           for (i,j) in x:
19               if model.getVal(x[i,j]) > EPS:
20                   edges.append( (i,j) )
21           if addcut(edges) == False:
22               if isMIP:     # integer variables, components connected: solution found
23                   break
24               model.freeTransform()
25               for (i,j) in x:     # all components connected, switch to integer model
26                   model.chgVarType(x[i,j], "B")
27                   isMIP = True
28       return model.getObjVal(),edges
```

Firstly, the linear optimization relaxation of problem (without subtour elimination constraints) is constructed from lines 4 to 12. Next, in the *while* iteration starting at line 15, the current model is solved and cut constraints are added until the number of connected components of the graph becomes one. When there is only one connected component, variables are restricted to be binary (lines 25 and 26) and the subtour elimination iteration proceeds. When there is only one connected component in the problem with integer variables, it means that the optimal solution has been obtained; therefore the iteration is terminated and the optimal solution is returned.

In the method described above, we used a method to re-solve the mixed integer optimization problem every time a subtour elimination constraint is added. However, it is also possible to add these constraints during the execution of the branch-and-bound process.

!!!!! How, in SCIP ????? ..

but applying the branch and bound method by using the cbLazy function added in Gurobi 5.0

Note: **Margin seminar 6**

Cutting plane and branch-and-cut methods

The *cutting plane method* was originally applied to the traveling salesman problem by George Dantzig, one of the founders of linear optimization, and his colleagues Ray Fulkerson and Selmer Johnson, in 1954. Here, let us explain it by taking as an example the maximum stable set problem, introduced in Section mssp.

Let's consider a simple illustration consisting of three points (Figure *Polyhedra for the maximum stable set problem*, top). Binary variables x_1, x_2, x_3, represented as a point in the three-dimensional space (x_1, x_2, x_3), indicate whether the corresponding vertices are in the maximum stable set or not. Using

---

these variables, the stable set problem can be formulated as an integer optimization problem as follows.

$$\text{maximize}$$
$$x_1 + x_2 + x_3$$
$$\text{subject to}$$
$$x_1 + x_2 \leq 1$$
$$x_1 + x_3 \leq 1$$
$$x_2 + x_3 \leq 1$$
$$x_1, x_2, x_3 \in \{0, 1\}$$

The constraints in the above formulation state that both endpoints of an edge can not be placed in a stable set at the same time. This instance has four feasible solutions: $(0, 0, 0), (1, 0, 0), (0, 1, 0), (0, 0, 1)$. The smallest space that "wraps around" those points is called a polytope; in this case, it is a tetrahedron, defined by these 4 points, and shown in the bottom-left image of Figure *Polyhedra for the maximum stable set problem*. An optimal solution of the linear relaxation can be obtained by finding a vertex of the polyhedron that maximizes the objective function $x_1 + x_2 + x_3$. This example is obvious, and any of the points $(1, 0, 0), (0, 1, 0), (0, 0, 1)$, is an optimal solution, with optimum value 1.
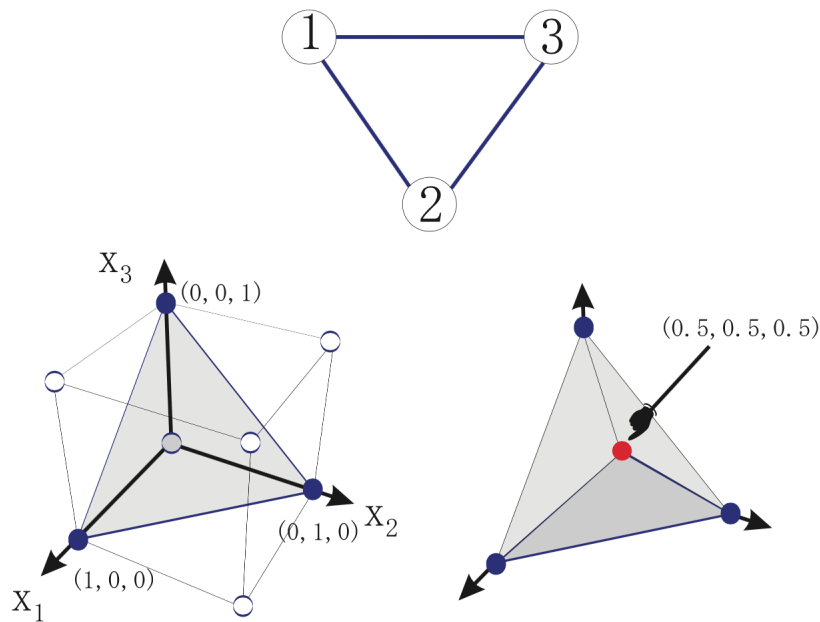
Fig. 2: Polyhedra for the maximum stable set problem
Maximum stable set instance (upper figure). Representation of the feasible region as a polyhedron, based on its extreme points (lower-left figure). The inequality system corresponding to its linear relaxation is is
$x_1 + x_2 \leq 1; x_1 + x_3 \leq 1; x_2 + x_3 1; x_1, x_2, x_3 0$; this space, and an optimum solution, are represented in the lower-right figure.

In general, finding a linear inequality system to represent a polyhedron — the so-called *convex envelope* of the feasible region — is more difficult than solving the original problem, because all the vertices of that region have to be enumerated; this is usually intractable. As a realistic approach, we will consider below a method of gradually approaching the convex envelope, starting from a region, containing it, defined by a linear inequality system.

First, let us consider the linear optimization relaxation of the stable set problem, obtained from the formulation of the stable set problem by replacing the integrality constraint of each variable ($x_i \in \{0, 1\}$) by the constraints:

$$0 \leq x_1 \leq 1,$$
$$0 \leq x_2 \leq 1,$$
$$0 \leq x_3 \leq 1.$$

Solving this relaxed linear optimization problem (the *linear relaxation*) yields an optimum of 1.5, with optimal solution (0.5, 0.5, 0.5) (Figure *Polyhedra for the maximum stable set problem*, bottom-right figure). In general, only solving the linear relaxation does not lead to an optimal solution of the maximum stable set problem.

It is possible to exclude the fractional solution (0.5, 0.5, 0.5) by adding the condition that x must be integer, but instead let us try to add an linear constraint that excludes it. In order not to exclude the optimal solution, it is necessary to generate an expression which does not intersect the polyhedron of the stable set problem. An inequality which does not exclude an optimal solution is called a *valid inequality*. For example, $x1 + x2 \leq 1$ or $x1 + x2 + x3 \leq 10$ are valid inequalities. Among the valid inequalities, those excluding the solution of the linear relaxation problem are called *cutting planes*. For example, $x_1 + x_2 + x_3 \leq 1$ is a cutting plane. In this example, the expression $x_1 + x_2 + x_3 \leq 1$ is in contact with the two-dimensional surface (a facet) of the polyhedron of the stable set problem. Such an expression is called a *facet-defining* inequality. Facets of the polyhedron of a problem are the strongest valid inequalities.

The cutting plane method is a process to iteratively solve the linear optimization problem by sequentially adding separating, valid inequalities (facet-defining inequalities are preferable) (Fig. 5.3).

The cutting plane method was extended to the general integer optimization problem by Ralph Gomory, at Princeton University, in 1958. Although this method has been shown to converge to the optimum on a finite number of iterations, in practice it failed to solve even medium-size instances (at that time). New theoretical developments have since that time been added to cutting plane method, which has been successfully incorporated into the branch-and-bound method, as a general solution technique for integer optimization problems. The method of adding these inequalities at a node of the branch-and-bound method is called the *branch-and-cut*, and forms the core of a modern mathematical optimization solver. SCIP also includes this technique, which is essential in solving large and difficult integer optimization problems.

Tip: Modeling tip 7

When the number of constraints is very large, use the cutting plane method or the branch-and-cut method.

In formulations in which the number of constraints becomes enormous, the subtour elimination constraint in the traveling salesman problem, it is necessary to use the cutting plane method for finding only the necessary constraints.

6.1.2 Miller-Tucker-Zemlin (potential) formulation

Let us now consider a formulation with a number of constraints of polynomial order.

Consider an asymmetric traveling salesman problem. The input is a directed graph $G = (V, A)$, where V is the set of vertices and A is a set of (directed) arcs, and a distance function on the arcs $c : A \rightarrow \mathbb{R}$, and the aim is to find the shortest distance cycle through all the vertices.

We introduce 0-1 variables x_{ij} which is 1 when visiting vertex j next to vertex i, 0 otherwise, and real variable u_i which represent the visiting order of vertex i. Interpret variable u_1 as the *potential* at starting vertex u_1 (we could define any other vertex as the starting point).

When visiting vertex j next to vertex i, a constraint will force the potential of j to be $u_j = u_i + 1$, for any vertex except 1; hence, possible values for u_i are $1, 2, ..., n - 1$.

Using these variables, the asymmetric traveling salesman problem can be formulated as follows.

$$\sum_{i \neq j} c_{ij} x_{ij}$$

$$\sum_{j:j \neq i} x_{ij} = 1 i = 1, \cdots, n$$

$$\sum_{j:j \neq i} x_{ji} = 1 i = 1, \cdots, n$$

$$u_i + 1 - (n - 1)(1 - x_{ij}) \leq u_j i = 1, \cdots, n,$$

$$j = 2, \cdots, n : i \neq j$$

$$1 \leq u_i \leq n - 1 i = 2, \cdots, n$$

$$x_{ij} \in \{0, 1\} \forall i \neq j$$

The first and second constraints are assignment constraints, and ensure that each vertex is incident to one outgoing arc and one incoming arc.

The third constraint is sometimes called Miller-Tucker-Zemlin constraint, proposed in [10], and define the order in which each vertex i is visited on a tour. Here, since u_i can be interpreted as the potential at vertex i, we will call it the potential constraint.

The extended formulation using the potential constraint is much weaker than the previous formulation using subtour elimination constraints. This is because the constraint for forcing $u_j = u_i + 1$ only when $x_{ij} = 1$, the coefficient $n - 1$ of the term $(1 - x_{ij})$ is a "Big M" (See section 2.2).

The fourth constraint indicates the upper and lower bounds of the potential.

In the following, we will strengthen the formulation using potential constraints by performing and operation called *lifting*.

First, consider applying the lifting operation based on the potential constraint. Add the term of x_{ji} to the left side and let its coefficient be $alpha$.

$$u_i + 1 - (n - 1)(1 - x_{ij}) + \alpha_{ji} x_{ji} \leq u_j$$

Consider making the coefficient $alpha$ as large as possible, as long as not to exclude feasible solutions.

When $x_{ji} = 0$, $alpha$ has no impact; we obtain the original potential constraint itself, which is a valid inequality.

In the case of $x_{ji} = 1$, since $x_{ij} = 0$ and $u_j + 1 = u_i$ are always valid solutions, at this time the range of $alpha$ is:

$$\alpha \leq u_j - u_i - 1 + (n - 1) = n - 3$$

In order to have a constraint which is as strong as possible, while keeping feasibility, the inequality must be: .. math:

```
u_i + 1 - (n - 1)(1 - x_{ij}) + (n-3) x_{ji} \leq u_j
```

Next, let us apply the lifting operation based on the lower limit inequality $1 leq u_i$. Consider adding the term $(1 - x_{1i})$ to the left side, and let $beta$ be its coefficient.

$$1 + \beta(1 - x_{1i}) \leq u_i$$

When $x_{1i} = 1$, this expression is reduced to the original form, and hence is a valid inequality. When $x_{1i} = 0$, since point i is visited from the second place onward, there must be $u_i geq 2$. Therefore, we can see that $beta = 1$, obtaining:

$$1 + (1 - x_{1i}) \leq u_i$$

Consider now adding a term of x_{i1} to the left side, with coefficient $gamma$.

$$1 + (1 - x_{1i}) + \gamma x_{i1} \leq u_i$$

When $x_{i1} = 0$ the expression is feasible, because it is reduced to the original form.

When $x_{i1} = 0$, in the executable solution, since point i is visited last, $x_{1i} = 0$ and $u_i = n - 1$.

Therefore, it can be found that $gamma = n - 3$ is satisfied, obtaining:

$$1 + (1 - x_{1i}) + (n - 3)x_{1i} \leq u_i$$

Similarly, lifting the upper bound constraint $u_i leq n - 1$, we obtain

$$u_i \leq (n - 1) - (1 - xi1) - (n - 3)x_{1i}$$

Tip: Modeling tip 8

Strengthening expressions with lifting

> Often it is necessary to use a formulation that includes large numbers ("Big M") (like the potential constraints of the MTZ formulation for the asymmetric traveling salesman problem).

> When solving the problem, if the solver takes a long time and the difference between the lower bound and the upper bound (dual gap) is large, consider strengthening the expression by lifting.

> Theoretically, it is desirable to derive the strongest feasible inequality (a *facet*), but even simple lifting may have a great impact.

6.1.3 Single-commodity flow formulation

!!!!!!!!! unchecked

In this section and the next section, we introduce the formulation using the concept of "flow" (flow) of "things".

This is referred to as a single type flow formulation (single accommodation flow formulation).

Let's consider that "things" in n-1 units are placed at a specific point (1), and they are brought by a salesman for all other points.

(Of course, we assume that the salesman leaves point 1.)

From point 1, "n" th unit of "n-1" goes out, and one point is consumed at each point.

Also, it is assumed that "sales person" can only be flown on branches where salesman did not move.

In network theory, flowing "things" are called commodity, and in this formulation, we think here of carrying one kind of things, so it is called a single-product flow formulation.

In the formulation up to the previous section, we used the 0-1 variable xij that indicates that the salesman passes the branch (i, j).

Furthermore, we introduce fij as a continuous variable representing the quantity of "things" (varieties) passing through branches (i, j).

Using these symbols, the single variety flow formulation can be written as follows.

Here, the first two constraints are degree constraints, which stipulate that there are exactly one branch and one branch that will enter each point.

The third constraint represents that "things" are shipped from the first point 1 to n-1 units, and the fourth constraint represents that "things" are consumed one at each point.

The fifth and sixth constraints are capacity constraints, which means that "things" do not flow on branches where salesmen do not move.

However, for the branch (1, j) connected to point 1, "thing" with the maximum of n-1 flows, and for the other branches "max" of "n" I have stipulated.

(All may be specified as n - 1 or less, but it is somewhat enhanced expression.)

6.1.4 Multi-commodity flow formulation

As in the previous section, consider formulation based on the flow (thing) of "things".

The formulation shown here is called multi-commodity flow formulation because it thinks to flow multiple "things" (varieties).

In the multi-product flow formulation, "things" conveyed on a point by point basis are distinguished (this is the difference from the single-product flow formulation in the previous section).

From point 1, one unit type k going to another point k goes out, and at point k, one type of product k is consumed.

We introduce fkij as a continuous variable representing the quantity of the product k passing through the branch (i, j).

Using this, the multi-product flow formulation can be written as follows.

Here, the first two constraints are degree constraints, which prescribe that there are exactly one branch and one outgoing branch at each point.

The third constraint is that one unit of each product type k is shipped from the first point 1 and it is consumed at the point k.

The fourth constraint is the capacity constraint, which means that "things" do not flow on the branch where the salesman does not move.

The multi-product flow formulation is described by Gurobi / Python as follows.

6.2 Traveling Salesman Problem with Time Windows

Here we consider the traveling salesman problem (travelingsalesmanproblemwithtimewindows) with a time frame that added a time frame to the traveling salesman problem.

This problem is based on the assumption that the asymmetric traveling salesman problem assuming that a specific point 1 starts at time 0 is regarded as the traveling distance between points and the departure time for point i is the latest time ei and the latest time li It is a problem imposing the constraint that it must be between.

However, if we arrive at point i earlier than time ei, we can wait until point ei on point i.

6.2.1 One-index Potential Formulation

First of all, consider the extension of the potential (Miller-Tucker-Zemlin) constraint for the traveling salesman problem considered in Section 5.1.2.

We introduce a variable ti that represents the time to depart from point i.

ti must satisfy the following constraints.

eitili i=1,2,…,n

However, suppose $_1 = 0$, $ell\_1 = infty$

When visiting point j next to point i (xij = 1), the time tj to depart from point j is greater than or equal to the sum of travel time cij at the time of departing point i, obtain.

$$t\_i + c\_{ij} - M (1-x\_{ij}) leq t\_j qquad forall i,j : j neq 1, i neq j$$

Here, M is a constant representing a large number.

It is assumed that the movement time cij is a positive number.

When cij is 0, there is a possibility that ti = tj, and a partial tour circuit is formed.

In order to avoid this, it is necessary to add constraints similar to the traveling salesman problem, but under the assumption of cij> 0, it is possible to remove the partial tour circuit by the above constraint..

Since formulation including such a large number "BigM" is not very practical, consider strengthening by using the time frame.

The smaller the value of M, the stronger the restriction.

When xij = 0, the above constraint can be rewritten as M.

For all feasible solutions it is necessary to set M so that the above equation holds.

Since tili and tj ej, the value of M can be set equal to or more than li + cij-ej.

Of course, since it does not make sense as an expression unless M> 0, we obtain the following expression.

Here, [·] + is a symbol representing max {·, 0}.

By combining the order constraint on the traveling salesman problem and the above constraint, we obtain a potential formulation for a traveling salesman problem with time frame.

minimize

Like the traveling salesman problem, the potential constraint and the upper and lower limit constraints can be further enhanced by the lifting operation as follows.

6.3 Capacitated Vehicle Routing Problem

Here, consider the capacity constrained delivery planning problem (capacitatedvehiclerouting problem) as a practical extension of the traveling salesman problem.

The capacity constrained delivery planning problem has the following assumptions:

- A vehicle that departed from a specific point called a depot returns to the depot again via multiple customers.

In this case, the order of customers passing by the truck is called a route (see Fig. 5.4).

In this case, the truck is used as a generic term that refers to various means of transport such as trucks, trailers, and ships.

- The maximum load weight (referred to as capacity) of a transport vehicle waiting in a depot is known.

- The customer's location is known and the amount of demand for each customer is given in advance.

It is assumed that the customer's demand amount does not exceed the maximum carrying weight of the truck, and each customer is to be visited exactly once.

- The cost of moving between points is known.

- The total amount of customers' demand in one route does not exceed the maximum carrying weight of the truck (this is called capacity constraint).

- The number of types of trucks is one, and the number of trucks is predetermined.

Applications of the delivery planning problem include delivery planning to the retail store, decision of the school bus traveling circuit, delivery of mail and newspaper, garbage collection, delivery of fuel.

Of course, when applying to these applications, it is necessary to add various conditions to the above basic condition, but here we deal with only the capacity constraint as the basic form.

Let m be the number of trucks and n be the number of points (representing customers and depots).

It is assumed that customers i = 2, 3, ..., n have demand q_i and their demand is carried (or collected) by a certain transporter.

The transport vehicles k = 1, 2, ..., m have a finite load capacity upper limit Q, and the sum of the demand amounts carried by the trucks shall not exceed that value.

Normally, it is assumed that the maximum value max $\{q_i\}$ of the demand amount of the customer does not exceed the capacity Q of the transporter.

If customers with demand exceeding the maximum value of load capacity exist, they can be transformed to satisfy the above assumption by appropriately dividing the demand (so as to be within the upper limit of the load capacity).

Write expenses required when the truck moves from point i to point j as c_{ij}.

Here, the movement cost is symmetrical ($c_{ij} = c_{ji}$).

The aim of the shipping plan problem is to find the optimum route of m trucks that satisfies all customers' demand (a simple closed circuit that departs from the depot and returns to the depot again).

We introduce a variable x_{ij} that represents the number of times the truck moves between points i and j.

Because we assume a symmetric problem, the variable x_{ij} is defined only between points i and j that satisfy i < j.

For a branch where x_{ij} does not connect to the depot, it represents 1 when the haul passes and 0 when it is not, but in the case of so-called piston transport where it moves to the point j from the depot and returns immediately to the depot, X_{1j} is 2.

The formulation of capacity constrained delivery planning problem is as follows.

minimiz

Here, the first constraint specifies that there are m sets of carriers from the depot (point 1).

That is, it indicates that the number of branches representing a transport vehicle entering and leaving point 1 is 2 m.

The second constraint represents that a single truck visits each customer.

The third constraint is a constraint that simultaneously defines the capacity constraints of the truck and prohibits partial cruises.

N (S) used in this constraint is a function calculated when a customer's subset S is given, and is defined as follows.

$N(S) =$ number of vehicles required to carry customer demand within S

In order to calculate N (S), it is necessary to solve the packing problem described in Chapter 3, but in general the following lower bounds are substituted

As applied to the traveling salesman problem in Section 5.1.1, Consider a branch cut by solving a connected component on a graph where x e is a positive branch when x e (e E) is the solution of the linear relaxation problem.

CHAPTER 7

Scheduling problems

CHAPTER 8

Dynamic lot-sizing problems

CHAPTER 9

Piecewise linear approximation of nonlinear functions

Multiple objective optimization

CHAPTER 11

Second-order cone optimization

References

CHAPTER 13

Indices and tables

- genindex
- modindex
- search

Bibliography

[DFJ54] G. B. Dantzig, D. R. Fulkerson, and S. Johnson. Solution of a large scale traveling salesman problem. *Operations Research*, pages 393–410, 1954.

[Dan63] George B. Dantzig. *Linear programming and extensions*. Princeton University Press Princeton, N.J, 1963.

[FF56] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956. URL: http://www.rand.org/pubs/papers/P605/.

[GG61] P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting-stock problem. *Operations Research*, 9(6):849–859, 1961. URL: http://or.journal.informs.org/cgi/content/abstract/9/6/849, arXiv:http://or.journal.informs.org/cgi/reprint/9/6/849.pdf, doi:10.1287/opre.9.6.849.

[GG63] P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting stock problem–Part II. *Operations Research*, 11(6):863–888, 1963. URL: http://or.journal.informs.org/cgi/content/abstract/11/6/863, arXiv:http://or.journal.informs.org/cgi/reprint/11/6/863.pdf, doi:10.1287/opre.11.6.863.

[MTZ69] C. E. Miller, A. W. Tucker, and R. A. Zemlin. Integer programming formulation of traveling salesman problems. *Journal of ACM*, :326–329, 1969.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Chichester, 1986.

Index

Symbols

A

B

C

D

E

F

G

H

I

L