An Introduction to Optimization with PySCIPOpt

João Pedro Gonçalves Dionísio

February 20, 2023

Contents

Abstract

This documents serves as the lecture notes for the course “Topics in Optimization”, administered in 2023 at Faculdade de Ciências da Universidade do Porto. It can also serve as a gentle introduction to PySCIPOpt. Here, we will talk about different topics in Optimization, combining theory with practical implementation. The document is split into five main chapters: Convex Optimization, Linear Programming, Complexity Theory, Integer Optimization, and Decomposition Methods (namely Benders' and Dantzig-Wolfe). Alongside the theory, the examples will sometimes be translated into PySCIPOpt and it is with it that the algorithms will be implemented and tested.

Disclaimer: This document is unfinished and I am not an expert in everything in it. It may have incorrect information, and it is best if used as an introduction to Optimization.

Chapter 0

Preliminaries

0.1 Installing PySCIPOpt

0.1.1 Using Anaconda

1. install anaconda anaconda: download link
2. create new env: conda create –name envname
3. switch to new env: conda activate envname
4. install pyscipopt: conda install -c conda-forge pyscipopt

0.2 Enabling PySCIPOpt

Using VsCode

1. Press Ctrl + Shift + P
2. Write "Interpreter"
3. Select "Python: Select Interpreter"
4. Select the conda environment created in 0.1.1

Chapter 1

Convexity

Definition 1. *A set $X \in \mathbb{R}^n$ is said to be convex if*

$$\forall x, y \in X, (1-t)x + ty \in X, t \in [0,1] \tag{1.1}$$

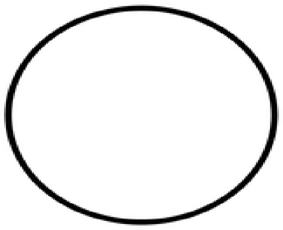


Figure 1.1: Example of a convex set

Figure 1.2: Example of a non-convex set

Theorem 1.0.1 (Supporting Hyperplanes). *Let $C \subset \mathbb{R}^n$ be a convex set, and $x \in \partial C$. Then, there exists a hyperplane H, s.t. $x \in H \cap C$ and C is contained in one of the half-spaces bounded by H.*

So every convex set can, in theory, be described as the intersection of hyperplanes, at most one for every point on the boundary of the convex set. This is one justification for the importance of linear programming - convex optimization problems where the objective functions and constraints are linear. It is the subject of the next chapter.

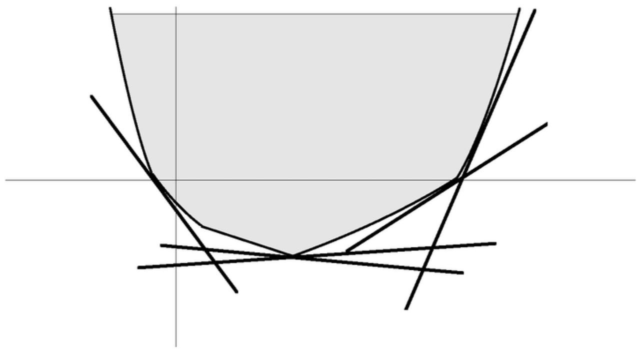


Figure 1.3: Example of approximation of a convex set by hyperplanes

Definition 2 (Convex Hull)**.** *Let $X \subseteq \mathbb{R}^n$ be a set. The* ***convex hull*** *of X, denoted by $conv(X)$, is the intersection of all convex sets containing X. Equivalently, it is the set of all convex combinations of X.*

Theorem 1.0.2 (Carathéodory's Theorem)**.** *Let $X \subseteq \mathbb{R}^n$ be a set, and consider $conv(X)$. Then,*

$$\exists x_1, \dots, x_n \in \mathbb{R}^n \mid x = \sum_{i=1}^{n+1} \lambda_i x_i \wedge \sum_{i=1}^{n+1} \lambda_i = 1 \tag{1.2}$$

Intuitively, we can think that a point P belongs to the convex hull of a set X if it is the "center of mass" of some $n+1$ points of X.

Definition 3. *A function $f : \mathbb{R}^n \to \mathbb{R}$ is said to be convex if $\forall x, y \in \mathbb{R}^n, \forall \lambda \in [0,1]$ we have*

$$f((1-\lambda)x + \lambda y) \leq (1-\lambda)f(x) + \lambda f(y) \tag{1.3}$$

Definition 4. *We call the epigraph of a function $f : X \to \mathbb{R}$, denoted by epi(f), to the following set:*

$$epi(f) = \{(x,y) \in X \times \mathbb{R} \mid f(x) \leq y\} \tag{1.4}$$

It has the geometric intuition of being the set of points "above" the function.

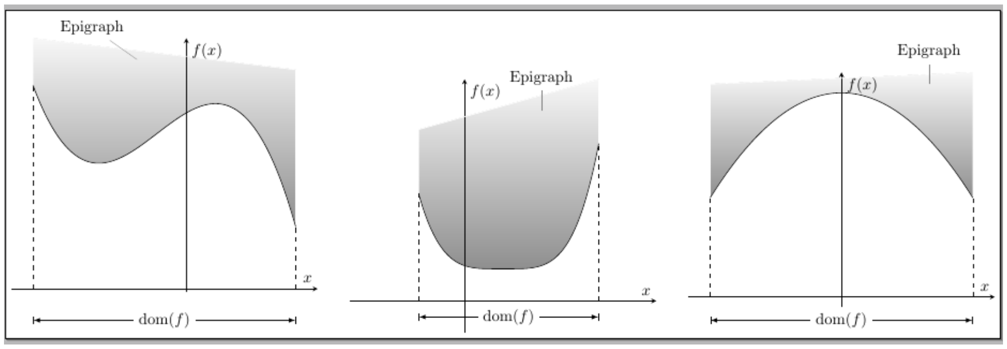


Figure 1.4: Example of epigraphs

Proposition 1.0.3. *The following two statements are equivalent:*

1. *f is convex;*
2. *epi(f) is a convex set.*

1.1 Convex Optimization

Definition 5. *We will express an optimization problem as*

$$\begin{aligned} \min_x \quad & f(x) \\ s.t. \quad & x \in S \end{aligned} \tag{1.5}$$

We call f the objective function, x the decision variable, and S the feasible region.

Definition 6. *A convex optimization problem is*

$$\begin{aligned} \min_x \quad & f(x) \\ s.t. \quad & g_i(x) \leq 0, i \in [m] \end{aligned} \tag{1.6}$$

where f is a convex function and $g_i : \mathbb{R}^n \to \mathbb{R}, i = 1 \dots, m$ are convex functions as well. It implies that $\cap_{i \in [m]} g_i$ forms a convex set.

Some texts require only that the objective function is convex and the feasible region is convex. From a theoretical perspective, the results would be the same in both cases. However, when implementing algorithms in practice, this definition could be more troublesome, due to the possibility of having to deal with non-convex functions, even if they are restricted to a domain where they are convex. For a more in-depth explanation, see Michael Grant's answer to the question The definition of a convex function, in Math Stack Exchange.

1.2 Properties of Convex Problems

Definition 7. *A point x is said to be a minimum of an optimization problem $\mathcal{P}$, if x is feasible for $\mathcal{P}$, and $f(x) \leq f(x'), \forall x'$ also feasible in $\mathcal{P}$.*

Definition 8. *A point x is said to be a local minimum of an optimization problem $\mathcal{P}$, if x is feasible for $\mathcal{P}$, and $\exists \varepsilon \mid f(x) \leq f(x'), \forall x' \in B(x, \varepsilon)$.*

Theorem 1.2.1. *Let $\mathcal{P}$ be a convex optimization problem, and without loss of generality, assume it is a minimization problem. Then, if x^* is a local minimizer, then x^* is a global minimizer.*

Proof. Let x^* be a local minimizer of the convex minimization problem $\mathcal{P}$, and $f(x^*)$ its objective value. Let $y \in \mathbf{dom}(f)$. Since P is convex, we have that $\exists x_\lambda \in P \mid x_\lambda = \lambda y + (1-\lambda)x^*$. Using the convexity of f, we arrive at the following.

$$f(x_\lambda) - f(x^*) \leq \lambda(f(y) - f(x^*))$$

The left-hand side is greater than 0 for small enough λ, given the assumption of the local optimality of x^*. From this, we get that $f(y) \geq x^*$, as required. $\square$

Thus, in convex optimization, we have a theoretical guarantee that any local optimum is also a global optimum.

1.3 Exercises

Exercise 1.3.1. *Say whether the following is a convex optimization problem.*

$$\begin{aligned} \min_x \quad & e^{x^2} \\ s.t. \quad & x \leq \sin(x) \\ & 0 \leq x \leq \pi \end{aligned} \tag{1.7}$$

Exercise 1.3.2. *Prove Proposition 1.0.3, that states that a function f is convex if and only if its epigraph is convex.*

Exercise 1.3.3. *Which functions in Figure 1.4 are convex?*

Exercise 1.3.4. *PySCIPOpt can solve general non-linear optimization problems. PySCIPOpt does not handle non-linearities in the objective function. How do these sentences not contradict each other?*

Chapter 2

Linear Programming

An important subset of Convex Optimization is Linear Programming, which tries to solve the following problems:

$$\begin{aligned} \min_x \quad & c^\intercal x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0 \end{aligned} \tag{2.1}$$

Throughout this text, we will assume that $c, b, x \in \mathbb{R}^n$ and A is a $m \times n$ matrix with real entries.

Example 2.0.1. *A paper company can produce two types of industrial-sized sheets, type A and type B. Type A can be produced at a ratio of $200m$ per hour, while type B can be produced at a ratio of $140m$ per hour. The profits from each type of paper are 25 cents per meter and 30 cents per meter, respectively. Taking the market demand into account, next week's production schedule cannot exceed $6000m$ for paper of type A and $4000m$ for paper of type B. If on that week there is a limit of 40 production hours, how many meters of each product should be produced to maximize the profit?*

$$\begin{aligned} \max_{A,B} \quad & 25A + 30B \\ s.t. \quad & A/200 + B/140 \leq 40 \\ & A \leq 6000 \\ & B \leq 4000 \\ & A, B \geq 0 \end{aligned} \tag{2.2}$$

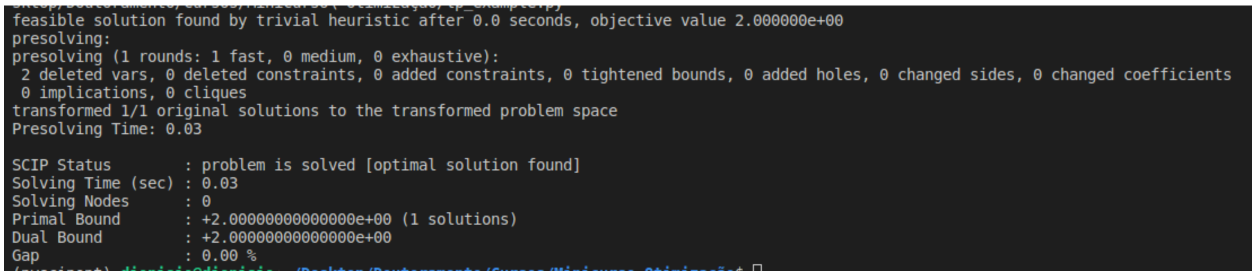
In PySCIPOpt, the following code solves the LP:

```python
from pyscipopt import Model

m = Model()                                 # creates the Model m
A = m.addVar(lb=0)                          # adds a variable to m with a lower bound of 0 and calls it x
B = m.addVar(lb=0)                          # same as above, but called y
m.addCons(A/200 + B/140 <= 40)              # adds a constraint to the model
m.addCons(A <= 6000)
m.addCons(B <= 4000)

m.setObjective(25*A + 30*B, "maximize")     # sets the objective of the model.
m.optimize()                                # optimizes model m
```

Listing 2.1: Linear Programming in PySCIPOpt

Which yields the output:

As this problem is too simple, SCIP's trivial heuristic automatically finds a feasible solution. Then, the presolving starts. We will expand a bit on reformulations and ways to make a model easier in Chapter 4. Then, SCIP starts to solve the problem. Immediately, it is able to prove the optimality of the initial solution it obtained.



```
feasible solution found by trivial heuristic after 0.0 seconds, objective value 2.000000e+00
presolving:
presolving (1 rounds: 1 fast, 0 medium, 0 exhaustive):
 2 deleted vars, 0 deleted constraints, 0 added constraints, 0 tightened bounds, 0 added holes, 0 changed sides, 0 changed coefficients
 0 implications, 0 cliques
transformed 1/1 original solutions to the transformed problem space
Presolving Time: 0.03

SCIP Status        : problem is solved [optimal solution found]
Solving Time (sec) : 0.03
Solving Nodes      : 0
Primal Bound       : +2.00000000000000e+00 (1 solutions)
Dual Bound         : +2.00000000000000e+00
Gap                : 0.00 %
```

Figure 2.1: PySCIPOpt log

Example 2.0.2 (Max-flow)**.** *Suppose you have a network of pipes and receive money based on the amount of a valuable liquid that reaches a single destination, coming from a single source. The pipes have limited capacity and, given that it is a liquid, none of it is gained or lost along the way. This is the max-flow problem, whose linear programming model follows.*

Definition 9. *Let $G(V, E)$ be a graph with $s, t \in V$ being defined as the source and target. The* ***max-flow problem*** *is the following LP:*

$$\begin{aligned} \max \quad & \sum_{v:(s,v)\in E} f_{sv} \\ s.t. \quad & f_{uv} \leq c_{uv}, \forall (u,v) \in E \\ & \sum_u f_{uv} - \sum_w f_{vw} = 0, \forall v \in V \setminus \{s,t\} \end{aligned} \tag{2.3}$$

Example 2.0.3. *A furniture company makes desks, tables, and chairs. The making of each type of furniture requires wood and two types of specialized work: finishes and carpentry. The quantity required of each of these resources for each of the types of furniture is the following:*

Resource	Desks	Tables	Chairs
Wood	8 boards	6 boards	1 board
Finishes	4 hours	2 hours	1.5 hours
Carpentry	2 hours	1.5 hours	0.5 hours

The company can expend 48 boards, 20h of finishes, and 8h of carpentry. The selling price is 60€ for desks, 30€ for tables, and 20€ for chairs. What is the production plan that maximizes the revenue?

```python
from pyscipopt import Model

m = Model()
d = m.addVar(lb=0)                          # desks
t = m.addVar(lb=0)                          # tables
c = m.addVar(lb=0)                          # chairs
m.addCons(8*d + 6*t + c <= 40)              # boards
m.addCons(4*d + 2*t + 1.5*c <= 20)          # finishes
m.addCons(2*d + 1.5*t + 0.5*c <= 8)         # carpentry

m.setObjective(60*d + 30*t + 20*c, "maximize")
m.optimize()
```

Listing 2.2: Furniture LP in PySCIPOpt

2.1 Duality

Definition 10. *The dual problem of an LP of the form 2.1 (which we now call the primal) is:*

$$\begin{aligned} \max_{y} \quad & b^\intercal y \\ s.t. \quad & A^\intercal y \geq c \\ & y \geq 0 \end{aligned} \tag{2.4}$$

Remark 2.1.1. *The dual of the dual problem is the primal problem.*

Every constraint in the primal problem is associated with a variable in the dual, and vice-versa. The value of a dual variable can be seen as the change in the objective value by relaxing the corresponding constraint by one unit. In the case where the primal problem is seen as a resource-allocation problem, the dual also has a real-world economical interpretation, called the *shadow price*. See the following example, where we exemplify and expand upon this notion.

Example 2.1.2. *Recall Exercise 2.0.3, with an LP for maximizing the revenue of a furniture store. The dual of the LP follows.*

$$\begin{aligned} \min_{y} \quad & 40u + 20v + 8w \\ s.t. \quad & 8u + 4v + 2w \geq 60 \\ & 6u + 2v + 1.5w \geq 30 \\ & u + 1.5v + 0.5w \geq 20 \\ & u, v, w \geq 0 \end{aligned} \tag{2.5}$$

The variable u is associated with the first constraint, with the limitations on the boards. The value of u tells us the effect that increasing the number of available boards by 1 in the original problem would have on the revenue. To see this in another light, would be to say that the value of u is the maximum amount of money that the decision maker will be willing to spend to buy an additional board.

The dual can also be seen as obtaining the linear combination of constraints that gives the best lower bound of the primal.

Consider the following LP:

$$\begin{aligned} \max \quad & c_1 x_1 + \cdots + c_n x_n \\ \text{s.t.} \quad & A_1 x_1 + \ldots A_n x_n \leq b \\ & x \geq 0 \end{aligned} \tag{2.6}$$

For a given $n \times m$ vector y, if we have that $A_i^\intercal y \geq c_i, \forall i$, then the sum of all $A_i^\intercal y$'s is an upper bound to the objective function of the original problem. This gives us the constraints

$$\begin{aligned} & A_1^\intercal y_1 \geq c_1 \\ & \quad \vdots \\ & A_n^\intercal y_n \geq c_n \\ & y \geq 0 \end{aligned} \tag{2.7}$$

Equivalently, we can simply write $A^\intercal y \geq c, y \geq 0$, precisely the constraints of the dual problem. We want to find the lowest upper bound we can find, and we know that $(A_i x_1 + \ldots A_n x_n)^\intercal y \leq b^\intercal y$, so now we arrive at the objective function of the dual, $\min b^\intercal y$, arriving at the dual presented in 10.

Proposition 2.1.3 (Weak Duality for LPs)**.** *Given an LP P and the corresponding dual D, we have that* $b^\intercal y \leq c^\intercal x$.

Theorem 2.1.4 (Strong Duality for LPs)**.** *Let P be an LP, D the corresponding dual, and* x^*, y^* *be the respective optimal solutions. We have that* $b^\intercal y^* = c^\intercal x^*$.

The strong duality theorem is very useful for knowing whether a point x is optimal or not.

Proposition 2.1.5. *Let P be an LP and D the corresponding dual. We have the following:*

1. *D is infeasible $\implies$ P is unbounded*
2. *D is unbounded $\implies$ P is infeasible*
3. *D has an optimal solution y^* $\implies$ P has an optimal solution x^* and $b^\intercal y \leq c^\intercal x^*$*

Proof. Statement 2 follows directly from statement 1 and the remark that the dual of the dual is the primal. Statement 3 follows directly from the weak duality theorem. So now all we have to do is prove statement 1.

Let P be a primal problem, and D its unbounded dual. Suppose that P has a feasible solution x. By the weak duality theorem, then we get that $c^\intercal x \geq b^\intercal y$ for all feasible solutions y. But then D is not unbounded, and we have a contradiction. □

The value of the dual variables also gives us important information about the constraints of the original problem. This information is called the *complementary slackness* conditions. Continuing with the economic interpretation of the dual, suppose that the optimal value of a dual variable is 0. Then this means that no benefit would come from relaxing the corresponding constraint, which implies that, at optimality, this constraint is not tight, but is satisfied with "room to spare".

2.2 Big-O notation

In order to compare algorithmic performance from a theoretical point of view, we will anticipate some notions we will use later in the Complexity Theory chapter, the Big-O notation.

Definition 11. *Given functions $f, g : \mathbb{R}^n \to \mathbb{R}$, we say that $f \in O(g(x))$ if*

$$\forall x \geq x_0, |f(x)| \leq Mg(x) \tag{2.8}$$

This is used in the description of the *worst case* asymptotic behavior of algorithms, where the function $f(n)$ is the number of steps of an algorithm for an input of size n. Informally, we can also say that the algorithm takes $O(f(n))$ time.

Example 2.2.1. *Suppose you want to write every number from 1 to n. Any algorithm that does this will be at least $O(n)$*[1]*, since it will take at least n iterations. If, on the other hand, you wanted to write all the elements of the power set of size n, then that algorithm will be at least $O(2^n)$, as you will have to write out every element of the power set, and you are forced to go through every one of them.*

With very simple algorithms, it may happen that its asymptotic behavior is the same across all families of inputs. When that is not the case, when the algorithm runs more poorly on one set of inputs than another, then we need to define which complexity we are referring to. When studying algorithms from a theoretical point of view, it is more common to talk about their worst-case complexity, meaning the complexity of the more difficult instances. Average-case complexity is more prevalent in practical applications of these algorithms. It may happen that they have a very poor worst-case complexity, but the instances that incur that complexity are rare, and it runs well in most cases. In the next section, we will talk about both. The best-case complexity can also be studied, but its interest is very limited.

2.3 Simplex Method

The simplex method is one of the oldest in linear optimization, dating back to 1947 with George Dantzig. Its purpose is to optimally solve LPs.

Definition 12. *A point x of a convex set P is an* ***extreme point*** *if it cannot be expressed as a convex combination of two other points in P.*

[1]Here I say at least because there is no information on the complexity of the actual writing process. Most likely it will be $O(log(n))$, given that writing a number in the standard way scales directly with the number of digits.

Definition 13. *A point x is a **vertex** of polyhedron P if $x \in P$ and there are n linearly independent constraints such that $A_i x = b$.*

Theorem 2.3.1. *Let $P = \{x \in \mathbb{R}^n \mid Ax \leq b$ be a non-empty polyhedron with $A \in \mathbb{R}^{m \times n}$. Let $\overline{x} \in P$. Then,*

$$\overline{x} \textit{ is an extreme point} \iff \overline{x} \textit{ is a vertex}$$

Definition 14. *A polyhedron P contains a **line** if $\exists x \in P, d \in \mathbb{R}^n \mid x + \lambda d \in P, \forall \lambda \in \mathbb{R}$.*

Proposition 2.3.2. *Let P be a polyhedron. Then the following statements are equivalent:*

1. *P does not contain a line.*
2. *P has an extreme point.*

Theorem 2.3.3. *Consider the following LP:*

$$\begin{aligned} \min_{x} \quad & c^\intercal x \\ s.t. \quad & Ax \leq b \quad (P) \\ & x \geq 0 \end{aligned} \tag{2.9}$$

Suppose it has at least an extreme point. Then, if an optimal solution exists, there is also an optimal solution at a vertex.

Proof. Let Q be the set of optimal solutions, non-empty by assumption, and let v be the optimal value. By Proposition 2.3.2, P has no lines, which implies that Q has an extreme point (given that it is a subset of P. Let x^* be an extreme point of Q. Suppose that it is not an extreme point of P. Then $\exists y \neq x^*, z \neq x^*, \lambda \in [0, 1]$, s.t. $x^* = \lambda y + (1 - \lambda)z$. Multiplying by $c^\intercal$ on both sides, we get $v = c^\intercal x^* = \lambda c^\intercal y'(1 - \lambda)c^\intercal z$. Given the optimality of v, we know that $c^\intercal y \geq v, c^\intercal z \geq v$. Using this and the previous equality, we get that $c^\intercal y = v, c^\intercal z = v \implies y \in Q, z \in Q$, which implies that x^* is not an extreme point of Q. Thus, we arrived at a contradiction. $\square$

This theorem is very important since it proves that an algorithm to solve LPs only needs to search the vertices of its polyhedron, which can be computationally easier. Furthermore, we know that a vertex x is an optimal solution if all adjacent vertices have worsening solutions. This can be seen by considering that otherwise, we would have a local optimum that is not also a global optimum. Given that linear programs are convex, Chapter 1 gives us that this is impossible.

The first algorithm for solving LPs that we will discuss, the simplex method, takes advantage of the above result. It will iteratively visit adjacent vertices of the polyhedron until it reaches an optimal solution. We know that we have reached an optimal extreme point if none of its adjacent vertices improve the objective.

The simplex method requires a reformulation of the LP problem in 2.1, called the *standard form*:

$$\begin{aligned} \min_{x,s} \quad & c^\intercal x \\ \text{s.t.} \quad & A^\intercal x + s = c \\ & x, s \geq 0 \end{aligned} \tag{2.10}$$

The standard form is characterized by being a minimization problem with equality constraints (except for the bounds). Given an inequality constraint $A \intercal x \leq b$, we can add a variable s, called a *slack variable*, to turn it into $A^\intercal x + s = b$. The slack variables are artificial variables whose value in a particular solution will let us know which constraints are tight in the original model, and which are not.

This version of the Simplex has some problems, namely the possibility of cycling through a subset of vertices without finding the optimal solution. This can happen if we find vertices with the same objective. For that reason, the revised Simplex algorithm is more common in practice.

As with all the algorithms that we will discuss in this report, there are many variants in the literature. We will disregard them for brevity's sake but may name some of them if they present short and valuable insights. For example, many of the algorithms have a Primal-Dual version, including the Simplex Method.

Instead of focusing solely on the primal, these primal-dual methods solve both the primal and its dual at the same time. Apart from the possibility of speeding up the algorithm by switching from one problem to the other, we gain information regarding the distance to optimality, since we have an upper bound by computing the difference between the dual and the primal.

For the vast majority of problems, the Simplex Method runs in polynomial time, but so-called pathological examples have been found that ensure that the Simplex Method has to visit an exponential number of vertices. So, in theory, its complexity is $O(2^n)$.

2.4 Interior Point Methods

Interior Point Methods (IPM) are alternative methods to the simplex that have become the standard in many applications. Some specific problems have a structure that is better exploited by the Simplex, but they are few. Complexity-wise, IPMs are better than Simplex, with a polynomial running time ($O(n^{3.5}log(1/\varepsilon))$, in fact). They also have the added bonus of being usable in nonlinear problems. The main idea is to remove restrictions from the problem, but penalize solutions that get close to violating them. The penalization gets iteratively softer, making the solution closer and closer to the optimum of the original LP.

$$\begin{aligned} \min_x \quad & c^\intercal x \\ \text{s.t.} \quad & c_i(x) \geq 0, i = 1, \dots, m \end{aligned} \tag{2.11}$$

We replace the constraints with what is called a *barrier function*, most commonly a logarithm.

$$\min_x \quad c^\intercal x - \mu \sum_{i=1}^{m} log(c_i(x)) \tag{2.12}$$

This gives us an unconstrained optimization problem, that penalizes solutions that approach the limits imposed by the constraints of the original problem. With each iteration, the value of μ gets progressively smaller, allowing solutions to get closer and closer to the boundary. In practice, the convergence is extremely quick and does not require many iterations.

In reality, interior point methods do not solve this unconstrained nonlinear problem but optimize instead for the derivative, relying on optimality conditions (KKT, see below) to stop the algorithm when a predefined tolerance is reached. Each iteration corresponds to a Newton step, pushing the solution closer and closer to the optimum. Doing this in unconstrained optimization can be very fast.

As a curiosity, we leave here the famous KKT conditions. The Karush-Kuhn-Tucker (KKT) conditions establish sufficient and necessary conditions to determine optimality at a point in an LP.

Theorem 2.4.1 (KKT conditions)**.** *Consider the LP 2.1 and let $x \in \mathbb{R}^n$. Then x is an optimal solution if and only if the following conditions are satisfied:*

1. *Primal feasibility:* $Ax \leq b, x \geq 0$
2. *Dual feasibility:* $\lambda A + v = c, \lambda \geq 0, v \geq 0$
3. *Complementary Slackness:* $\lambda(Ax - b) = 0, vx = 0$

2.5 Exercises

Modeling

In these exercises, I suggest first developing a mathematical programming model and then trying to implement it in PySCIPOpt.

Exercise 2.5.1. *A factory produces two different kinds of cloth, using* 3 *different colors of wool. For each meter of cloth, the following quantities of wool are necessary (in grams):*

The factory has in stock 100*kg of yellow wool,* 100*kg of green wool, and* 120*kg of black wool. The manager of this factory wants to determine how to establish the production, supposing that it profits* 500*e/m on cloth A and* 200*e/m on cloth B. Formulate its problem.*

Wool	Cloth A	Cloth B
yellow	400	500
green	500	200
black	300	800

Table 2.1: Wool requirements (in grams)

Exercise 2.5.2. *Consider the problem of finding the cheapest diet that satisfies nutritional requirements for an entire week (700% of the daily minimum). There are several available dishes, each with its nutritional value and cost. The nutritional value is presented in percentage (per serving) of the daily minimum dose of vitamins A, C, B1, and B2. The data follows:*

Dish	Price	A	C	B1	B2
Steak	370	60	20	10	15
Chicken	270	8	0	20	20
Apple	130	8	10	15	10
Hamburger	500	40	40	35	10
Macaroni	220	15	35	15	15
Veg. Pie	210	70	30	15	15
Rice	215	25	50	25	15
Codfish	170	60	20	15	10

Table 2.2: Available dishes and their price

Formulate the problem in linear programming. How would you adapt the model so that its result could be accomplished in the real world?

Exercise 2.5.3. *Note: This exercise is harder.*

An oil company produced three types of gasoline: G_1, G_2, and G_3. Each type of gasoline is produced using three types of crude oil: C_1, C_2, and C_3. The selling price of each barrel of gasoline, and the buying price of each barrel of crude oil are in the following tables.

Gasoline	Selling Price
G_1	70
G_2	60
G_3	50

Table 2.3: Gasoline selling price

Oil	Buying Price
C_1	45
C_2	35
C_3	25

Table 2.4: Oil buying price

The company has up to 5000 *daily barrels of crude oil available. The three kinds of gasoline differ in their contents in sulfur and octane index. The oil mix used to produce each type of gasoline should have the minimum octane index and the maximum sulfur indicated in the table on the left. The indices of octanes and sulfur in each of the crude oils are indicated in the table on the right.*

It costs 4 *euros to transform a barrel of crude oil into gasoline, and the company can produce up to* 14000 *barrels of gasoline per day. The deliveries of the customers should be satisfied exactly (there are no stocks), and are of* 3000, 2000, *and* 1000 *barrels per day, for types of gasoline G_1, G_2, and G_3, respectively.*

The company can also invest in advertising, to stimulate orders. Each euro spent daily on advertisement of a particular kind of gasoline increases the demand for that kind of gasoline by 10 *barrels per day. (For example, if* 20 *euros per day are invested in an advertisement for gasoline G_2, then its orders will increase by* 200 *barrels per day). Formulate the linear problem that allows the company to maximize the daily profits (revenue − costs). Bonus: Solve it using PySCIPOpt.*

	Octane (min.)	Sulfur (max.)
G_1 mixture	10	1.0%
G_2 mixture	8	2.0%
G_3 mixture	6	1.0%

Table 2.5: Gasoline mixture requirements

	Octane	Sulfur
C_1	12	0.5%
C_2	6	2.0%
C_3	8	3.0%

Table 2.6: Crude oil characteristics

Duality

Exercise 2.5.4. *Consider the following LP:*

$$\begin{aligned} \min \quad & 3x + 4y \\ s.t. \quad & x + y \geq 3 \\ & -3x + 4 \leq 2 \end{aligned} \tag{2.13}$$

What is its dual? Solve both problems geometrically, with an algorithm similar to the Simplex method.

Exercise 2.5.5. *Prove the weak duality theorem.*

Complexity

Exercise 2.5.6. *Given two numbers a and b, provide an algorithm that adds them which takes $O(\min(log(a), log(b))$ time, and another that takes $O(\max(a, b))$ time.*

Exercise 2.5.7. *Provide an algorithm for multiplying two numbers with sizes a and b that takes $O(\max(a, b))$ time.*

Exercise 2.5.8. *Calculate the time complexity of the standard algorithm for multiplying 2 numbers a and b.*

Exercise 2.5.9. *Describe an algorithm for multiplying two numbers a and b that has $O(1)$ time complexity. Hint: Imagine you are an elementary school student.*

Exercise 2.5.10. *Calculate the time complexity of the Laplace expansion for calculating the determinant of a matrix.*

Chapter 3

Complexity Theory

This chapter can be seen as more of a curiosity and an introduction to problems that we will talk about in later chapters but can be skipped without missing much.

3.1 Complexity Classes

Skipping over a lot of details, complexity classes are sets of problems characterized by their difficulty. The two most famous classes are **P** and **NP**. The first is the set of problems that can be solved by an algorithm that runs in polynomial time. Given a problem p, $p \in \mathbf{P}$ if there exists an algorithm $\mathcal{A}$ that solves it, such that the running time of algorithm $\mathcal{A}$, $T_{\mathcal{A}} \in O(n^k)$, for some natural k. On the other hand, a problem $p \in \mathbf{NP}$ if there is an algorithm $\mathcal{A}$ that verifies if an input x is a solution to problem p such that $T_{\mathcal{A}} \in O(n^k)$ for some natural k. These notions will be used in the context of optimization. Using the language that we have used so far, we say that an optimization problem p is in **NP** if we can say if a vector x is feasible for p in polynomial time.
Naturally, $\mathbf{P} \subseteq \mathbf{NP}$, since by solving the problem in polynomial time, we can verify a solution in polynomial time. The other inclusion is a famous open problem, but the general sentiment is that it does not hold.

There are hundreds of different computational classes. Complexity Zoo is a wiki archiving them, and it has reached 546 at the time of writing (01/2023), but some have received considerably more attention over the decades.

3.2 Famous problems in NP

As with complexity classes, there are also hundreds of problems known to be **NP**-complete (the hardest problems in **NP**, expanded in the next section).

- SAT. Given a boolean expression, is there an assignment such that it evaluates to TRUE?
- TSP. Given a graph, what is the shortest Hamiltonian cycle?
- Knapsack. Given a bag and items with weight and value, what is the most value we can carry?
- Generalized Sudoku. Sudoku on an $n \times n$ grid.

We will now describe some **NP**-complete problems that will be useful in the following Chapters, starting with Vertex Cover.

Vertex Cover Problem

Given a graph $G(V, E)$, a vertex cover V' is a subset of V such that $\forall (uv) \in E, u \in V' or v \in V'$. The vertex cover problem tries to answer the following question "Is there a vertex cover $V' such that |V| \geq k$?"

Many decision problems can be easily converted into optimization problems. In vertex cover, we can formulate a problem that tries to minimize the number of vertices in V', thus creating what is called the Minimum Vertex Cover. If this number is less than or equal to k, then the answer to the original vertex is "Yes", otherwise it is "No". Every optimization problem, on the other hand, can be seen as a decision problem. Given an optimization problem with an objective $f(x)$, we can always ask if there is an x_0 such that $f(x_0) = x^*$. From now on, we will focus on the problems from an optimization point of view. Additionally, even though we have not defined what it was, we will formulate these problems as integer programs. Think of linear programs with the possibility of some (or all) of the variables being integers.

Knapsack problem

The knapsack problem asks which weighed items with an associated price should be taken, knowing that we have a knapsack with limited capacity and we want to maximize the total value.

$$\begin{aligned} \max \quad & \sum_{i=1}^{n} v_i x_i \\ \text{s.t.} \quad & \sum_{i=1}^{n} w_i x_i \leq W \\ & x_i \in \{0,1\} \end{aligned} \tag{3.1}$$

Cutting-stock Problem

The cutting-stock problem is usually seen as the minimization of the number of sheets of metal that need to be cut in order to satisfy the demand. The demand consists of a series of orders of different-sized sheets.

$$\begin{aligned} \min \quad & \sum_{j=1}^{M} y_j \\ \text{s.t.} \quad & \sum_{j=1}^{M} x_{ij} = d_i, & i \in [n] \\ & \sum_{i=1}^{n} l_i x_{ij} \leq L, & j \in [m] \\ & x_{ij} \leq d_i y_j, & i \in [n], j \in [m] \\ & x_{ij} \in \mathbb{Z}^+, & i \in [n], j \in [m] \\ & y_j \in \{0,1\}, & j \in [m] \end{aligned} \tag{3.2}$$

Traveling Salesman Problem

Perhaps the most famous optimization problem is the Traveling Salesman (TSP), where the objective is to find the least costly Hamiltonian cycle in a directed graph with weighted edges.

$$
\begin{aligned}
\min \quad & \sum_{i=1}^{n} \sum_{j\neq i, j=1}^{n} c_{ij} x_{ij} \\
\text{s.t.} \quad & \sum_{i\neq j, i=1}^{n} x_{ij} = 1, j \in [n] \\
& \sum_{i\neq j, j=1}^{n} x_{ij} = 1, i \in [n] \\
& \sum_{i\in Q} \sum_{j\neq i, j\in Q} x_{ij} \leq |Q| - 1, \forall Q \subsetneq \{1, \ldots, n\}, |Q| \geq 2 \\
& x_{ij} \in \{0, 1\}
\end{aligned}
\tag{3.3}
$$

A variable x_{ij} should be equal to 1 if and only if the edge (ij) is present in the path. The objective function calculates the cost of the tour, while the first constraint stipulates that every vertex must have exactly one destination, while the second states that every vertex must be the destination of exactly one other vertex. The third constraint, more complicated, is there to prevent sub-cycles. In other words, to ensure that the result forms a connected component.

The TSP gets its name because it usually comes up as finding the shortest possible route for a vehicle to visit a list of cities exactly once while returning to the original city. Despite this, it has numerous applications in different areas.

3.3 Reductions

A big area in the study of Complexity Theory is the reduction of optimization problems from one to the other, often with the objective of indirectly proving that a certain problem is in a particular complexity class.

Definition 15. *Given two problems P and Q, we say that P is reducible to Q if by solving Q we are able to quickly solve P. We say that $P \leq Q$.*

This notion tries to relate the asymptotic difficulty of two problems. If we find a reduction from P to Q, then intuitively, we can say that Q is at least as hard as P. Suppose that we can solve Q very quickly. Then with this reduction, we are also able to solve P very quickly. If, on the other hand, we know that P is very hard to solve, then we have indirectly proven that Q is very hard as well.

Definition 16. *A problem Q is said to be **NP**-hard if $\forall P \in \textbf{NP}, P \leq Q$. Additionally, if $Q \in \textbf{NP}$, we say that it is **NP**-complete.*

If we know that problem Q is **NP**-hard and that it is reducible to P, then we also know that P is NP-hard. See the following example:

Proposition 3.3.1. *Integer programming is **NP**-hard.*

Proof. To achieve this, we will present a reduction from Vertex Cover to Integer Programming. Let $G = (V, E)$ be an undirected graph. Consider the following integer program.

$$
\begin{aligned}
\min \quad & \sum_{v\in V} y_v \\
\text{s.t.} \quad & y_v + y_u \geq 1, \qquad uv \in E \\
& y_v \in \mathbb{N}_0, \qquad v \in V
\end{aligned}
\tag{3.4}
$$

Since y_v is either 0 or 1, any feasible solution corresponds to a subset of vertices, and it is clear that finding the optimal solution to this integer program corresponds to finding a minimum vertex cover. Thus, integer programming is **NP**-hard. □

Naturally, a decision problem is easier than its optimization counterpart. Intuitively, it is easier to find an element that satisfies a set of conditions, than it is to find the best element that satisfies that set of conditions (where best is defined by the objective function). This is reflected complexity-wise, as the optimization counterpart of the **NP**-complete decision problems we discussed are **NP**-hard.

There is a very famous open problem in Complexity-Theory asking whether **P**=**NP**. To prove this equality, one could find a reduction from a problem in **P** to another that is **NP**-complete. Alternatively, the equality could also be proven if an algorithm for solving an **NP**-complete problem in polynomial time is invented. As far as I know, the majority of researchers think that the equality does not hold.

3.4 Exercises

Exercise 3.4.1. *An independent set of a graph $G(V, E)$ is a subset V' of V such that no two vertices in V' share an edge.*

Give a reduction from vertex cover to independent set.

Exercise 3.4.2. *Give a reduction from Cutting-Stock to knapsack.*

Exercise 3.4.3. *For the following list of problems, explain how you would use a solution of the ones discussed here to find a solution.*

- *In an exam, knowing the marks of each question and an estimate of the time it takes you to solve them, which should you attempt?*
- *You want to send a message ????*
-

Chapter 4

Integer Optimization

Integer Optimization focuses on the following set of problems:

$$\begin{aligned} \min_{x,y} \quad & c^\intercal (xy)^\intercal \\ \text{s.t.} \quad & f(x,y) \leq 0 \\ & x \geq 0 \\ & y \in \mathbb{Z} \end{aligned} \tag{4.1}$$

If x has dimension 0, we say that it is a Pure Integer Program (IP), otherwise, we refer to it as a Mixed-Integer Program (MIP). By default, f is assumed to be linear, but we can also define Mixed-Integer Linear Program (MILP) and Mixed-Integer Nonlinear Program (MINLP) accordingly.

4.1 Branch & Bound

The most popular method for solving integer programs is the Branch & Bound algorithm, which combines relaxations and variable fixings in order to arrive at an optimal solution.

For the following explanation, let us assume that the integer variables are all binary and that we have an ordered list of the binary variables, $x_1, \ldots, x_n$. We start by relaxing the integrality requirements of all variables so that we remain with an LP. As we have seen previously, LPs are very easy to solve, relative to MIPs. Since this is a relaxation, it serves as a lower bound to the original problem. Call it l^B. The next step is called branching, where we fix variable x_1 to 0 and 1 to create two subproblems. The integrality of the other variables remains relaxed. Solving one of the LPs, we get a lower bound, $l^B_{x_{1,0}}$. Remaining in this path, suppose we branch on variable x_2, solve the LP, get a bound, branch on x_3, etc., until we reach variable x_n and perform the final branching. With all the integer variables fixed to some value, the problem is easy, and we now get a solution to the integer problem, and hence an upper bound on the optimal solution. Call it $u^B_{x_{0,0}\cdots x_{n,0}}$. We can now turn to the first LP that we ignored, resulting from fixing x_1 to 1. Suppose we solve and it has a solution $l^B_{x_{1,1}}$ and that $l^B_{x_{1,1}} > u^B_{x_{0,0}\cdots x_{n,0}}$. Remember that the left-hand side is a lower bound to the problem resulting from fixing x_1 to 1. If the inequality holds, then we no longer need to further investigate what happens afterward, since any solution will be worse than the one we obtained.

A generic version of Branch & Bound (Land & Doig (1960)) follows:

This Branch & Bound algorithm assumes that the integer variables are binary for simplicity, but can easily be adapted to the more general integer case, by branching on the continuous variable x to $\lfloor x \rfloor$ and $\lceil x \rceil$.

There are variants to the algorithm we presented. Namely, regarding the node selection and the variable selection when branching. In line 3, it is taking a FIFO (first in, first out) strategy, which will give us a depth-first search. On line 8, we pick a non-binary variable x, but there are many of them. A common strategy is to pick the one "most fractional" one, the one closer to 0.5, that is. The intuition behind this heuristic is that if the linear relaxation gives a value for x that is 0.9999, then it is reasonable to assume that its value will be 1 in the original problem.

```
input  : IP $\min c^\intercal x, x \in X$ // assumed feasible and bounded
output: output: optimal solution $x^*$
$L \leftarrow \{X\}$ // List of unexplored subproblems
$\overline{z} = \infty$ // Upper bound
while $L \neq 0$ do
    select a subproblem $S$ from $L$ // search strategy
    solve LP relaxation of $S$, with solution $x\prime$ and objective $z'$, if they exist
    remove $S$ from $L$
    if $S$ infeasible or $z' \geq \overline{z}$ then
        continue
    if $x'$ is integer and $z' < \overline{z}$ then
        $\overline{z} \leftarrow z'$
        $x^* = x'$ // $x'$ becomes new best solution
        continue
    // otherwise S is not pruned, $x'$ is feasible and fractional
    split $S$ into subproblems $S_1, S_2$
    insert $S_1, S_2$ into $L$
return $x^*$
```

Algorithm 1: Branch & Bound

Example 4.1.1. *Let us look at the example found in geeks for geeks, of branch & bound in an instance of the* $0-1$ *knapsack. If needed, recall the knapsack problem in Chapter 3.*

In this example, the knapsack capacity W *is* 10*, and the items available, together with their weights and values can be found below.*

	A	B	C	D	E
Weight	2	3.14	1.98	5	3
Value	40€	50€	100€	95€	30€

Note: In the figure below there is a small mistake. On the first feasible leaf (corresponding to taking every item except E), the

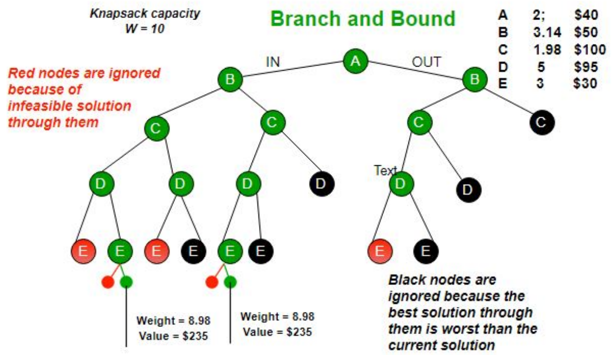

Knapsack capacity W = 10
Branch and Bound
A 2; $40
B 3.14 $50
C 1.98 $100
D 5 $95
E 3 $30
Red nodes are ignored because of infeasible solution through them
IN
OUT
A
B
B
C
C
C
C
D
D
D
D
D
D
Text
E
E
E
E
E
E
E
E
Weight = 8.98 Value = $235
Weight = 8.98 Value = $235
Black nodes are ignored because the best solution through them is worst than the current solution

Figure 4.1: Branch and Bound tree for the $0-1$ knapsack

4.2 Heuristics

Heuristics are algorithms whose objective is to obtain solutions cheaply. They may do this by solving easier restrictions of the problem or simply by not requiring an optimal solution. Given the difficulty of integer programming, they are widely used both in practical applications and in academia. Heuristics can also be

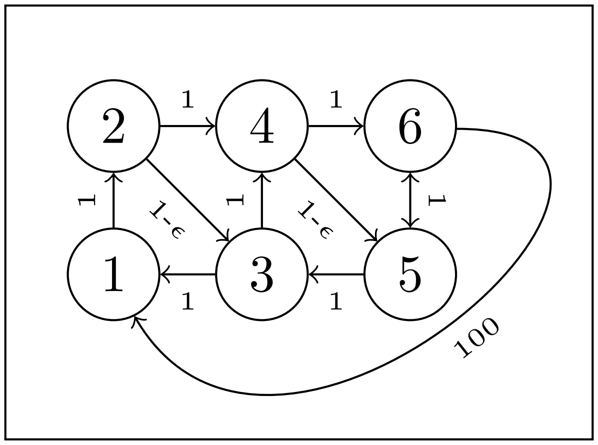


Figure 4.2: An example where nearest neighbor works poorly

very useful as a starting solution for other exact methods. For example, a heuristic can quickly provide a good bound for the Branch & Bound tree, allowing for early pruning.

In this Section, we will focus on heuristics for the TSP, which was introduced in 3.2.

Nearest Neighbor

Starting with a chosen vertex, choose the closest neighbor until no vertex remains.

While most novel work on heuristics tends to be underdeveloped in terms of theory, that is not true for all cases, where guarantees on approximation algorithms have been found. For example, in the variant of the TSP where the distance satisfies the triangle inequality, an algorithm based on minimum spanning trees was found to produce solutions that are at worst 2 times the optimal solution. It may seem quite bad, but considering the difference in time for very large instances, having this algorithm, especially with this guarantee, can be very valuable.

Definition 17. *A **minimum spanning tree** is a subset of the edges of a connected, edge-weighted undirected graph that connects all vertices, without any cycles, such that total edge weight is minimum.*

```
  Data: $G(V, E)$
1 construct a minimum spanning tree T of G with root on the TSP depot;
2 return a depth-first search on T;
```

Algorithm 2: 2-aprox algorithm for euclidean TSP

Proposition 4.2.1. *Algorithm 2 is at most twice as bad as the optimal solution. Additionally, it has a time complexity of $O(V^2)$, where V is the number of vertices.*

Proof. The assumption of the triangle inequality of the distance allows us to go from a leaf to the root node in at most the distance it took to get there.

It is easy to see that the complexity of the algorithm is tied to the complexity of finding the minimum spanning tree, which is known to be $O(V^2)$. □

4.3 Meta-Heuristics

As important as heuristics are meta-heuristics, dating their roots back to the 1970's (as far as I know). Instead of working directly with the problem and its variables, the search space of a meta-heuristic is the set of solutions themselves.

A very popular meta-heuristic is local search where, given a solution, we search its neighborhood by altering it slightly. For example, in a TSP solution, instead of doing $\cdots - A - B - C - \ldots$, we try $\cdots - C - B - A - \ldots$. This is called 2-OPT

The downside of local search is its tendency to converge to local optima, alongside the large difficulty in escaping it.

Tabu Search

The idea behind taboo search is to allow worsening solutions in order to avoid local optima. keep a list of recently visited solutions, called a tabu list, with forbidden solutions.

Below follows a generic python code for tabu search (from Wikipedia). In the code, the stoppingCondition() and getNeighbors() functions are self-evident. fitness() is a function that gives a number for the quality of a solution.

```python
sBest = s0
bestCandidate = s0
tabuList = []
tabuList.append(s0)
while not stoppingCondition():
    sNeighborhood = getNeighbors(bestCandidate)
    bestCandidate = sNeighborhood[0]
    for sCandidate in sNeighborhood:
        if (sCandidate not in tabuList) and (fitness(sCandidate) > fitness(bestCandidate)):
            bestCandidate = sCandidate
    if fitness(bestCandidate) > fitness(sBest):
        sBest = bestCandidate
    tabuList.append(bestCandidate)
    if len(tabuList) > maxTabuSize:
        del tabuList[0]
return sBest
```

Listing 4.3: Generic Python code for tabu search

Simulated Annealing

The basic idea is to assign a probability to neighbors of the current given solution, and randomly sample according to some distribution and the assigned probabilities. Neighbors with better solutions should be given higher probabilities. As the iterations increase, the probability of moving to another solution should converge to zero, converging to a solution. However, by still assigning a non-zero probability to worsening solutions, the chance of being stuck at a local optimum is decreased.

```
  Data: $s_0, k_{\max}$
1 Let $s = s_0, k = 0$;
2 while $k \leq k_{max}$ do
3     $P = Probability(1 - (k+1)/k_{\max}$;
4     Pick a random neighbor $s_{\text{new}}$ of $s$;
5     if $P(E(s), E(s_{new}, P) \geq random(0,1)$ then
6         $s = s_{\text{new}}$;
7 return $s$;
```

Algorithm 3: An algorithm for simulated annealing

Genetic Algorithm

Inspired by evolution in biology, a genetic algorithm is a local search meta-heuristic where on a given iteration (generation), a set of candidates (called a population), each with an associated fitness, creates "off-spring" by combining elements of the population. The elements with the highest fitness will have a higher chance of passing their characteristics to the next generation. ?????????????????????????

```
    Data: $k_{\max}$
1 create initial population $P_0$;
2 while $k \leq k_{max}$ do
3     get fitness of solutions;
4     keep the best individuals;
5     combine the parents to create off-spring;
6     mutate the off-spring;
7 return solution with highest fitness;
```

Algorithm 4: An algorithm for genetic algorithm

For a single problem, it is possible to use multiple heuristics, using heuristic A until we arrive at a solution x, and then try heuristic B. The decision of which heuristic to use at any given moment can also be done heuristically by what is called a *hyperheuristic*.

It is off-topic in this document, but there are a lot of Machine Learning meta-heuristics. There are neural networks, gaussian processes, genetic algorithms, etc.

4.4 Reformulations

An interesting aspect of optimization in general, but more prominent in integer optimization (also due to its complexity), is the concept of reformulations. Some problems can be modeled in different ways, that behave much differently and can lead to vastly different solving times.

Reformulations in integer optimization also appear naturally due to the same problem admitting infinitely many models. In general, we say that a formulation is better if its linear relaxation is closer to the convex hull of the integer problem. A formulation that achieves this equality is called an *ideal* formulation. The following chapter will discuss two special classes of reformulations.

4.4.1 Symmetry

Consider, for example, the generalization of the traveling salesman problem called the *vehicle routing problem*, which asks the following question: “”What is the optimal set of routes for a fleet of vehicles to traverse in order to deliver to a given set of customers?” If we assume that the vehicles in the fleet are identical, then permutating the vehicles leads to identical solutions, increasing the search space by $n!$. With the increase in difficulty with integer programming, compared with linear programming, having symmetric possible solutions can greatly worsen the runtime. Symmetry-breaking constraints try to deal with this problem. In the example of the vehicle routing problem, such a constraint could somehow attribute a number to each of the vehicles, and associate each route with a vehicle. Vehicle 1 is responsible for route 1, etc.

4.4.2 Logical constraints

The possibility of using integer variables also allows us to reformulate a lot of logical constraints, such as disjunctions, conjunctions, and negations, but also quantifiers (which allows us to model any first-order logic proposition). See the following table, where x and y are binary variables.

$\neg x$	$1 - x$
$x \implies y$	$x \leq y$
$x \wedge y$	$x + y = 2$
$x \vee y$	$x + y \geq 1$
$x \dot{\vee} y$	$x + y = 1$
$\exists x$	$\sum_{i=1}^{n} x_i \geq 1$
$\exists! x$	$\sum_{i=1}^{n} x_i = 1$

Table 4.1: Formulating logical expressions with integer programming

4.4.3 Cutting Planes

A widely studied type of reformulation are the *cutting planes*. They are commonly used in order to get stronger linear relaxations. For example, imagine that an integer program had the Constraint $y \leq 1.5$, and y is an integer. Then we can simply say that $y \leq 1$, and the resulting linear relaxation would be closer to the original integer program. Cuts like these are typically applied many times throughout the solving process of an integer program, sometimes reaching hundreds of thousands of cuts for larger problems.

They are both used independently, but also in conjunction with the Branch & Bound method, to create the Branch & Cut method. When branching on an integer variable (by fixing it to a specific value), it can be possible to add additional cuts, so that the linear relaxation provides a better bound. This can result in earlier pruning, saving valuable time.

Cutting planes are also of extreme importance in nonlinear optimization, where they are used to approximate nonlinear convex functions by half-spaces. This idea is supported by the supporting hyperplane theorem 1.0.1.

4.5 Exercises

Exercise 4.5.1. *What is the worst-case time complexity of the Branch & Bound we described?*

Exercise 4.5.2. *What is the worst-case time complexity of the nearest neighbor heuristic for TSP?*

Exercise 4.5.3. *Coach Night is trying to choose the starting lineup for the basketball team. The team consists of seven players who have been rated (on a scale of 1 = poor to 3 = excellent) according to their ball handling, shooting, rebounding, and defensive abilities. The positions that each player is allowed to play and the player's abilities are listed in the following table:*

Player	Position	Ball Handling	Shooting	Rebounding	Defense
1	G	3	3	1	3
2	C	2	1	3	2
3	G,F	2	3	2	2
4	F,C	1	3	3	1
5	G,F	3	3	3	3
6	F,C	3	1	2	3
7	G,F	3	2	2	1

Table 4.2: Team with players' positions and skills

The five-player starting lineup must satisfy the following restrictions:

(a) At least 4 members must be able to play guard, at least 2 members must be able to play forward, and at least 1 member must be able to play center.

(b) The average ball-handling, shooting, and rebounding level of the starting lineup must be at least 2.

(c) If player 3 starts, then player 6 cannot start.

(d) If player 1 starts, then players 4 and 5 must both start.

(e) Either player 2 or player 3 (or both) must start.

Given these constraints, Coach Night wants to maximize the total defensive ability of the starting team. Formulate an IP that will help him choose his starting team.

Exercise 4.5.4. *Nancy would like to reorganize the numbers in the following* 3×3 *table in such a way that the lines and columns sum up as follows:*

Formulate Nancy's problem.

1	10	5	11
10	5	10	12
1	5	1	25
21	16	11	

Job	t_i	d_i
1	5	8
2	5	4
3	5	12
4	8	16

Table 4.3: Jobs' required time and due date

Exercise 4.5.5. *Four jobs must be processed on a single machine. The time required ti to process each job and the date the job is due d_i are shown, for each job i, in this table:*

The delay of a job is the number of days after the due date that a job is completed (if a job is completed on time or early, the job's delay is zero). Use the branch and bound algorithm method to determine the order in which the jobs should be processed to minimize the total delay of the four jobs.

Chapter 5

Decomposition Methods

In this section, we will discuss techniques that take advantage of very large LPs, some of which require them to have a specific structure.

5.1 Column Generation

Column generation is a method for solving very large LPs, by starting with a subset of the variables (which correspond to a column in the model) and iteratively adding variables (generating columns) that improve the solution. To achieve this, the original model is divided into two models, the *master problem*, which is the original problem considering only a subset of its variables, and the *pricing problem*, which identifies new columns to be added to the master problem. When the pricing problem detects that there are no solutions that would improve the model, the algorithm stops.

Other alternatives exist, the column generation algorithm we will consider will use the *reduced costs* of the candidate variables. This notion could have been introduced in Section 2.3, as the simplex method uses it to arrive at candidate vertices, but given that it was not discussed in-depth, it is presented here.

Definition 18. *Consider the following LP:*

$$\begin{aligned} \min_x \quad & c^\intercal x \\ s.t. Ax &\leq b \\ x &\geq 0 \end{aligned} \tag{5.1}$$

The reduced cost vector can be computed as $c - A^\intercal y$, where y is the dual vector.

The reduced cost of a variable x will tell us the amount by which its corresponding coefficient in the objective function would have to decrease before x takes a non-zero value in the objective function.

Without loss of generality, we will assume a minimization problem, so if every variable has an associated positive reduced cost, it means there is no variable that would improve the objective of the master problem, and thus we have reached at the optimal solution.

```
Data: $X'$
1 Initialize master problem and pricing problem;
2 while TRUE do
3     Solve master problem with $X'$;
4     Get dual values $\pi$ from master;
5     Solve the pricing problem;
6     if Optimal solution from pricing $\leq$ 0 then
7         add column to $X'$;
8     else
9         break;
```

Algorithm 5: Column generation

When implementing a column generation algorithm, some considerations may greatly improve performance. For example, solving the pricing problem to optimality in every iteration may prove to be a waste of resources. Theoretically speaking, we only ever need to solve the pricing problem to optimality once, precisely to prove that the smallest reduced cost is non-negative, and so the RMP is optimal in relation to the MP. Thus, heuristically solving the pricing problem will often be a good idea.

Another idea that may significantly improve performance, is to add more than one column per iteration. As long as their reduced cost is negative, they should improve the solution. Furthermore, in the situations where it makes sense to use column generation (when we have a very large number of variables), the master problem is much smaller and it should be easy to solve, no matter the (reasonable) number of columns we add. For this reason, column generation tends to work best in cases where one knows how to solve the pricing problem easily.

5.2 Dantzig-Wolfe Decomposition

Dantzig-Wolfe decomposition is often confused with column generation, but they are two different concepts. Dantzig-Wolfe is a reformulation of the LP that uses the Minkovski-Weil Theorem, which says that any point in a convex set can be represented as a convex combination of its extreme points and extreme rays. We only now present the theorem because only now do we use it.

Definition 19.

Theorem 5.2.1 (Minkovski-Weyl)**.** *Let P be a bounded convex polyhedron, $x_1, \dots, x_n$ its extreme points, and $x \in P$. Then*

$$\exists \lambda_1, \dots, \lambda_n \mid \sum_{i=1}^{n} \lambda_i = x \wedge \sum_{i=1}^{n} \lambda_i = 1 \tag{5.2}$$

This technique assumes that the LP has the following structure:

$$\begin{aligned} \min_x \quad & c^\intercal x \\ \text{s.t.} \quad & Ax \geq b \\ & Dx \geq d \\ & x \geq 0 \end{aligned} \tag{5.3}$$

Typically, this division is done on the “hardness” of the constraints. Suppose that the LP would be easy to solve if $Ax \geq b$ would be removed, and that is not true for the constraints $Dx \geq d$. We will now deduce the reformulation.

Let P be the bounded polyhedron defined by 5.4, and $X = x_1, \dots, x_{|X|}$ be the set of extreme points. From 5.2.1, we know that every point $x \in P$ can be expressed as

$$\begin{aligned} x = & \sum_{i \in X} \lambda_i x_i \\ & \sum_{i \in X} \lambda_i = 1 \\ & \lambda_i \geq 0, x \in X \end{aligned}$$

Substituting in 5.4 and rearranging the terms, we get:

$$
\begin{aligned}
\min \quad & \sum_{i \in X} \lambda_i c^\intercal x_i \\
\text{s.t.} \quad & \sum_{i \in X} \lambda_i A x_i \geq b \\
& \sum_{i \in X} \lambda_i D x_i \geq d \\
& \sum_{i \in X} \lambda_i x_i \geq 1
\end{aligned}
\tag{5.4}
$$

This reformulation, equivalent to the initial problem, is called the *master problem.*

However, the set X is exponential is very large, making this problem even more difficult that the previous one. That is unless we use delayed-column generation to solve it. With this formulation, we start with a small subset of X and alternate between solving the RMP to get the dual variables and solving the subproblem to keep adding new patterns to the RMP. The same as before, this process repeats until the reduced costs are found to be non-negative.

Despite being developed with LPs in mind, Dantzig-Wolfe decomposition can be applied to nonlinear problems, as long as the nonlinearities can be separated into the pricing problem.

A structure that usually benefits from Dantzig-Wolfe decomposition is the following:

$$
\begin{array}{lllll}
\min\limits_{x} & c^\intercal x & & & \\
\text{s.t.} & A_{01}x_1 + \cdots + A_{0n}x_n & & & = b_0 \\
& A_1 x_1 & & & \leq b_1 \\
& & \ddots & & \vdots \\
& & & A_n x_n & \leq b_n \\
& x \geq 0 & & &
\end{array}
\tag{5.5}
$$

This is called a block-diagonal structure, and the first row of constraints is called the coupling or complicating constraints. That is because if they were not present, then there would be nothing connecting the different blocks, and they could be solved independently of one another, resulting in a series of much simpler problems. See 5.1 for a visual representation of the structure.

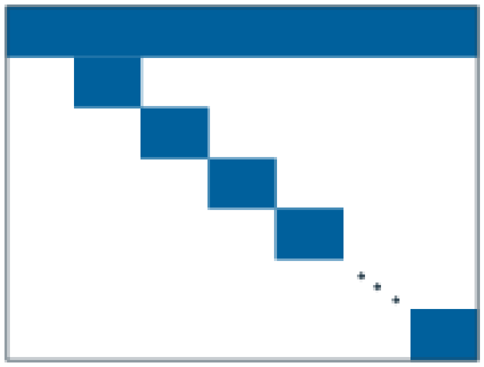


Figure 5.1: Block diagonal structure

Problems with this structure are very common in practical applications. Think of a set of machines working in parallel that must satisfy a common goal, for example.

Instead of looking at the individual variables, the reformulation will look at the potential solutions of the original problem, and pick a convex combination of them. Consider an example with two variables x and y of the initial formulation. The variables of the reformulation will be pairs (x_1, y_1), such that they satisfy the original constraints. The unrestricted solution space is $X \times Y$, with X and Y being the unrestricted solution spaces of x,y respectively.

5.2.1 Strength of the reformulation

5.3 Benders' Decomposition

Benders decomposition requires problems with variables that can be decomposed in two different sets. Afterward, it will reformulate the problem so that it has an outer problem and an inner problem, each concerned with one set of variables. Ideally one would want that a problem such that fixing the variables of the outer problem makes the inner problem easy to solve. The main idea behind Benders' is to iteratively fix the variables of the outer problem to certain values and guide them toward optimality.

Suppose we have the following MIP:

$$\begin{aligned} \min \quad & c^\intercal x + d^\intercal y \\ \text{s.t.} \quad & Ax + By \geq b \\ & y \in Y \\ & x \geq 0 \end{aligned} \tag{5.6}$$

Notice that if we fix the y variable we have an LP, that after some rearrangement has the following dual:

$$\begin{aligned} \max \quad & (b - B\overline{y})^\intercal u + d^\intercal \overline{y} \\ \text{s.t.} \quad & A^\intercal u \leq c \\ & u \geq 0 \end{aligned} \tag{5.7}$$

Recall Theorem 2.1.4 on the strong duality of LPs. The optimal solution of 5.7 is equal to that of 5.6, for the same fixed $\overline{y}$. Again, the objective now is to find the $\overline{y}$ that is the greatest maximizer for 5.7. Thus, equivalently to 5.6 we can reach the min-max problem below:

$$\min_{y \in Y} \quad d^\intercal y + \max_{u \geq 0} [(b - By)^\intercal u \mid A^\intercal u \leq c] \tag{5.8}$$

The minimization problem on y is called the outer problem, and the maximization problem on u is called the inner problem. Benders' decomposition will iteratively choose different values of y, and based on the results of the inner problem, add cuts to the outer problem.

For this next paragraph, recall Section 2.1 on duality. After solving the inner problem, we have three possibilities.

Inner problem is infeasible

This means that the primal problem is unbounded, and thus we can terminate the process immediately.

Inner problem is unbounded

Then we can infer that the primal is infeasible, and so we need to exclude the y we fixed on the outer problem. The infeasibility occurs because the choice $\overline{y}$ does not satisfy $Ax + B\overline{y} \geq b$. To counteract this, we add what is called a *feasibility cut*, $(b - By)^\intercal \overline{u} \leq 0$.

Inner problem has a finite optimal solution

From this, we know that we have a lower bound for the original problem (since we obtained an optimal solution for a giver $\overline{y}$). Then we add an *optimality cut* establishing a lower bound for the objective, $z \geq (b - By)^\intercal \overline{u} + d^\intercal y$.

The master problem associated with Benders at iteration k:

$$\begin{aligned} \min_{z} & \\ \text{s.t.} \quad & z \geq (b - By)^\intercal u^{(k)}, k = 1 \dots K \\ & u \geq 0 \end{aligned} \tag{5.9}$$

```
   Data: $\varepsilon$
1  while $UB - LB \geq \varepsilon$ do
2      solver inner problem;
3      if If it is infeasible then
4          break;
5      else if If it is unbounded
        then
6          add feasibility cut;
7      else
8          add optimality cut;
9          update UB;
10     solve outer problem;
11     update LB;
```

Algorithm 6: Benders' Decomposition

5.4 Branch & Price

These methods are very useful for solving specific LPs efficiently, so naturally, the next step is to try to use them to solve MIPs. The way this is usually done is by incorporating them into a branch-and-bound tree. Solving the MP to optimality may leave us with fractional solutions, so just like in Section 4.1, we branch and keep solving the problems in the tree by column generation, for example, until we find a solution that satisfies the integrality requirements.

Recall the cutting-stock problem, introduced in section 3.2, and consider its linear relaxation:

$$\begin{aligned} \min \quad & \sum_{j=1}^{M} y_j \\ \text{s.t.} \quad & \sum_{j=1}^{M} x_{ij} = d_i, i \in [n] \\ & \sum_{i=1}^{n} l_i x_{ij} \leq L, j \in [m] \\ & x_{ij} \leq d_i y_j, i \in [n], j \in [m] \\ & x_{ij} \in \mathbb{Z}^+, i \in [n], j \in [m] \\ & y_j \in 0, 1, j \in [m] \end{aligned} \tag{5.10}$$

In PySCIPOpt, we have the following code:

```python
from pyscipopt import Model, quicksum

x = {}
for i in I:
    x[i] = m.addVar(vtype="INTEGER")

for j in J:
    m.addCons(q[j] <= quicksum(a[i,j]*x[i] for i in I)
    model.setObjective(quicksum(c[i]*x[i] for i in I))
    model.optimize()
```

Listing 5.4: Cutting-stock in PySCIPOpt

5.5 Exercises

Exercise 5.5.1. *Recall the Diet problem exercise, initially presented in 2.5. Consider the modified problem of .... Suppose that you are a nutritionist in charge of defining the diets of* 3 *types of people, and want to create a shopping list that will take care of the dietary needs of each. You must keep in mind that they will all buy their groceries at a small local food store, and so you will need to take into account the available stock. Each item in the store has specific nutrients. The objective will be to minimize the total shopping costs. Formulate an LP (preferably in PySCIPOpt) amenable to a Dantzig-Wolfe reformulation to solve this problem with the information below:*

$$A = \begin{array}{|c|c|c|c|c||c|} \hline & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 1 & 2 & 1 & 2 & 1 \\ 2 & 3 & 4 & 2 & 2 & 1 \\ 3 & 3 & 4 & 2 & 2 & 1 \\ \hline \end{array}$$

, the available items and their nutrients and cost.

$b_0 = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, b_1 = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, b_2 = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, b_3 = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$*, with the available stock and the dietary needs of each type of person.*

Exercise 5.5.2. *Write the Dantzig-Wolfe reformulation of the above LP.*

Exercise 5.5.3. *In Benders' decomposition, why is the inner problem the dual in relation to a fixed* $\overline{y}$ *for Model 5.6? Why not use the primal directly?*