LINEAR OPTIMISATION NOTES

A COMPILATION OF LECTURE NOTES
FROM GRADUATE-LEVEL
OPTIMISATION COURSES

WRITTEN BY
FABRICIO OLIVEIRA

ASSOCIATE PROFESSOR OF OPERATIONS RESEARCH
AALTO UNIVERSITY, SCHOOL OF SCIENCE

SOURCE CODE AVAILABLE AT
github.com/gamma-opt/linopt-notes

Preface

These notes comprise the compilations of lecture notes prepared for teaching Linear Optimisation and Integer Optimisation at Aalto University, Department of Mathematics and Systems Analysis, since 2017.

These notes are mostly based on these references:

- *Bertsimas, Dimitris, and John N. Tsitsiklis. Introduction to linear optimization. Vol. 6. Belmont, MA: Athena Scientific, 1997.*
- *Wolsey, Laurence A. Integer programming. John Wiley & Sons, 2020.*

Over the years, errors and typos have been found and introduced as the notes evolve. It is likely that some still exist. If so, please feel free to open an issue in the GitHub repository and report it or make a pull request proposing a fix.

These notes have been prepared with the help of several students over the years. I am immensely grateful for their support and dedication to this material and teaching of this course.

F.

Contents

Chapter 1

Introduction

1.1 What is optimisation?

Optimisation is one of these words that has many meanings, depending on the context you take as a reference. In the context of this book, optimisation refers to *mathematical optimisation*, which is a discipline of applied mathematics.

In mathematical optimisation, we build upon concepts and techniques from calculus, analysis, linear algebra, and other domains of mathematics to develop methods to find values for variables (or solutions) within a given domain that maximise (or minimise) the value of a function. Specifically, we are trying to solve the following general problem:

$$\begin{aligned} &\text{min. } f(x) \\ &\text{s.t.: } x \in X. \end{aligned} \tag{1.1}$$

That is, we would like to find the solution x that *minimises* the value of the *objective function* f, such that (s.t.) x belongs to the *feasibility set* X. In a general sense, problems like this can be solved by employing the following strategy:

1. Analysing properties of the function $f(x)$ under specific domains and deriving the conditions that must be satisfied such that a point x is a candidate optimal point.
2. Applying numerical methods that iteratively search for points satisfying these conditions.

This idea is central to several knowledge domains and is often described with area-specific nomenclature. Fields such as economics, engineering, statistics, machine learning, and, perhaps more broadly, operations research are intensive users and developers of optimisation theory and applications.

1.1.1 Mathematical programming and optimisation

Operations research and mathematical optimisation are somewhat intertwined, as they both were born around a similar circumstance.

I personally like to separate *mathematical programming* from (mathematical) *optimisation*. Mathematical programming is a modelling paradigm in which we rely on (very powerful) analogies to model *real-world* problems. In that, we look at a given decision problem considering that:

- *variables* represent *decisions*, as in a business decision or a course of action. Examples include setting the parameters of (e.g., prediction) models, production systems layouts, geometries of structures, topologies of networks, and so forth;

- *domain* represents business rules or *constraints*, representing logic relations, design or engineering limitations, requirements, and such;
- *objective function* is a function that provides a measure of solution quality.

With these in mind, we can represent the decision problem as a *mathematical programming model* of the form of (1.1) that can be solved using *optimisation* methods. From now on, we will refer to this specific class of models as mathematical optimisation models or optimisation models for short. We will also use the term *solve the problem* to refer to the task of finding optimal solutions to optimisation models.

This book mostly focuses on the optimisation techniques employed to find optimal solutions for these models. As we will see, depending on the nature of the functions that are used to formulate the model, some methods might be more or less appropriate. Further complicating the issue, for models of a given nature, there might be alternative algorithms that can be employed, with no generalised consensus on whether one method is generally better performing than another, which is one of the aspects that make optimisation so exciting and multifaceted when it comes to alternative approaches. I hope that this makes more sense as we progress through the chapters.

1.1.2 Types of mathematical optimisation models

In general, the simpler the assumptions on the parts forming the optimisation model, the more efficient the methods to solve such problems.

Let us define some additional notation that we will use from now on. Consider a model in the general form

$$\begin{aligned} \text{min. } & f(x) \\ \text{s.t.: } & g_i(x) \leq 0, \ \forall i \in [m] \\ & h_i(x) = 0, \ \forall i \in [l] \\ & x \in X, \end{aligned}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is the objective function, $g : \mathbb{R}^n \to \mathbb{R}^m$ is a collection of m inequality constraints and $h : \mathbb{R}^n \to \mathbb{R}^l$ is a collection of l equality constraints, and the notation $[n]$ indicates the set of indices $\{1, \dots, n\}$.

In fact, every inequality constraint can be represented by an equality constraint by making $h_i(x) = g_i(x) + x_{n+1}$ and augmenting the decision variable vector $x \in \mathbb{R}^n$ to include the slack variable x_{n+1}. However, since these constraints behave very differently from an algorithmic standpoint, we will explicitly represent both whenever necessary.

The most general types of models are the following. We also use this as an opportunity to define some (admittedly confusing) nomenclature from the field of operations research that we will be using in these notes.

1. *Unconstrained models:* in these, the set $X = \mathbb{R}^n$ and $m = l = 0$. These are prominent in, e.g., machine learning and statistics applications, where f represents a measure of model fitness or prediction error.
2. *Linear programming (LP):* presumes linear objective function $f(x) = c^\top x$ and affine constraints g and h, i.e., of the form $a_i^\top x - b_i$, with $a_i \in \mathbb{R}^n$ and $b \in \mathbb{R}$. Normally, $X = \{x \in \mathbb{R}^n \mid x_j \geq 0, \ \forall j \in [n]\}$ enforce that the domain of the decision variables is the nonnegative orthant.

3. *Nonlinear programming (NLP):* some or all of the functions f, g, and h are nonlinear.

4. *Mixed-integer (linear) programming (MIP):* consists of an LP in which some (or all, being then simply integer programming) of the variables are constrained to be integers. In other words, $X \subseteq \mathbb{R}^k \times \mathbb{Z}^{n-k}$. Very frequently, the integer variables are constrained to be binary terms, i.e., $x_i \in \{0, 1\}$, $\forall i \in [n-k]$ and are meant to represent true-or-false or yes-or-no conditions.

5. *Mixed-integer nonlinear programming (MINLP):* are the intersection of MIPs and NLPs.

Remark: notice that we use the vector notation $c^\top x = \sum_{j \in J} c_j x_j$, with $J = \{1, \ldots, N\}$. This is just a convenience for keeping the notation compact.

1.2 Linear programming applications

We will consider now a few examples of linear programming models with a somewhat general structure. Many of these examples have features that can be combined into more general models.

1.2.1 Resource allocation

Most linear programming (LP) problems can be interpreted as a *resource allocation* problem. In that, we are interested in defining an optimal allocation of resources (i.e., a plan) that maximises return or minimises costs and satisfies allocation rules.

Specifically, let $I = \{1, \ldots, m\}$ be a set of resources that can be combined to produce products in the set $J = \{1, \ldots, n\}$. Assume that we are given a return c_j per unit of product j, $\forall j \in J$, and a list of a_{ij}, $\forall i \in I, \forall j \in J$, describing which and how much of the resources $i \in I$ are required for making product $j \in J$. Assume that the availability b_i of resource i, $\forall i \in I$, is known.

Our objective is to define the amounts x_j representing the production of $j \in J$. We would like to define those in a way that we optimise the resource allocation plan performance (in our case, maximise the return from the production quantities x_j) while making sure the resources needed for production do not exceed the availability of resources. For that, we need to define:

The *objective function*, which measures the *quality* of a production plan. In this case, the total return for a given plan is given by:

$$\text{max.} \sum_{j \in J} c_j x_j \Rightarrow c^\top x,$$

where $c = [c_1, \ldots, c_N]^\top$ and $x = [x_1, \ldots, x_N]^\top$ are n-sized vectors. Notice that $c^\top x$ denotes the inner (or dot) product. The transpose sign $^\top$ is meant to reinforce that we see our vectors as column vectors unless otherwise stated.

Next, we need to define *constraints* that state the conditions for a plan to be *valid*. In this context, a valid plan is a plan that does not utilise more than the amount of available resources b_i, $\forall i \in I$. This can be expressed as the collection (one for each $i \in I$) of affine (more often wrongly called, as we will too, linear) inequalities

$$\text{s.t.:} \sum_{j \in J} a_{ij} x_j \leq b_i, \forall i \in I \Rightarrow Ax \leq b,$$

where a_{ij} are the components of the $m \times n$ matrix A and $b = [b_1, \ldots, b_m]^\top$. Furthermore, we also must require that $x_i \geq 0$, $\forall i \in I$.

Combining the above, we obtain the generic formulation that will be used throughout this text to represent linear programming models:

$$\begin{aligned} \text{max. } & c^\top x \\ \text{s.t.: } & Ax \leq b \\ & x \geq 0. \end{aligned} \tag{1.2}$$

Illustrative example: the paint factory problem [5]

Let us work on a more specific example that will be useful for illustrating some important concepts related to the geometry of linear programming problems.

Let us consider a paint factory that produces *exterior* and *interior paint* from raw materials *M1* and *M2*. The *maximum demand* for interior paint is 2 tons/day. Moreover, the amount of interior paint produced *cannot exceed* that of exterior paint by more than 1 ton/day.

Our goal is to determine the optimal paint production plan. Table 1.1 summarises the data to be considered. Notice the constraints that must be imposed to represent the daily availability of paint.

	material (ton)/paint (ton)		
	exterior paint	interior paint	daily availability (ton)
material M1	6	4	24
material M2	1	2	6
profit ($1000 /ton)	5	4	

Table 1.1: Paint factory problem data

The paint factory problem is an example of a resource allocation problem. Perhaps one aspect that is somewhat dissimilar is the constraint representing the production rules regarding the relative amounts of exterior and interior paint. Notice, however, that this type of constraint also has the same format as the more straightforward resource allocation constraints.

Let x_1 be the amount produced of exterior paint (in tons) and x_2 the amount of interior paint. The complete model that optimises the daily production plan of the paint factory is:

$$\begin{aligned} \text{max. } & z = 5x_1 + 4x_2 \\ \text{s.t.: } & 6x_1 + 4x_2 \leq 24 \\ & x_1 + 2x_2 \leq 6 \\ & x_2 - x_1 \leq 1 \\ & x_2 \leq 2 \\ & x_1, x_2 \geq 0. \end{aligned}$$

Notice that paint factory model can also be *compactly represented* as in (1.2), where

$$c = [5, 4],\ x = [x_1, x_2],\ A = \begin{bmatrix} 6 & 4 \\ 1 & 2 \\ -1 & 1 \\ 0 & 1 \end{bmatrix},\ \text{and } b = [24, 6, 1, 2].$$

1.2.2 Transportation problem

Another important class of linear programming problems is those known as transportation problems. These problems are often modelled using the abstraction of graphs since they consider a network of nodes and arcs through which some flow must be optimised. Transportation problems have several important characteristics that can be exploited to design specialised algorithms, the so-called *network simplex* method. Although we will not discuss network simplex in this text, the simplex method (and its variant, the dual simplex method) will be at the centre of our developments later on.

The problem can be summarised as follows. We would like to plan the production and distribution of a certain product, taking into account that the transportation cost is known (e.g., proportional to the distance travelled), the factories (or source nodes) have a capacity limit, and the clients (or demand nodes) have known demands. Figure 1.1 illustrates a small network with two factories, located in San Diego and Seattle, and three demand points, located in New York, Chicago, and Miami. Table 1.2 presents the data related to the problem.

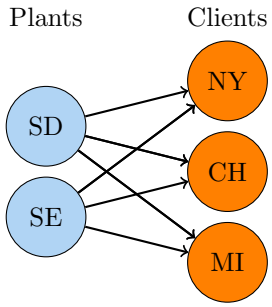


Figure 1.1: Schematic illustration of a network with two source nodes and three demand nodes

	Clients			
Factory	NY	Chicago	Miami	Capacity
Seattle	2.5	1.7	1.8	350
San Diego	3.5	1.8	1.4	600
Demands	325	300	275	-

Table 1.2: Problem data: unit transportation costs, demands and capacities

To formulate a linear programming model representing the transportation problem, let $i \in I = \{\text{Seattle}, \text{San Diego}\}$ be the index set representing factories. Similarly, let $j \in J = \{\text{New York}, \text{Chicago}, \text{Miami}\}$.

The decisions, in this case, are represented by x_{ij}, which represents the amount produced in factory i and sent to client j. Such a distribution plan can then be assessed by its total transportation cost, which is given by

$$\text{min. } z = 2.5x_{11} + 1.7x_{12} + 1.8x_{13} + 3.5x_{21} + 1.9x_{22} + 1.4x_{23}.$$

The total transportation cost can be more generally represented as

$$\text{min. } z = \sum_{i \in I} \sum_{j \in J} c_{ij} x_{ij}$$

where c_{ij} is the unit transportation cost from i to j. The problem has two types of constraints that must be observed, relating to the supply capacity and demand requirements. These can be stated as the following linear constraints

$$\begin{aligned}
&x_{11} + x_{12} + x_{13} \leq 350 \text{ (capacity limit in Seattle)}\\
&x_{21} + x_{22} + x_{23} \leq 600 \text{ (capacity limit in San Diego)}\\
&x_{11} + x_{21} \geq 325 \text{ (demand in New York)}\\
&x_{12} + x_{22} \geq 300 \text{ (demand in Chicago)}\\
&x_{13} + x_{23} \geq 275 \text{ (demand in Miami).}
\end{aligned}$$

These constraints can be expressed in the more compact form

$$\sum_{j \in J} x_{ij} \leq C_i, \ \forall i \in I \tag{1.3}$$

$$\sum_{i \in I} x_{ij} \geq D_j, \ \forall j \in J, \tag{1.4}$$

where C_i is the production capacity of factory $i \in I$ and D_j is the client demand $j \in J$. Notice that the terms on the left-hand side in (1.3) account for the total production in each of the source nodes $i \in I$. Analogously, in constraint (1.4), the term on the left-hand side accounts for the total of the demand satisfied at the demand nodes $j \in J$.

Using an optimality argument, we can see that any solution for which, for any $j \in J$, $\sum_{i \in I} x_{ij} > D_j$ can be improved by making $\sum_{i \in I} x_{ij} = D_j$. This shows that this constraint under these conditions will always be satisfied as an equality constraint instead and could be replaced as such.

The complete transportation model for the example above can be stated as

$$\begin{aligned}
\text{min. } z = {} & 2.5x_{11} + 1.7x_{12} + 1.8x_{13} + 3.5x_{21} + 1.9x_{22} + 1.4x_{23}\\
\text{s.t.: } & x_{11} + x_{12} + x_{13} \leq 350, \ x_{21} + x_{22} + x_{23} \leq 600\\
& x_{11} + x_{21} \geq 325, \ x_{12} + x_{22} \geq 300, \ x_{13} + x_{23} \geq 275\\
& x_{11}, \ldots, x_{23} \geq 0.
\end{aligned}$$

Or, more compactly, in the so-called *algebraic (or symbolic) form*

$$\begin{aligned}
\text{min. } z = {} & \sum_{i \in I} \sum_{j \in J} c_{ij} x_{ij}\\
\text{s.t.: } & \sum_{j \in J} x_{ij} \leq C_i, \ \forall i \in I\\
& \sum_{i \in I} x_{ij} \geq D_j, \ \forall j \in J\\
& x_{ij} \geq 0, \forall i \in I, \ \forall j \in J.
\end{aligned}$$

One interesting aspect to notice regarding algebraic forms is that they allow to represent the main structure of the model while being independent of the instance being considered. For example, regardless of whether the instance would have 5 or 50 nodes, the algebraic formulation is the same, allowing for detaching the problem instance (in our case the 5 node network) from the model itself. Moreover, most computational tools for mathematical programming modelling (hereinafter referred to simply as modelling - such as `JuMP.jl`) empower the user to define the optimisation model using this algebraic representation.

Algebraic forms are the main form in which we will specify optimisation models. This abstraction is a peculiar aspect of mathematical programming and is perhaps one of its main features, the fact that one must *formulate* models for each specific setting, which can be done in multiple ways and might have consequences for how well an algorithm performs computationally. This is a point we will discuss in more detail later on in this book.

1.2.3 Production planning (lot-sizing)

Production planning problems, commonly referred to as lot-sizing problems in contexts related to industrial engineering, consider settings where a planning horizon is taken into consideration. Differently from the previous examples, lot-sizing problems allow for the consideration of a time flow aspect, in which production that takes place in the past can be "shifted" to a future point in time by means of inventories (i.e., stocks). Inventories are important because they allow for taking advantage of different prices at different time periods, circumventing production capacity limitations, or preparing against uncertainties in the future (e.g., uncertain demands).

The planning horizon is represented by a collection of chronologically ordered elements $t \in \{1, \ldots, T\}$ representing a set of uniformly-sized time periods (e.g., months or days). Then, let us define the decision variables p_t as the amount produced in period t and k_t the amount stored in period t, which is available for use in periods $t' > t$. These decisions are governed by two costs: P_t, $\forall t \in T$, representing the production cost in each time period t and the unit holding cost H, that is, how much it costs to hold one unit of product for one time period.

Our objective is to satisfy the demands D_t, $\forall t \in T$, at the minimum possible cost. Figure 1.2 provides a schematic representation of the process to be modelled. Notice that each node represents a *material balance* to be considered, that is, at any period t, the total produced plus the amount held in inventory from the previous period $(t-1)$ must be the same as the amount used to satisfy the demand plus the amount held in inventory for the next period $(t+1)$.

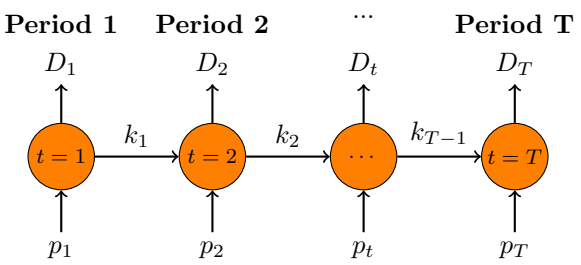


Figure 1.2: A schematic representation of the lot-sizing problem. Each node represents the material balance at each time period t

The production planning problem can be formulated as

$$\begin{aligned} \text{min.} \quad & \sum_{t \in T} (C_t p_t + H k_t) \\ \text{s.t.:} \quad & p_t + k_{t-1} = D_t + k_t, \ \forall t \in T \\ & p_t, k_t \geq 0, \ \forall t \in T. \end{aligned}$$

A few points must be considered carefully when dealing with lot-sizing problems. First, one must carefully consider boundary condition, that is, what the model is deciding in time periods $t = T$

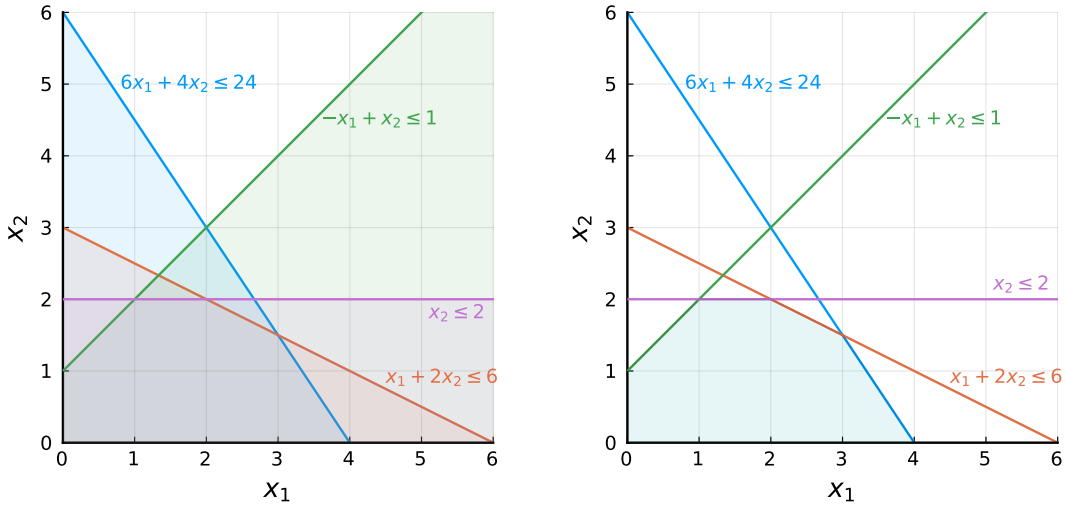
and what is the initial inventory (carried from $t = 0$). While the former will be seen by the model as the "end of the world", leading to the realization that optimal inventory levels at period $|T|$ must be zero, the latter might considerably influence how much production is needed during the planning horizon T. These must be observed and handled accordingly.

1.3 The geometry of LPs - graphical method

Let us now focus our attention to the geometry of linear programming (LP) models. As will become evident later on, LP models have a very peculiar geometry that is exploited by one of the most widespread methods to solve them, the *simplex method*.

1.3.1 The graphical method

In order to create a geometric intuition, we will utilise a graphical representation of the resource allocation example (the paint factory problem). But first, recall the general LP formulation (1.2), where A is an $m \times n$ matrix, and b, c, and x have suitable dimensions. Let a_i be one of the m rows of A. Notice each constraint $a_i^\top x \leq b_i$ defines a closed half-space, with its boundary defined by a hyperplane $a_i^\top x = b_i$, $\forall i \in I = [m]$ (we will return to these definitions in chapter 2; for now, just bear with me if these technical terms are unfamiliar to you). By plotting all of these closed half-spaces, we can see that their intersection will form the *feasible region* of the problem, that is, the (polyhedral) set of points that satisfy all constraints $Ax \leq b$. Figure 1.3 provides a graphical representation of the feasible region of the paint factory problem.



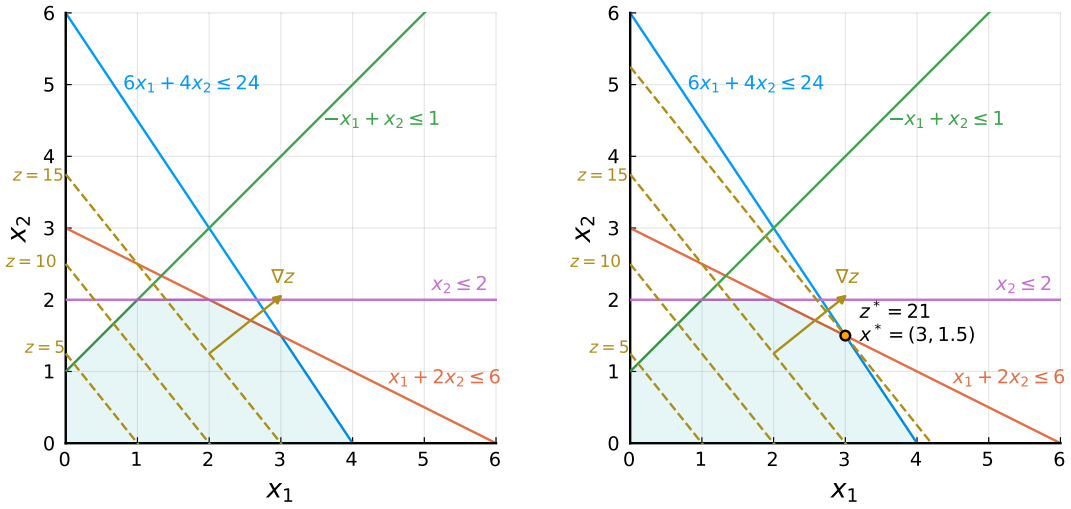
(a) (b)

Figure 1.3: The feasible region of the paint factory problem (in Figure 1.3b), represented as the intersection of the four closed-half spaces formed by each of the constraints (as shown in Figure 1.3a). Notice how the feasible region is a polyhedral set in $\mathbb{R}^2$, as there are two decision variables (x_1 and x_2)

We can use this visual representation to find the optimal solution for the problem, that is, the point

(x_1, x_2) within the feasible set such that the objective function value is maximal (recall that the paint factory problem is a maximisation problem). For that, we must consider how the objective function $z = c^\top x$ can be represented in the (x_1, x_2)-plane. Notice that the objective function forms a hyperplane in $(x_1, x_2, z) \subset \mathbb{R}^3$, of which we can plot level curves (i.e., projections) onto the (x_1, x_2)-plane. Figure 1.4a shows the plotting of three level curves, for $z = 5, 10$, and 15.

This observation provides us with a simple graphical method to find the optimal solution to linear problems. One must simply *sweep* the feasible region in the direction of the gradient $\nabla z = [\frac{\partial z}{\partial x_1}, \frac{\partial z}{\partial x_2}]^\top = [5, 4]^\top$ (or in its opposite direction, if minimising) until one last point (or edge) of contact remains, meaning that the whole of the feasible region is behind that furthermost level curve. Figure 1.4b illustrates the process of finding the optimal solution for the paint factory problem.



(a) (b)

Figure 1.4: Graphical representation of some of the level curves of the objective function $z = 5x_1 + 4x_2$. Notice that the constant gradient vector $\nabla z = (5, 4)^\top$ points to the direction in which the level curves increase in value. The optimal point is represented by $x^* = (3, 1.5)^\top$ with the furthermost level curve being that associated with the value $z^* = 21$

The graphical method is important because it allows us to notice several key features that will be used later on when we analyse a method that can search for optimal solutions for LP problems. The first is related to the notion of active or inactive constraints. We say that a constraint is *active* at a given point x if the constraint is satisfied as equality at the point x. For example, the constraints $6x_1 + 4x_2 \leq 24$ and $x_1 + 2x_2 \leq 6$ are active at the optimum $x^* = (3, 1.5)$, since $6(3) + 4(1.5) = 24$ and $3 + 2(1.5) = 6$. An active constraint indicates that the resource (or requirement) represented by that constraint is being fully depleted (or minimally satisfied).

Analogously, *inactive constraints* are constraints that are satisfied as strict inequalities at the optimum. For example, the constraint $-x_1 + x_2 \leq 1$ is inactive at the optimum, as $-(3) + 1.5 < 1$. In this case, an inactive constraint represents a resource (or requirement) that is not fully depleted (or is over-satisfied).

1.3.2 Geometrical properties of LPs

One striking feature concerning the geometry of LPs that becomes evident when we analyse the graphical method is that the number of candidate solutions is not infinite, but instead, only *a finite set* of points are potential candidates for optimal solution. This is because the process of sweeping in the direction of the gradient of the (linear) objective function will, in general, lead to a unique solution that must lie on a vertex of the polyhedral feasible set. The only exceptions are either when the gradient ∇z happens to be perpendicular to a facet of the polyhedral set (and in the direction of the sweeping) or in case the sweeping direction is not bounded by some of the facets of the polyhedral set. These exceptional cases will be discussed in more detail later on, but, as we will see, the observation still holds.

In the graphical example (i.e., in $\mathbb{R}^2$), notice how making $n = 2$ constraints active out of $m = 4$ constraints *forms a vertex*. However, not all vertices are feasible. This allows us to devise a mechanism to describe vertices by activating n of the m constraints at a time, in which we could exhaustively test and select the best (i.e., that with the largest objective function value). The issue, however, is that the number of candidates increases *exponentially* with the number of constraints and variables of the problem, which indicates this would quickly become computationally infeasible. As we will see, it turns out that this search idea can be made surprisingly efficient and is, in fact, the underlying framework of the *simplex method*. However, there are indeed artificially engineered worst-case settings where the method does need to consider every single vertex.

The simplex method exploits the above idea to *heuristically* search for solutions by selecting n constraints to be active from the m constraints available. Starting from an initial selection of constraints to be active, it selects one inactive constraint to activate and one to deactivate in a way that improvement in the objective function can be observed while feasibility is maintained. This process repeats until no improvement can be observed. In such a case, the *geometry* of the problem guarantees *(global) optimality*. In the following chapters, we will concentrate on defining algebraic objects that we will use to develop the simplex method.

1.4 Exercises

Exercise 1.1: Introduction to JuMP

Use the Julia package JuMP.jl to implement the problems below and find the optimal solution. For the two-dimensional problems, use Plots.jl to illustrate the feasible region and the optimal solution.

(a)

$$\begin{aligned} \text{max. } & x_1 + 2x_2 + 5x_3 \\ \text{s.t.: } & x_1 - x_2 - 3x_3 \geq 5 \\ & x_1 + 3x_2 - 7x_3 \leq 10 \\ & x_1 \leq 10 \\ & x_1, x_2 \geq 0. \end{aligned}$$

(b)

$$\begin{aligned} \text{max. } & 2x_1 + 4x_2 \\ \text{s.t.: } & x_1 + x_2 \leq 5 \\ & -x_1 + 3x_2 \leq 1 \\ & x_1 \leq 5 \\ & x_2 \leq 5 \\ & x_1, x_2 \geq 0. \end{aligned}$$

(c)

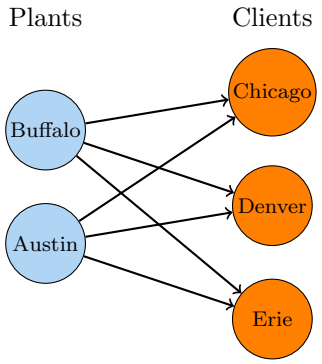
$$\begin{aligned} \text{min. } & -5x_1 + 10x_2 + x_3 + 2000x_4 \\ \text{s.t.: } & x_1 - x_2 \leq 1500 \\ & 4x_2 - x_3 \leq 5000x_4 \\ & x_1 + 3x_2 \geq 1000 \\ & x_1 \leq 10000 \\ & x_1, x_2 \in \mathbb{R}, x_3 \leq 0, x_4 \in \{0, 1\}. \end{aligned}$$

(d)

$$\begin{aligned} \text{max. } & 5x_1 + 3x_2 \\ \text{s.t.: } & x_1 + 5x_2 \leq 3 \\ & 3x_1 - x_2 \leq 5 \\ & x_1 \leq 2 \\ & x_2 \leq 30 \\ & x_1, x_2 \geq 0. \end{aligned}$$

Exercise 1.2: Transportation problem [4]

Consider the following network, where the supplies from the Austin and Buffalo nodes need to meet the demands in Chicago, Denver, and Erie. The data for the problem is presented in Table 1.3.



	Clients			
Factory	Chicago	Denver	Eire	
Buffalo	4	9	8	25
Austin	10	7	9	600
Demands	15	12	13	-

Table 1.3: Problem data: unit transportation costs, demands and capacities

Solve the transportation problem, finding the minimum cost transportation plan.

Exercise 1.3: Capacitated transportation

The Finnish company Next has a logistics problem in which used oils serve as raw material to form a class of renewable diesel. The supply chain team need to organise, to a minimal cost, the acquisition of two types of used oils (products p1 and p2) from three suppliers (supply nodes s1, s2, and s3) to feed the line of three of their used oils processing factories (demand nodes d1, d2, and d3). As the used oils are the byproduct of the supplier's core activities, the only requirement is that Next need to fetch the amount of oil and pay the transportation costs alone.

The oils have specific conditioning and handling requirements, so transportation costs vary between p1 and p2. Additionally, not all the routes (arcs) between suppliers and factories are available, as some distances are not economically feasible. Table 1.4a shows the volume requirement of the two types of oil from each supply and demand node, and Table **??** shows the transportation costs for each oil type per L and the arc capacity. Arcs with "-" for costs are not available as transportation routes.

node	p1	p2
s1 / d1	80 / 60	400 / 300
s2 / d2	200 / 100	1500 / 1000
s3 / d3	200 / 200	300 / 500

(a) Supply availability and demand per oil type [in L]

	d1 p1/p2 (cap)	d2 p1/p2 (cap)	d3 p1/p2 (cap)
s1	5/- (∞)	5/18 (300)	-/- (0)
s2	8/15 (300)	9/12 (700)	7/14 (600)
s3	-/- (0)	10/20 (∞)	8/- (∞)

(b) Arcs costs per oil type [in € per L] and arc capacities [in L]

Table 1.4: Supply chain data

Find the optimal oil acquisition plan for Next, i.e., solve its transportation problem to the minimum cost.

Exercise 1.4: The farmer's problem [2]

Consider a farmer who produces wheat, corn, and sugar beets on his 500 acres of land. During the winter, the farmer wants to decide how much land to devote to each crop.

The farmer knows that at least 200 tons (T) of wheat and 240T of corn are needed for cattle feed. These amounts can be raised on the farm or bought from a wholesaler. Any production in excess of the feeding requirement would be sold. Over the last decade, mean selling prices have been $ 170 and $ 150 per ton of wheat and corn, respectively. The purchase prices are 40 % more than this due to the wholesaler's margin and transportation costs. The planting costs per acre of wheat and corn are $ 150 and $ 230, respectively.

Another profitable crop is sugar beet, which he expects to sell at $36/T; however, the European Commission imposes a quota on sugar beet production. Any amount in excess of the quota can be sold only at $10/T. The farmer's quota for next year is 6000T. The planting cost per acre of sugar beet is $ 260.

Based on past experience, the farmer knows that the mean yield on his land is roughly 2.5T, 3T, and 20T per acre for wheat, corn, and sugar beets, respectively.

Based on the data, build up a model to help the farmer allocate the farming area to each crop and how much to sell/buy of wheat, corn, and sugar beets considering the following cases.

(a) The predictions are 100% accurate and the mean yields are the only realizations possible.

(b) There are three possible equiprobable scenarios (i.e, each one with a probability equal to $\frac{1}{3}$): a good, fair, and bad weather scenario. In the good weather, the yield is 20% better than the yield expected whereas in the bad weather scenario it is reduced 20% of the mean yield. In the regular weather scenario, the yield for each crop keeps the historical mean - 2.5T/acre, 3T/acre, and 20T/acre for wheat, corn, and sugar beets, respectively.

(c) What happens if we assume the same scenarios as item (b) but with probabilities 25%, 25%, and 50% for good, fair, and bad weather, respectively? How the production plan changes and why?

Exercise 1.5: Factory planning [6]

A factory makes seven products (PROD 1 to PROD 7) using the following machines: four grinders, two vertical drills, three horizontal drills, one borer and one planer. Each product yields a certain contribution to the profit (defined as $/unit selling price minus the cost of raw materials). These quantities (in $/unit) together with the unit production times (hours) required on each process are given in Table 1.5. A dash indicates that a product does not require a process. There are also marketing demand limitations on each product each month. These are given in Table 1.6.

	PROD1	**PROD2**	**PROD3**	**PROD4**	**PROD5**	**PROD6**	**PROD7**
Profit	10	6	8	4	11	9	3
Grinding	1.5	2.1	–	–	0.9	0.6	1.5
Vert. drilling	0.3	0.6	–	0.9	–	1.8	–
Horiz. drilling	0.6	–	2.4	–	–	–	1.8
Boring	0.15	0.09	–	0.21	0.3	–	0.24
Planing	–	–	0.03	–	0.15	–	0.15

Table 1.5: Product yields

	PROD1	**PROD2**	**PROD3**	**PROD4**	**PROD5**	**PROD6**	**PROD7**
January	500	1000	300	300	800	200	100
February	600	500	200	0	400	300	150
March	300	600	0	0	500	400	100
April	200	300	400	500	200	0	100
May	0	100	500	100	1000	300	0
June	500	500	100	300	1100	500	60

Table 1.6: Maximum demand

It is possible to store up to 100 of each product at a time at a cost of $0.5 per unit per month. There are no stocks at present, but it is desired to have a stock of 50 of each type of product at the end of June.

The factory works six days a week with two shifts of 8h each day. Assume that each month consists of only 24 working days. Also, there are no penalties for unmet demands. What is the factory's production plan (how much of which product to make and when) in order to maximise the total profit?

Exercise 1.6: Linear classifier

Suppose we have a set of pre-classified data points x_i in $\mathbb{R}^n$, divided into two sets based on their classification. For example, the data could be information on patients and the two sets would correspond to patients who have or do not have a certain disease. Note that unlike so far, x is now data, not a decision variable. Denote the respective index sets by I_1 and I_2, respectively. In order to predict the class of a new point x_{new}, we want to infer a classification rule from the data.

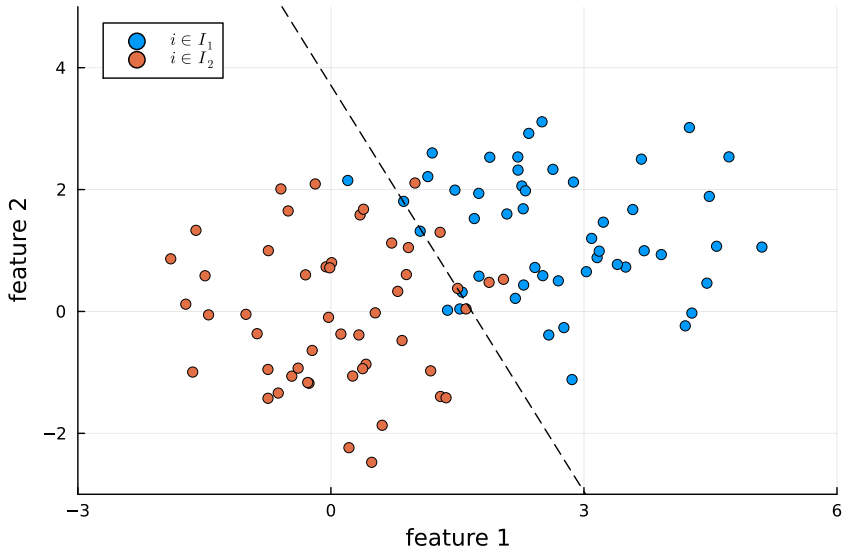


Figure 1.5: Two sets of data points defined by two features, separated by a line $ax = b$

Write a linear programming problem that finds the hyperplane $a^\top x = b$ such that if $a^\top x_{new} > b$, the point x_{new} is predicted to be in class 1, and if $a^\top x_{new} < b$, the predicted class is 2. The hyperplane should be optimal in the sense that it minimises the sum of absolute deviations $|a^\top x_i - b|$ for the misclassified points x_i in the training data, that is, points on the wrong side of the hyperplane. In Figure 1.5, any orange point x_i, $i \in I_2$ that is on the top/right of the line is on the wrong side and thus accumulates the error, and similarly for blue points on the bottom/left of the line.

Chapter 2

Basics of Linear Algebra

2.1 Basics of linear problems

As we have seen in the previous chapter, the feasible region of a linear programming problem can be represented as

$$Ax \leq b, \tag{2.1}$$

where A is a $m \times n$ matrix, x is a n-dimensional column vector (or more compactly, $x \in \mathbb{R}^n$), and b is an m-dimensional column vector ($b \in \mathbb{R}^m$). Notice that $\leq$ is considered component-wise. Also, let $\mathbf{dim}(x)$ denote the dimension of vector x.

Before introducing the simplex method, let us first revisit a few key elements and operations that we will use in the process. The first of them is presented in Definition 2.1.

Definition 2.1 (Matrix inversion)**.** *Let A be a square $n \times n$ matrix. A^{-1} is the inverse matrix of A if it exists and $AA^{-1} = I$, where I is the $(n \times n)$ identity matrix.*

Matrix inversion is the "kingpin" of linear (and nonlinear) optimisation. As we will see later on, performing efficient matrix inversion operations (in reality, operations that are equivalent to matrix inversion but that can exploit the matrix structure to be made more efficient from a computational standpoint) is of utmost importance for developing a linear optimisation solver.

Another important concept is the notion of *linear independence*. We formally state when a collection of vectors is said to be linearly independent (or dependent) in Definition 2.2.

Definition 2.2 (Linearly independent vectors)**.** *The vectors $\{x_i\}_{i=1}^k$ with $x_i \in \mathbb{R}^n$, $\forall i \in [k]$, are linearly dependent if there exist real numbers $\{a_i\}_{i=1}^k$ with $a_i \neq 0$ for at least one $i \in [k]$ such that*

$$\sum_{i=1}^{k} a_i x_i = 0;$$

otherwise, $\{x_i\}_{i=1}^k$ are linearly independent.

In essence, for a collection of vectors to be linearly independent, it must be so that none of the vectors in the collection can be expressed as a linear combination (that is, multiplying the vectors by nonzero scalars and adding them) of the others. Analogously, they are said to be linearly dependent if one vector in the collection can be expressed as a linear combination of the others.

This is simpler to see in $\mathbb{R}^2$. Two vectors are linearly independent if one cannot obtain one by multiplying the other by a constant, which effectively means that they are not parallel. If the two vectors are not parallel, then one of them must have a component in a direction that the other

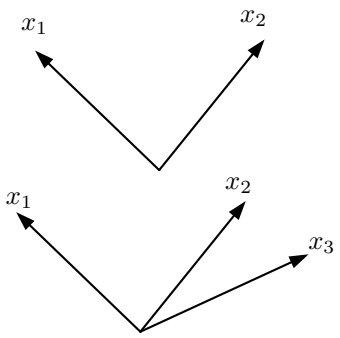


Figure 2.1: Linearly independent (top) and dependent (bottom) vectors in $\mathbb{R}^2$. Notice how, in the bottom picture, any of the vectors can be obtained by appropriately scaling and adding the other two

vector cannot "reach". The same idea can be generalised to any n-dimensional space. Also, that explains why one can only have up to n independent vectors in $\mathbb{R}^n$. Figure 2.1 illustrates this idea.

Theorem 2.3 summarises results that we will utilise in the upcoming developments. These are classical results from linear algebra and the proof is left as an exercise.

Theorem 2.3 (Inverses, linear independence, and solving $Ax = b$)**.** *Let A be a $m \times m$ matrix. Then, the following statements are equivalent:*

1. *A is invertible*
2. *$A^\top$ is invertible*
3. *The determinant of A is nonzero*
4. *The rows of A are linearly independent*
5. *The columns of A are linearly independent*
6. *For every $b \in \mathbb{R}^m$, the linear system $Ax = b$ has a unique solution*
7. *There exists some $b \in \mathbb{R}^m$ such that $Ax = b$ has a unique solution.*

Notice that Theorem 2.3 establishes important relationships between the geometry of the matrix A (its rows and columns) and consequences it has to our ability to calculate its inverse A^{-1} and, consequently, solve the system $Ax = b$, to which the solution is obtained as $x = A^{-1}b$. As we will see, solving linear systems of equations is the most important operation in the simplex method.

2.1.1 Subspaces and bases

Let us define some objects that we will frequently refer to. The first of them is the notion of a *subspace*. A subspace of $\mathbb{R}^n$ is a set comprising all linear combinations of its own elements. Specifically, if S is a subspace, then

$$S = \{ax + by : x, y \in S; a, b \in \mathbb{R}\}.$$

A related concept is the notion of a *span*. A span of a collection of vectors $\{x_i\}_{i=1}^k \in \mathbb{R}^n$ is the subspace of $\mathbb{R}^n$ formed by all linear combinations of such vectors, i.e.,

$$\mathbf{span}(x_1, \ldots, x_k) = \left\{ y = \sum_{i=1}^{k} a_i x_i : a_i \in \mathbb{R}, i \in [k] \right\}.$$

Notice how the two concepts are related: the span of a collection of vectors forms a subspace. Therefore, a subspace can be characterised by the collection of vectors whose span forms it. In other words, the span of a set of vectors is the subspace formed by all points we can represent by some linear combination of these vectors.

The missing part in this is the notion of a *basis*. A *basis* of the subspace $S \subseteq \mathbb{R}^n$ is a collection of vectors $\{x_i\}_{i=1}^k \in \mathbb{R}^n$ that are linearly independent such that $\mathbf{span}(x_1, \ldots, x_k) = S$.

Notice that a basis is a "minimal" set of vectors that form a subspace. You can think of it in light of the definition of linearly independent vectors (Definition 2.2); if a vector is linearly dependent to the others, it is not needed for characterising the subspace that the vectors span since it can be represented by a linear combination of the other vectors (and thus is in the subspace formed by the span of the other vectors).

The above leads us to some important realisations:

1. All bases of a given subspace S have the same dimension. Any extra vector would be linearly dependent to those vectors that span S. In that case, we say that the subspace has size (or dimension) k, the number of linearly independent vectors forming the basis of the subspace. We can overload the notation $\mathbf{dim}(S)$ to represent the dimension of the subspace S.

2. If the subspace $S \subset \mathbb{R}^n$ is formed by a basis of size $m < n$, we say that S is a proper subspace with $\mathbf{dim}(S) = m$, because it is not the whole $\mathbb{R}^n$ itself, but is contained within $\mathbb{R}^n$. For example, two linearly independent vectors form (i.e., span) a hyperplane in $\mathbb{R}^3$; this hyperplane is a proper subspace since $\mathbf{dim}(S) = m = 2 < 3 = n$.

3. If a proper subspace has dimension $m < n$, then it means that there are $n-m$ directions in $\mathbb{R}^n$ that are perpendicular to the subspace and to each other. That is, there are nonzero vectors a_i that are orthogonal to each other and to S. Or, equivalently, $a_i^\top x = 0$ for $i = n-m+1, ..., n$. Referring to the $\mathbb{R}^3$, if $m = 2$, then there is a third direction that is perpendicular to (or not in) S. Figure 2.2 can be used to illustrate this idea. Notice how one can find a vector, say x_3 that is perpendicular to S. This is because the whole space is $\mathbb{R}^3$, but S has dimension $m = 2$ (or $\mathbf{dim}(S) = 2$).

Theorem 2.4 builds upon the previous points to guarantee the existence of bases and propose a procedure to form them.

Theorem 2.4 (Forming bases from linearly independent vectors)**.** *Suppose that $S = \mathbf{span}(x_1, \ldots, x_k)$ has dimension $m \leq k$. Then*

1. *There exists a basis of S consisting of m of the vectors $x_1, \ldots, x_k$.*

2. *If $k' \leq m$ and $x_1, \ldots, x_{k'} \in S$ are linearly independent, we can form a basis for S by starting with $x_1, \ldots, x_{k'}$ and choosing $m - k'$ additional vectors from $x_1, \ldots, x_k$.*

Proof. Notice that, if every vector $x_{k'+1}, \ldots, x_k$ can be expressed as a linear combination of $x_1, \ldots x_{k'}$, then every vector in S is also a linear combination of $x_1, \ldots, x_{k'}$. Thus, $x_1, \ldots, x_{k'}$

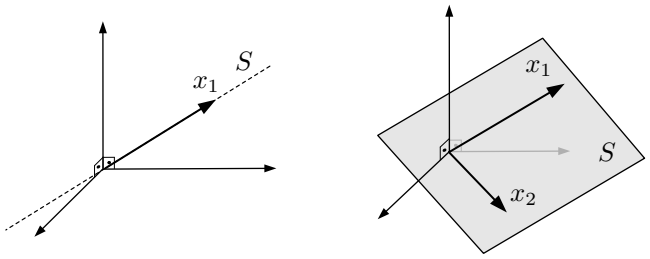


Figure 2.2: One- (left) and two-dimensional subspaces (right) in $\mathbb{R}^3$.

form a basis to S with $m = k'$. Otherwise, at least one of the vectors in $x_{k'+1}, \ldots, x_k$ is linearly independent from $x_1, \ldots, x_{k'}$. By picking one such vector, we now have $k' + 1$ of the vectors $x_{k'+1}, \ldots, x_k$ that are linearly independent. If we repeat this process $m - k'$ times, we end up with a basis for S. □

Our interest in subspaces and bases spans (pun intended!) from their usefulness in explaining how the simplex method works under a purely algebraic (as opposed to geometric) perspective. For now, we can use the opportunity to define some "famous" subspaces which will often appear in our derivations.

Let A be a $m \times n$ matrix. The *column space* of A consists of the subspace spanned by the n columns of A and has dimension m (recall that each column has as many components as the number of rows and is thus a m-dimensional vector). Likewise, the *row space* of A is the subspace in $\mathbb{R}^n$ spanned by the rows of A. Finally, the *null space* of A, often denoted as $\mathbf{null}(A) = \{x \in \mathbb{R}^n : Ax = 0\}$, consist of the vectors that are perpendicular to the row space of A.

One important notion related to those subspaces is their size. Both the row and the column space have the same size, which is the *rank* of A. If A is *full rank*, than it means that

$$\mathbf{rank}(A) = \min\{m, n\}.$$

Finally, the size of the null space of A is given $n - \mathbf{rank}(A)$, which is in line with Theorem 2.4.

2.1.2 Affine subspaces

A related concept is that of an *affine subspace*. Differently from linear subspaces (to which we have been referring to simply as subspaces), affine subspaces encode some form of translation, such as

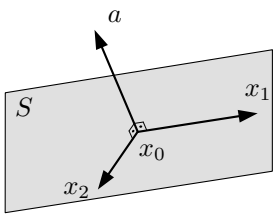
$$S = S_0 + x_0 = \{x + x_0 : x \in S_0\}.$$

Affine subspaces differ from linear subspaces because they do not contain the origin (recall that the definition of subspaces allows for a and b to be zero). Nevertheless, S has the *same dimension* as S_0.

Affine subspaces give a framework for representing linear programming problems algebraically. Specifically, let A be a $m \times n$ matrix with $m < n$ and b a m-dimensional vector. Then, let

$$S = \{x \in \mathbb{R}^n : Ax = b\}. \tag{2.2}$$

As we will see, the feasible set of any linear programming problem can be represented as an equality-constrained equivalent of the form of (2.2) by adding slack variables to the inequality



.

Figure 2.3: The affine subspace S generated by x_0 and $\mathbf{null}(a)$. Notice that all vectors in S, exemplified by x_1 and x_2 are perpendicular (i.e., have null dot product) to a

constraints, meaning that it will always lead to us having $m < n$. Now, assume that $x_0 \in \mathbb{R}^n$ is such that $Ax_0 = b$. Then, we have that

$$Ax = Ax_0 = b \Rightarrow A(x - x_0) = 0.$$

Thus, $x \in S$ if and only if the vector $(x - x_0)$ belongs to $\mathbf{null}(A)$, the nullspace of A. Notice that the feasible region S can be also defined as

$$S = \{x + x_0 : x \in \mathbf{null}(A)\},$$

being thus an affine subspace with dimension $n - m$, if A has m linearly independent rows (i.e., $\mathbf{rank}(A) = m$). This will have important implications in the way we can define multiple bases for S from the n vectors in the column space and what implications it has for the feasibility of the whole problem. Figure 2.3 illustrates this concept for a single-row matrix a. For a multiple-row matrix A, S becomes the intersection of multiple hyperplanes.

2.2 Convex polyhedral set

The feasible region of any linear programming problem is a convex polyhedral set, which we will simply refer to as a polyhedral set. That is because we are interested in polyhedral sets that are formed by an intersection of a finite number of half-spaces and can thus only be convex (as we will see in a moment), creating redundancy in our context but maybe some confusion overall.

2.2.1 Hyperplanes, half-spaces and polyhedral sets

Definition 2.5 formally states the structure that we refer to as polyhedral sets.

Definition 2.5 (Polyhedral set)**.** *A polyhedral set is a set that can be described as*

$$S = \{x \in \mathbb{R}^n : Ax \geq b\},$$

where A is an $m \times n$ matrix and b is a m-vector.

One important thing to notice is that polyhedral sets, as defined in Definition 2.5, as formed by the intersection multiple half-spaces. Specifically, let $\{a_i\}_{i=1}^m$ be the rows of A. Then, the set S can be described as

$$S = \left\{x \in \mathbb{R}^n : a_i^\top x \geq b_i,\ \forall i \in [m]\right\}, \tag{2.3}$$

which represents exactly the intersection of the half-spaces $a_i^\top x \geq b_i$. Furthermore, notice that the hyperplanes $a_i^\top x = b_i$, $\forall i \in [m]$, are the boundaries of each hyperplane, and thus describe one of the facets of the polyhedral set. Figure 2.4 illustrates a hyperplane forming two half-spaces (also polyhedral sets) and how the intersection of five half-spaces forms a (bounded) polyhedral set.

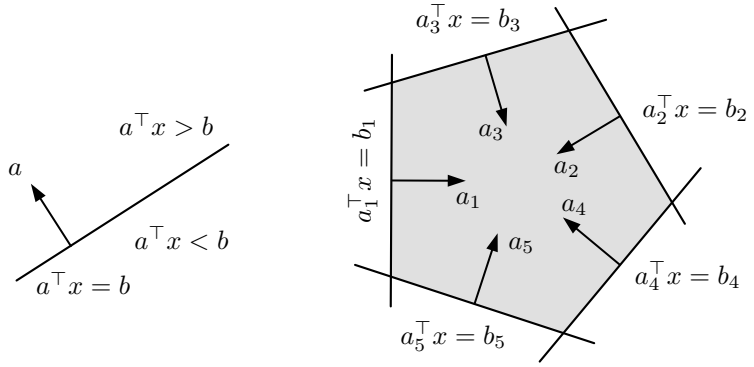


Figure 2.4: A hyperplane and its respective halfspaces (left) and the polyhedral set $\left\{x \in \mathbb{R}^2 : a_i^x \geq b_i, i = 1, \ldots, 5\right\}$ (right).

You might find authors referring to bounded polyhedral sets as polytopes. However, this is not used consistently across references, sometimes with switched meanings (for example, using polytope to refer to a set defined as in Definition 2.5 and using polyhedron to refer to a bounded version of S). In this text, we will only use the term polyhedral set to refer to sets defined as in Definition 2.5 and use the term bounded whenever applicable.

Also, it may be useful to formally define some elements in polyhedral sets. For that, let us consider a hyperplane $H = \left\{x \in \mathbb{R}^n : a^\top x = b\right\}$, with $a \in \mathbb{R}^n$ and $b \in \mathbb{R}$. Now assume that H is such that the set $F = H \cap S$ is not empty, and it only contains points that are in the boundary of S. This set is known as a *face* of a polyhedral set. If the face F has dimension zero, then F is called a vertex. Analogously, if $\mathbf{dim}(F) = 1$, then F is called an edge. Finally, if $\mathbf{dim}(F) = dim(S) - 1$, then F is called a facet. Notice that in $\mathbb{R}^3$, facets and faces are the same whenever the face is not an edge or a vertex.

2.2.2 Convexity of polyhedral sets

Convexity plays a crucial role in optimisation, being the "watershed" between easy and hard optimisation problems. One of the main reasons why we can solve challenging linear programming problems is due to the inherent convexity of polyhedral sets.

Let us first define the notion of convexity for sets, which is stated in Definition 2.6

Definition 2.6 (Convex set)**.** *A set $S \subseteq \mathbb{R}^n$ is convex if, for any $x_1, x_2 \in S$ and any $\lambda \in [0, 1]$, we have that $\overline{x} = \lambda x_1 + (1 - \lambda)x_2 \in S$.*

Definition 2.6 leads to a simple geometrical intuition: for a set to be convex, the line segment connecting any two points within the set must lie within the set. This is illustrated in Figure 2.5.

Associated with the notion of convex sets are two important elements we will refer to later when we discuss linear problems that embed *integrality requirements*. The first is the notion of a *convex combination*, which is already contained in Definition 2.6, but can be generalised for an arbitrary

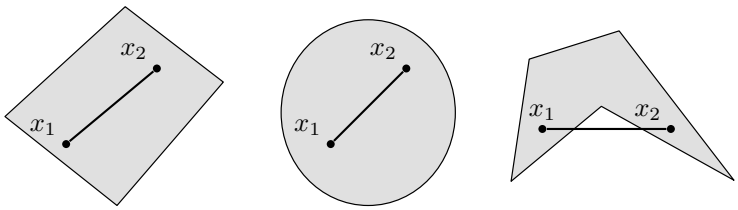


Figure 2.5: Two convex sets (left and middle) and one nonconvex set (right)

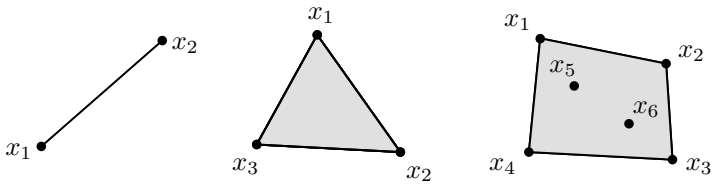


Figure 2.6: The convex hull of two points is the line segment connecting them (left); The convex hull of three (centre) and six (right) points in $\mathbb{R}^2$

number of points. The second consists of *convex hulls*, which are sets formed by combining the convex combinations of all elements within a given set. As one might suspect, convex hulls are always convex sets, regardless of whether the original set from which the points are drawn from is convex or not. These are formalised in Definition 2.7 and illustrated in Figure 2.6.

Definition 2.7 (Convex combinations and convex hulls)**.** *Let $x_1, \ldots, x_k \in \mathbb{R}^n$ and $\lambda_1, \ldots, \lambda_k \in \mathbb{R}$ such that $\lambda_i \geq 0$ for $i = 1, \ldots, k$ and $\sum_{i=1}^k \lambda_i = 1$. Then*

1. *$x = \sum_{i=1}^k \lambda_i x_i$ is a* convex combination *of $\{x_i\}_{i=1}^k \in \mathbb{R}^n$;*
2. *The* convex hull *of $\{x_i\}_{i=1}^k \in \mathbb{R}^n$, denoted* **conv**$(x_1, \ldots, x_k)$*, is the set of all convex combinations of $\{x_i\}_{i=1}^k \in \mathbb{R}^n$.*

We are now ready to state the result that guarantees the convexity of polyhedral sets of the form

$$S = \{x \in \mathbb{R}^n : Ax \leq b\}.$$

Theorem 2.8 (Convexity of polyhedral sets)**.** *The following statements are true:*

1. *The intersection of convex sets is convex*
2. *Every polyhedral set is a convex set*
3. *A convex combination of a finite number of elements of a convex set also belongs to that set*
4. *The convex hull of a finite number of elements is a convex set.*

Proof. We provide the proof for each of the statements individually.

1. Let S_i, for $i \in I = \{1, \ldots, n\}$, be a collection of n convex sets and suppose that $x, y \in \bigcap_{i \in I} S_i$. Let $\lambda \in [0, 1]$. Since S_i are convex and $x, y \in S_i$ for all $i \in I$, $\lambda x + (1 - \lambda)y \in S_i$ for all $i \in I$ and, thus, $\lambda x + (1 - \lambda)y \in \bigcap_{i \in I} S_i$.

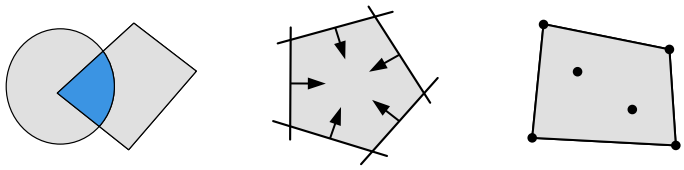


Figure 2.7: Illustration of statement 1 (left), 2 (centre), and 3 and 4 (right)

2. Let $a \in \mathbb{R}^n$ and $b \in \mathbb{R}$. Let $x, y \in \mathbb{R}^n$, such that $a^\top x \geq b$ and $a^\top y \geq b$. Let $\lambda \in [0,1]$. Then $a^\top(\lambda x + (1-\lambda)y) \geq \lambda b + (1-\lambda)b = b$, showing that half-spaces are convex. The result follows from combining this with (1).

3. By induction. Let S be a convex set and assume that the convex combination of $x_1, \ldots, x_k \in S$ also belongs to S. Consider $k+1$ elements $x_1, \ldots, x_{k+1} \in S$ and $\lambda_1, \ldots, \lambda_{k+1}$ with $\lambda_i \in [0,1]$ for $i \in [k+1]$ and $\sum_{i=1}^{k+1} \lambda_i = 1$ and $\lambda_{k+1} \neq 1$ (without loss of generality). Then

$$\sum_{i=1}^{k+1} \lambda_i x_i = \lambda_{k+1} x_{k+1} + (1-\lambda_{k+1}) \sum_{i=1}^{k} \frac{\lambda_i}{1-\lambda_{k+1}} x_i. \tag{2.4}$$

   Notice that $\sum_{i=1}^{k} \frac{\lambda_i}{1-\lambda_{k+1}} = 1$. Thus, using the induction hypothesis, $\sum_{i=1}^{k} \frac{\lambda_i}{1-\lambda_{k+1}} x_i \in S$. Considering that S is convex and using (2.4), we conclude that $\sum_{i=1}^{k+1} \lambda_{k+1} x_{k+1} \in S$, completing the induction.

4. Let $S = \mathbf{conv}(x_1, \ldots, x_k)$. Let $y = \sum_{i=1}^{k} \alpha_i x_i$ and $z = \sum_{i=1}^{k} \beta_i x_i$ be such that $y, z \in S$, $\alpha_i, \beta_i \geq 0$, and $\sum_{i=1}^{k} \alpha_i = \sum_{i=1}^{k} \beta_i = 1$. Let $\lambda \in [0,1]$. Then

$$\lambda y + (1-\lambda)z = \lambda \sum_{i=1}^{k} \alpha_i x_i + (1-\lambda) \sum_{i=1}^{k} \beta_i x_i = \sum_{i=1}^{k} (\lambda\alpha_i + (1-\lambda)\beta_i) x_i. \tag{2.5}$$

   Since $\sum_{i=1}^{k} \lambda\alpha_i + (1-\lambda)\beta_i = 1$ and $\lambda\alpha_i + (1-\lambda)\beta_i \geq 0$ for $i = 1, \ldots, k$, $\lambda y + (1-\lambda)z$ is a convex combination of $x_1, \ldots, x_k$ and, thus, $\lambda y + (1-\lambda)z \in S$, showing the convexity of S. □

Figure 2.7 illustrates some of the statements represented in the proof. For example, the intersection of the convex sets is always a convex set. One should notice however that the same does not apply to the union of convex sets. Notice that statement 2 proves that polyhedral sets as defined according to Definition 2.5 are convex. Finally the third figure on the right illustrates the convex hull of four points as a convex polyhedral set containing the lines connecting any two points within the set.

We will halt our discussion about convexity for now, as we have covered the key facts we will need to prove that the simplex method converges to an optimal point. The last result missing to achieve this objective is to show a simple yet very powerful result, which states that the presence of convexity is what allows us to conclude that a locally optimal solution returned by an optimisation algorithm applied to a linear programming problem is indeed optimal for the problem at hand. It so turns out that, in the context of linear programming, convexity is a given since linear functions are convex by definition and the feasibility set of linear programming is also convex (as we have just shown in 2.8).

Theorem 2.9 (Global optimality for convex problems)**.** *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a convex function, that is, $f(\lambda x_1 + (1-\lambda)x_2) \leq \lambda f(x_1) + (1-\lambda)f(x_2),\ \lambda \in [0,1]$, and let $S \subset \mathbb{R}^n$ be a convex set. Let x^* be an element of S. Suppose that x^* is a local optimum for the problem of minimising $f(x)$ over S. That is, there exists some $\epsilon > 0$ such that $f(x^*) \leq f(x)$ for all $x \in S$ for which $\|x - x^*\| \leq \epsilon$. Then, x^* is globally optimal, meaning that $f(x^*) \leq f(x)$ for all $x \in S$.*

Proof. Suppose, in order to derive a contradiction that $f(x) < f(x^*)$ for some $x \in S$. Using Definition 2.6, we have that

$$f(\lambda x + (1-\lambda)x^*) \leq \lambda f(x) + (1-\lambda)f(x^*) < \lambda f(x^*) + (1-\lambda)f(x^*) = f(x^*),\ \forall \lambda \in [0,1].$$

We see that the strict inequality $f(\lambda x + (1-\lambda)x^*) < f(x^*)$ holds for any point $\lambda x + (1-\lambda)x^*$, including those with $\|\lambda x + (1-\lambda)x^* - x^*\| \leq \epsilon$. Our assumption thus contradicts the local optimality of x^*, and this proves that $f(x) \geq f(x^*)$ for all $x \in S$. □

2.3 Extreme points, vertices, and basic feasible solutions

We focus on the algebraic representation of the most relevant geometric elements in the optimisation of linear programming problems. As we have seen in the graphical example in the previous chapter, the optimum of linear programming problems is generally located at the vertices of the feasible set. Furthermore, such vertices are formed by the intersection of n constraints (in a n-dimensional space), which comprises constraints that are active (or satisfied at the boundary of the half-space of said constraints).

First, let us formally define the notions of vertex and extreme point. Although in general these can refer to different objects, we will see that in the case of linear programming problems, if a point is a vertex, then it is an extreme point as well, the converse also being true.

Definition 2.10 (Vertex)**.** *Let P be a convex polyhedral set. The vector $x \in P$ is a vertex of P if there exists some c such that $c^\top x < c^\top y$ for all $y \in P$ with $y \neq x$.*

Definition 2.11 (Extreme points)**.** *Let P be a convex polyhedral set. The vector $x \in P$ is an extreme point of P if there are no two vectors $y, z \in P$ (different than x) such that $x = \lambda y + (1-\lambda)z$, for any $\lambda \in [0,1]$.*

Figure 2.8 provides an illustration of the Definitions 2.10 and 2.11. Notice that the definition of a vertex involves an additional hyperplane that, once placed on a vertex point, strictly contains the whole polyhedral set in one of the half-spaces it defines, except for the vertex itself. On the other hand, the definition of an extreme point only relies on convex combinations of elements in the set itself.

Definition 2.10 also hints at an important consequence for linear programming problems. As we can see from Theorem 2.8, P is convex, which guarantees that P is contained in the half-space $c^\top y > c^\top x$. This implies that $c^\top x \leq c^\top y$, $\forall y \in P$, which is precisely the condition that x must satisfy to be the minimum for the problem $\min._x \{c^\top x : x \in P\}$.

Now we focus on the description of active constraints from an algebraic standpoint. For that, let us first generalise our setting by considering all possible types of linear constraints. That is, let us

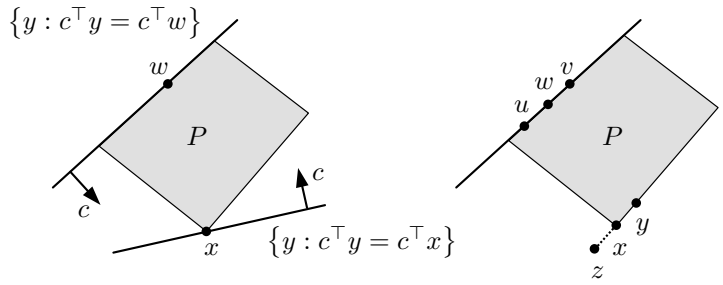


Figure 2.8: Representation of a vertex (left) and a extreme point (right)

consider the convex polyhedral set $P \subset \mathbb{R}^n$, formed by the set of inequalities and equalities:

$$a_i^\top x \geq b, i \in M_1,$$
$$a_i^\top x \leq b, i \in M_2,$$
$$a_i^\top x = b, i \in M_3.$$

Definition 2.12 (Active (or binding) constraints)**.** *If a vector $\overline{x}$ satisfies $a_i^\top \overline{x} = b_i$ for some $i \in M_1, M_2$, or M_3, we say that the corresponding constraints are active (or binding).*

Definition 2.12 formalises the notion of active constraints. This is illustrated in Figure 2.9, where the polyhedral set $P = \left\{x \in \mathbb{R}^3 : x_1 + x_2 + x_3 = 1,\ x_i \geq 0,\ i = 1, 2, 3\right\}$ is represented. Notice that, while points A, B, C and D have 3 active constraints, E only has 2 active constraints ($x_2 = 0$ and $x_1 + x_2 + x_3 = 1$).

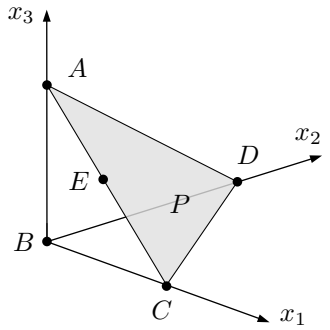


Figure 2.9: Representation of P in $\mathbb{R}^3$.

Theorem 2.13 sows a thread between having a collection of active constraints forming a vertex and being able to describe it as a basis of a subspace that is formed by the vectors a_i that form these constraints. This link is what will allow us to characterise vertices by their forming active constraints.

Theorem 2.13 (Properties of active constraints)**.** *Let $\overline{x} \in \mathbb{R}^n$ and $I = \left\{i \in M_1 \cup M_2 \cup M_3 : a_i^\top \overline{x} = b_i\right\}$. Then, the following are equivalent:*

1. *There exists n vectors in $\{a_i\}_{i \in I}$ that are linearly independent.*

2. *The* $\mathbf{span}(\{a_i\}_{i\in I})$ *spans* $\mathbb{R}^n$*. That is, every* $x \in \mathbb{R}^n$ *can be expressed as a linear combination of* $\{a_i\}_{i\in I}$*.*

3. *The system of equations* $\left\{a_i^\top x = b_i\right\}_{i\in I}$ *has a unique solution.*

Proof. Suppose that $\{a_i\}_{i\in I}$ spans $\mathbb{R}^n$, implying that the $\mathbf{span}(\{a_i\}_{i\in I})$ has dimension n. By Theorem 2.4 (part 1), n of these vectors form a basis for $\mathbb{R}^n$ and are, thus, linearly independent. Moreover, they must span $\mathbb{R}^n$ and therefore every $x \in \mathbb{R}^n$ can be expressed as a combination of $\{a_i\}_{i\in I}$. This connects (1) and (2).

Assume that the system of equations $\left\{a_i^\top x = b_i\right\}_{i\in I}$ has multiple solutions, say x_1 and x_2. Then, the nonzero vector $d = x_1 - x_2$ satisfies $a_i^\top d = 0$ for all $i \in I$. As d is orthogonal to every a_i, $i \in I$, d cannot be expressed as a combination of $\{a_i\}_{i\in I}$ and, thus, $\{a_i\}_{i\in I}$ do not span $\mathbb{R}^n$.

Conversely, if $\{a_i\}_{i\in I}$ do not span $\mathbb{R}^n$, choose $d \in \mathbb{R}^n$ that is orthogonal to $\mathbf{span}(\{a_i\}_{i\in I})$. If x satisfies $\left\{a_i^\top x = b_i\right\}_{i\in I}$, so does $\left\{a_i^\top (x + d) = b_i\right\}_{i\in I}$, thus yielding multiple solutions. This connects (2) and (3). □

Notice that Theorem 2.13 implies that there are (at least) n *active constraints* (a_i) that are *linearly independent* at $\overline{x}$. This is the reason why we will refer to $\overline{x}$, and any vertex-forming solution, as a *basic solution*, of which we will be interested in those that are feasible, i.e., that satisfy all constraints $i \in M_1 \cup M_2 \cup M_3$. Definition 2.14 provides a formal definition of these concepts.

Definition 2.14 (Basic feasible solution (BFS))**.** *Consider a convex polyhedral set* $P \subset \mathbb{R}^n$ *defined by linear equality and inequality constraints, and let* $\overline{x} \in \mathbb{R}^n$*.*

1. $\overline{x}$ *is a* basic solution *if*

   (a) *All equality constraints are active,*

   (b) *Out of the constraints active at* $\overline{x}$*,* n *of them are linearly independent, and*

   (c) $\overline{x}$ *is the unique solution of the linear system formed by* n *linearly-independent active constraints.*

2. *if* $\overline{x}$ *is a basic solution satisfying all constraints, we say* $\overline{x}$ *is a basic feasible solution.*

Figure 2.10 provides an illustration of the notion of basic solutions, and show how only a subset of the basic solutions are feasible. As one might infer, these will be the points of interest in out future developments, as these are the candidates for optimal solution.

We finalise stating the main result of this chapter, which formally confirms the intuition we have developed so far. That is, for convex polyhedral sets, the notion of vertices and extreme points coincide, and these points can be represented as basic feasible solutions. This is precisely the link that allows for considering the feasible region of linear programming problems under a purely algebraic characterisation of the candidates for optimal solutions, those described uniquely by a subset of constraints of the problem that is assumed to be active.

Theorem 2.15 (BFS, extreme points and vertices)**.** *Let* $P \subset \mathbb{R}^n$ *be a convex polyhedral set and let* $\overline{x} \in P$*. Then, the following are equivalent*

$$\overline{x} \text{ is a vertex} \iff \overline{x} \text{ is an extreme point} \iff \overline{x} \text{ is a BFS.}$$

Proof. Let $P = \{x \in \mathbb{R}^n : a_i^\top x \geq b_i, i \in M_1, a_i^\top x = b_i, i \in M_2\}$, and $I = \left\{i \in M_1 \cup M_2 \mid a_i^\top x = b_i\right\}$.

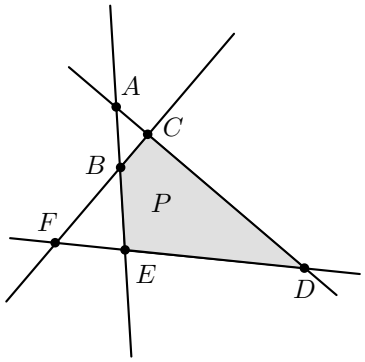


Figure 2.10: Points A to F are basic solutions; B,C,D, and E are BFS.

1. (Vertex $\Rightarrow$ Extreme point) Suppose $\overline{x}$ is a vertex. Then, there exists some $c \in \mathbb{R}^n$ such that $c^\top \overline{x} < c^\top x$, for every $x \in P$ with $x \neq \overline{x}$ (cf. Definition 2.10). Take $y, z \in P$ with $y, z \neq \overline{x}$. Thus $c^\top \overline{x} < c^\top y$ and $c^\top \overline{x} < c^\top z$. For $\lambda \in [0,1]$, $c^\top \overline{x} < c^\top(\lambda y + (1-\lambda)z)$ implying that $\overline{x} \neq \lambda y + (1-\lambda)z$, and is thus an extreme point (cf. Definition 2.11).

2. (Extreme point $\Rightarrow$ BFS) We will prove the contrapositive instead[1]. Suppose $\overline{x} \in P$ is not a BFS. Then, there are no n linearly independent vectors within $\{a_i\}_{i \in I}$. Thus the vectors $\{a_i\}_{i \in I}$ lie in a proper subspace of $\mathbb{R}^n$. Let the nonzero vector $d \in \mathbb{R}^n$ be such that $a_i^\top d = 0$, for all $i \in I$.

   Let $\epsilon > 0$, $y = \overline{x} + \epsilon d$, and $z = \overline{x} - \epsilon d$. Notice that $a_i^\top y = a_i^\top z = b_i$, for all $i \in I$. Moreover, for $i \neq I$, $a_i^\top x > b_i$ and, provided that ϵ is sufficiently small (such that $\epsilon |a_i^\top d| < a_i^\top \overline{x} - b_i$), we have that $a_i^\top x \geq b_i$ for all $i \in I$. Thus $y \in P$, and by a similar argument, $z \in P$. Now, by noticing that $\overline{x} = \frac{1}{2}y + \frac{1}{2}z$, we see that $\overline{x}$ is not an extreme point.

3. (BFS $\Rightarrow$ Vertex) Let $\overline{x}$ be a BFS. Define $c = \sum_{i \in I} a_i$. Then

$$c^\top \overline{x} = \sum_{i \in I} a_i^\top \overline{x} = \sum_{i \in I} b_i.$$

   Also, for any $x \in P$, we have that

$$c^\top x = \sum_{i \in I} a_i^\top x \geq \sum_{i \in I} b_i,$$

   since $a_i^\top x \geq b_i$ for $i \in M_1 \cup M_2$. Thus, for any $x \in P$, $c^\top \overline{x} \leq c^\top x$, making $\overline{x}$ a vertex (cf. Definition 2.10). □

Some interesting insights emerge from the proof of Theorem 2.15, upon which we will build our next developments. Once the relationship between being a vertex/extreme point and a BFS is made, it means that $\overline{x}$ can be recovered as the unique solution of a system of linear equations, these equations being the active constraints at that vertex. This means that the list of all optimal solution candidate points can be obtained by simply looking at all possible combinations of n active constraints, discarding those that are infeasible. This means that the number of candidates for optimal solution is *finite* and can be bounded by $\binom{m}{n}$, where $m = |M_1 \cup M_2|$.

[1]Consider two propositions A and B. Then, we have that $A \Rightarrow B \equiv \neg B \Rightarrow \neg A$. The latter is known as the *contrapositive* of the former.

2.4 Exercises

Exercise 2.1: Polyhedral sets [1]

Which of the following sets are polyhedral sets?

a) $\{(x, y) \in \mathbb{R}^2 \mid x \cos\theta + y \sin\theta \leq 1, \theta \in [0, \pi/2], x \geq 0, y \geq 0\}$

b) $\{x \in \mathbb{R} \mid x^2 - 8x + 15 \leq 0\}$

c) The empty set ($\emptyset$).

Exercise 2.2: Convexity of polyhedral sets

Prove the following theorem.

Theorem (Convexity of polyhedral sets)**.** The following convexity properties about convex sets can be said:

1. *The intersection of convex sets is convex*
2. *Every polyhedral set is a convex set*
3. *A convex combination of a finite number of elements of a convex set also belongs to that set*
4. *The convex hull of a finite number of elements is a convex set.*

Note: the proof of the theorem is proved in the notes. Use this as an opportunity to revisit the proof carefully, and try to take as many steps without consulting the text as you can. This is a great exercise to help you internalise the proof and its importance in the context of this book. I strongly advise against blindly memorising it, as I suspect you will never (in my courses, at least) be requested to recite the proof literally.

Exercise 2.3: Inverses, linear independence, and solving $Ax = b$

Prove the following theorem.

Theorem (Inverses, linear independence, and solving $Ax = b$)**.** *Let A be a $m \times m$ matrix. Then, the following statements are equivalent:*

1. *A is invertible*
2. *$A^\top$ is invertible*
3. *The determinant of A is nonzero*
4. *The rows of A are linearly independent*
5. *The columns of A are linearly independent*
6. *For every $b \in \mathbb{R}^m$, the linear system $Ax = b$ has a unique solution*
7. *There exists some $b \in \mathbb{R}^m$ such that $Ax = b$ has a unique solution.*

Exercise 2.4: Linear independence

a) According to Theorem 2.3, if the columns (or rows) of the $m \times m$ matrix A are linearly independent, the system $Ax = b$ has a unique solution for every vector b. Explain the significance of this result in the context of linear optimization.

b) If the columns of A are not linearly independent, the system $Ax = b$ has either no solution or infinitely many solutions. Using the notions of *span* and *proper subspace*, explain why this is the case.

c) Solve the following systems of equations. Which of these systems have an invertible coefficient matrix?

$$\begin{aligned} 2x_1 + 3x_2 + x_3 &= 12 \\ x_1 + 2x_2 + 3x_3 &= 12 \\ 3x_1 + x_2 + 2x_3 &= 12 \end{aligned}$$

$$\begin{aligned} x_1 + 2x_2 + 4x_3 &= 6 \\ 2x_1 + x_2 + 5x_3 &= 6 \\ x_1 + 3x_2 + 5x_3 &= 6 \end{aligned}$$

$$\begin{aligned} x_1 + 2x_2 + 4x_3 &= 5 \\ 2x_1 + x_2 + 5x_3 &= 4 \\ x_1 + 3x_2 + 5x_3 &= 7 \end{aligned}$$

Exercise 2.5: Properties of active constraints

Let us consider the convex polyhedral set $P \subset \mathbb{R}^n$, formed by the set of equalities and inequalities:

$$\begin{aligned} a_i^\top x &\geq b, i \in M_1, \\ a_i^\top x &\leq b, i \in M_2, \\ a_i^\top x &= b, i \in M_3. \end{aligned}$$

Prove the following result.

Theorem (Properties of active constraints)**.** *Let $\overline{x} \in \mathbb{R}^n$ and $I = \left\{ i \in M_1 \cup M_2 \cup M_3 \mid a_i^\top \overline{x} = b_i \right\}$. Then, the following are equivalent:*

1. *There exists n vectors in $\{a_i\}_{i \in I}$ that are linearly independent.*
2. *The* **span**$(\{a_i\}_{i \in I})$ *spans $\mathbb{R}^n$. That is, every $x \in \mathbb{R}^n$ can be expressed as a combination of $\{a_i\}_{i \in I}$.*
3. *The system of equations $\left\{ a_i^\top x = b_i \right\}_{i \in I}$ has a unique solution.*

Note: see Exercise 2.2.

Exercise 2.6: Vertex, extreme points, and BFSs

Prove the following result.

Theorem (BFS, extreme points and vertices)**.** *Let $P \subset \mathbb{R}^n$ be a convex polyhedral set and let $\overline{x} \in P$. Then, the following are equivalent*

1. *$\overline{x}$ is a vertex;*
2. *$\overline{x}$ is a extreme point;*
3. *$\overline{x}$ is a BFS;*

Note: see Exercise 2.2.

Exercise 2.7: Binding constraints

Given the linear program defined by the system of inequalities below,

$$
\begin{aligned}
\text{max. } & 2x_1 + x_2 \\
\text{s.t.: } & 2x_1 + 2x_2 \leq 9 \\
& 2x_1 - x_2 \leq 3 \\
& x_1 - x_2 \leq 1 \\
& x_1 \leq 2.5 \\
& x_2 \leq 4 \\
& x_1, x_2 \geq 0.
\end{aligned}
$$

Assess the following points relative to the polyhedron defined in $\mathbb{R}^2$ by this system and classify them as in (i) belonging to which active constraint(s), and (ii) being a (basic) non-feasible/feasible solution. Use Definitions 2.12 and 2.14 to check if your classification is correct.

a) $(1.5, 0)$

b) $(1, 0)$

c) $(2, 1)$

d) $(1.5, 3)$

Chapter 3

Basis, Extreme Points and Optimality in Linear Programming

3.1 Polyhedral sets in standard form

In the context of linear programming problems, we will often consider problems written in the so-called *standard form*. The standard form can be understood as posing the linear programming problem as an underdetermined system of equations (that is, with fewer equations than variables). Then, we will work on selecting a subset of the variables to be set to zero so that the number of remaining variables is the same as that of equations, making the system solvable.

A key point in this chapter will be devising how we relate this process of selecting variables with that of selecting a subset of active constraints (forming a vertex, as we have seen in the previous chapter) that will eventually lead to an optimal solution.

3.1.1 The standard form of linear programming problems

First, let us formally define the notion of a standard-form polyhedral set. Let A be a $m \times n$ matrix and $b \in \mathbb{R}^m$. The *standard form* polyhedral set P is given by

$$P = \{x \in \mathbb{R}^n : Ax = b,\ x \geq 0\}.$$

We assume that the m equality constraints are linearly independent, i.e., A is full (row) rank ($m \leq n$). We know that a basic solution can be obtained from a collection of n active constraints since the problem is defined in $\mathbb{R}^n$.

One important point is that *any* linear programming problem can be represented in the standard form. This is achieved utilising nonnegative *slack variables*. Thus, a feasibility set that is, say, originally represented as

$$P = \{x \in \mathbb{R}^n : A_1 x \leq b_1, A_2 x \geq b_2, x \geq 0\}$$

can be equivalently represented as a standard-form polyhedral set. For that, it must be modified to consider slack variables $s_1 \geq 0$ and $s_2 \geq 0$ such that

$$P = \left\{(x, s_1, s_2) \in \mathbb{R}^{(n+|b_1|+|b_2|)} : A_1 x + s_1 = b_1, A_2 x - s_2 = b_2, (x, s_1, s_2) \geq 0\right\},$$

where $|u|$ represents the cardinality of the vector u. Another transformation that may be required consists of imposing the condition $x \geq 0$. Let us assume that a polyhedral set P was such that (notice the absent nonnegativity condition)

$$P = \{x \in \mathbb{R}^n : Ax = b\}.$$

It is a requirement for standard-form linear programs to have all variables to be assumed nonnegative. To achieve that in this case, we can simply include two auxiliary variables, say x^+ and x^-, with the same dimension as x, and reformulate P as

$$P = \left\{x^+, x^- \in \mathbb{R}^n : A(x^+ - x^-) = b, x^+, x^- \geq 0\right\}.$$

These transformations, as we will see, will be required for employing the simplex method to solve linear programming problems with inequality constraints and, inevitably, will always render standard form linear programming problems with more variables than constraints, or $m < n$.

The standard-form polyhedral set P always has, by definition, m active constraints because of its equality constraints. To reach the total of n active constraints, $n-m$ of the remaining constraints $x_i \geq 0$, $i = 1, \ldots, n$, must be made active, which can be done by selecting $n - m$ of those to be set as $x_i = 0$. These n active constraints (the original m plus the $n - m$ variables set to zero) form a basic solution, as we have seen in the last chapter. If it happens that the m equalities can hold while the constraints $x_i \geq 0$, $i = 1, \ldots, n$, are satisfied, then we have a basic feasible solution (BFS). Theorem 3.1 summarises this process, guaranteeing that the setting of $n - m$ variables to zero will render a basic solution.

Theorem 3.1 (Linear independence and basic solutions)**.** *Consider the constraints $Ax = b$ and $x \geq 0$, and assume that A has m linearly independent (LI) rows $I = \{1, \ldots, m\}$. A vector $\overline{x} \in \mathbb{R}^n$ is a basic solution if and only if we have that $A\overline{x} = b$ and there exists indices $B(1), \ldots, B(m)$ such that*

(1) The columns $A_{B(1)}, \ldots, A_{B(m)}$ of A are LI

(2) If $j \neq B(1), \ldots, B(m)$, then $\overline{x}_j = 0$.

Proof. Assume that (1) and (2) are satisfied. Then the active constraints $\overline{x}_j = 0$ for $j \notin \{B(1), \ldots, B(m)\}$ and $Ax = b$ imply that

$$\sum_{i=1}^{m} A_{B(i)}\overline{x}_{B(i)} = \sum_{j=1}^{n} A_j\overline{x}_j = A\overline{x} = b.$$

Since the columns $\left\{A_{B(i)}\right\}_{i\in I}$ are LI, $\left\{\overline{x}_{B(i)}\right\}_{i\in I}$ are uniquely determined and thus $A\overline{x} = b$ has a unique solution, implying that $\overline{x}$ is a basic solution (cf. Theorem 2.15).

Conversely, assume that $\overline{x}$ is a basic solution. Let $\overline{x}_{B(1)}, \ldots, \overline{x}_{B(k)}$ be the nonzero components of $\overline{x}$. Thus, the system

$$\sum_{i=1}^{n} A_i\overline{x}_i = b \text{ and } \{\overline{x}_i = 0\}_{i\notin\{B(1),\ldots,B(k)\}}$$

has a unique solution, and so does $\sum_{i=1}^{k} A_{B(i)}\overline{x}_{B(i)} = b$, implying that the columns $A_{B(1)}, \ldots, A_{B(k)}$ are LI. Otherwise, there would be scalars $\lambda_1, \ldots, \lambda_k$, not all zeros, for which $\sum_{i=1}^{k} A_{B(i)}\lambda_i = 0$; this would imply that $\sum_{i=1}^{k} A_{B(i)}(\overline{x}_{B(i)} + \lambda_i) = b$, contradicting the uniqueness of $\overline{x}$.

Since $A_{B(1)}, \ldots, A_{B(k)}$ are LI, $k \leq m$. Also, since A has m LI rows, it must have m LI columns spanning $\mathbb{R}^m$. Using Theorem 2.4, we can obtain $m-k$ additional columns $A_{B(k+1)}, \ldots, A_{B(m)}$ so that $A_{B(1)}, \ldots, A_{B(m)}$ are LI.

Finally, since $k \leq m$, $\{\overline{x}_j = 0\}_{i\notin\{B(1),\ldots,B(m)\}} \subset \{\overline{x}_j = 0\}_{i\notin\{B(1),\ldots,B(k)\}}$, satisfying (1) and (2). □

The proof of Theorem 3.1 highlights an important aspect in the process of generating basic solutions. Notice that once we set $n - m$ variables to be zero, the system of equations forming P becomes uniquely determined, i.e.,

$$\sum_{i=1}^{m} A_{B(i)}\overline{x}_{B(i)} = \sum_{j=1}^{n} A_j\overline{x}_j = A\overline{x} = b.$$

3.1.2 Forming bases for standard-form linear programming problems

Theorem 3.1 provides us with a way to develop a simple procedure to generate all basic solutions of a linear programming problem in standard form:

1. Choose m LI columns $A_{B(1)}, \ldots, A_{B(m)}$;
2. Let $x_j = 0$ for all $j \notin \{B(1), \ldots, B(m)\}$;
3. Solve the system $Ax = b$ to obtain $x_{B(1)}, \ldots, x_{B(m)}$.

You might have noticed that in the proof of Theorem 3.1, the focus shifted to the columns of A rather than its rows. The reason for that is because, when we think of solving the system $Ax = b$, what we are truly doing is finding a vector x representing the linear combination of the columns of A that yield the vector b. This creates an association between the columns of A and the components of x (i.e., the variables). Notice, however, that the columns of A are not *the* variables per se, as they have dimension m (while $x \in \mathbb{R}^n$).

One important interpretation for Theorem 3.1 is that we will form bases for the column space of A by choosing m components to be nonzero in a vector of dimension n. Since $m < n$ by definition, A can only have rank $\mathbf{rank}(A) = \min\{m, n\} = m$, which happens to be the size of the row space of A (as the rows are assumed LI). This, in turn, means that both the column and the row spaces have dimension m. Thus, these bases are bases for the column space of A. Finally, finding the vector x is the same as finding how the vector b can be expressed as a linear combination of that basis. Notice that this is always possible when the basis spans $\mathbb{R}^m$ (as we have m LI column vectors) and $b \in \mathbb{R}^m$.

You will notice that from here onwards, we will implicitly refer to the columns of A as variables (although we actually mean the weight associated with that column, represented by the respective component in the variable x). Then, when we say that we are setting some $(n-m)$ of the variables to be zero, it means that we are ignoring the respective columns of A (the mapping between variables and columns being their indices: x_1 referring to the first column, x_2 to the second, and so forth), while using the remainder to form a (unique) combination that yields the vector b, being the weights of this combination precisely the solution x, which in turn represent the coordinates in $\mathbb{R}^n$ of the vertex formed by the n (m equality constraints plus $n - m$ variables set to zero) active constraints.

As we will see, this procedure will be at the core of the simplex method. Since we will often refer to elements associated with this procedure, it will be useful to define some nomenclature.

We say that $B = \left\{A_{B(i)}\right\}_{i \in I_B}$ is a *basis* (or, perhaps more precisely, a basic matrix) with basic indices $I_B = \{B(1), \ldots, B(m)\}$. Consequently, we say that the variables x_j, with $j \in I_B$, are *basic variables*. Somewhat analogously, we say that the variables chosen to be set to zero are the *nonbasic variables* x_j, with $j \in I_N$, where $I_N = J \setminus I_B$, with $J = \{1, \ldots, n\}$ being the indices of all variables (and all columns of A).

Notice that the basic matrix B is invertible since its columns are LI (c.f. Theorem 2.4). For $x_B = (x_{B(1)}, \ldots, x_{B(m)})$, the *unique solution* for $Bx_B = b$ is

$$x_B = B^{-1}b, \text{ where } B = \begin{bmatrix} | & | & | \\ A_{B(1)} & \ldots & A_{B(m)} \\ | & | & | \end{bmatrix} \text{ and } x_B = \begin{bmatrix} x_{B(1)} \\ \vdots \\ x_{B(m)} \end{bmatrix}.$$

Let us consider the following numerical example. Consider the following set P

$$P = \left\{ x \in \mathbb{R}^3 : \begin{array}{r} x_1 + x_2 + 2x_3 \leq 8 \\ x_2 + 6x_3 \leq 12 \\ x_1 \leq 4 \\ x_2 \leq 6 \\ x_1, x_2, x_3 \geq 0 \end{array} \right\}, \tag{3.1}$$

which can be written in the standard by adding slack variables $\{x_i\}_{i \in \{4,\ldots,7\}}$, yielding

$$P = \left\{ x \in \mathbb{R}^7 : \begin{array}{r} x_1 + x_2 + 2x_3 + x_4 = 8 \\ x_2 + 6x_3 + x_5 = 12 \\ x_1 + x_6 = 4 \\ x_2 + x_7 = 6 \\ x_1, \ldots, x_7 \geq 0 \end{array} \right\}. \tag{3.2}$$

The system $Ax = b$ can be represented as

$$\begin{bmatrix} 1 & 1 & 2 & 1 & 0 & 0 & 0 \\ 0 & 1 & 6 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} x = \begin{bmatrix} 8 \\ 12 \\ 4 \\ 6 \end{bmatrix}.$$

Following our notation, we have that $m = 4$ and $n = 7$. The rows of A are LI, meaning that $\mathbf{rank}(A) = 4$[1]. We can make arbitrary selections of $n - m = 3$ variables to be set to zero (i.e., nonbasic) and calculate the value of the remaining (basic) variables. For example:

- Let $I_B = \{4, 5, 6, 7\}$; in that case $x_B = (8, 12, 4, 6)$ and $x = (0, 0, 0, 8, 12, 4, 6)$, which is a basic feasible solution (BFS), as $x \geq 0$.
- For $I_B = \{3, 5, 6, 7\}$, $x_B = (4, -12, 4, 6)$ and $x = (0, 0, 4, 0, -12, 4, 6)$, which is basic but not feasible, since $x_5 < 0$.

3.1.3 Adjacent basic solutions

Now that we know how a solution can be recovered, the next important concept that we need to define is how we, from one basic solution, move to an *adjacent* solution. This will be the mechanism that the simplex method will utilise to move from one solution to the next in the search for the optimal solution.

Let us start formally defining the notion of an adjacent basic solution.

[1] You can see for yourself using Gaussian elimination or row reduction. Tip: do the elimination on the transpose $A^\top$ instead, recalling that $\mathbf{rank}(A) = \mathbf{rank}(A^\top)$.

Definition 3.2 (Adjacent basic solutions)**.** *Two basic solutions are adjacent if they share $n-1$ LI active constraints. Alternatively, two bases B_1 and B_2 are adjacent if all but one of their columns are the same.*

For example, consider the set polyhedral set P defined in (3.2). Our first BFS was defined by making $x_1 = x_2 = x_3 = 0$ (nonbasic index set $I_N = \{1, 2, 3\}$). Thus, our basis was $I_B = \{4, 5, 6, 7\}$. An adjacent basis was then formed, by replacing the basic variable x_4 with the nonbasic variable x_3, rendering the new (not feasible) basis $I_B = \{3, 5, 6, 7\}$.

Notice that the process of moving between adjacent bases has a simple geometrical interpretation. Since adjacent bases share all but one basic element, this means that the two must be connected by a line segment (in the case of the example, it would be the segment between $(0, 8)$ and $(4, 0)$, projected onto $(x_3, x_4) \in \mathbb{R}^2$, or equivalently, the line between $(0, 0, 0, 8, 12, 4, 6)$ and $(0, 0, 4, 0, -12, 4, 6)$ (Notice how the coordinate x_6 also changed values; this is necessary, so the movement is made along the edge of the polyhedral set. This will become clearer when we analyse the simplex method in further detail in Chapter 4.

3.1.4 Redundancy and degeneracy

An important underlying assumption in Theorem 3.1 is that the matrix A in the definition of the polyhedral set P is full (row) rank, that is, there are m linearly independent rows and thus m independent columns. Theorem 3.3 shows that this assumption can actually be made without loss of generality.

Theorem 3.3 (Redundant constraints)**.** *Let $P = \{x \in \mathbb{R}^n : Ax = b, x \geq 0\}$, where A is $m \times n$ matrix with rows $\{a_i\}_{i \in I}$ and $I = \{1, \ldots, m\}$. Suppose that* $\mathbf{rank}(A) = k < m$ *and that the rows $a_{i_1}, \ldots, a_{i_k}$ are LI. Then P is the same set as $Q = \left\{x \in \mathbb{R}^n : a_{i_1}^\top x = b_{i_1}, \ldots, a_{i_k}^\top x = b_{i_k}, x \geq 0\right\}$.*

Proof. Assume, without loss of generality, that $i_1 = 1$ and $i_k = k$. Clearly, $P \subset Q$, since a solution satisfying the constraints forming P also satisfies those forming Q.

As $\mathbf{rank}(A) = k$, the rows $a_{i_1}, \ldots, a_{i_k}$ form a basis in the row space of A and any row a_i, $i \in I$, can be expressed as $a_i^\top = \sum_{j=1}^k \lambda_{ij} a_j^\top$ for $\lambda_{ij} \in \mathbb{R}$.

For $y \in Q$ and $i \in I$, we have $a_i^\top y = \sum_{j=1}^k \lambda_{ij} a_j^\top y = \sum_{j=1}^k \lambda_{ij} b_j = b_i$, which implies that $y \in P$ and that $Q \subset P$. Consequently, $P = Q$. □

Theorem 3.3 implies that any linear programming problem in standard form can be reduced to an equivalent problem with linearly independent constraints. It turns out that, in practice, most professional-grade solvers (i.e., software that implements solution methods and can be used to find optimal solutions to mathematical programming models) have *preprocessing* routines to remove redundant constraints. This means that the problem is automatically treated to become smaller by not incorporating unnecessary constraints.

Degeneracy is somewhat related to the notion of redundant constraints. We say that a given vertex is a *degenerate* basic solution if it is formed by the intersection of more than n active constraints (in $\mathbb{R}^n$). Effectively, this means that more than $n - m$ variables (i.e., some of the basic variables) are set to zero, which is the main way to identify a degenerate BFS. Figure 3.1 illustrates a case in which degeneracy is present.

Notice that, while in the figure on the left, the constraint causing degeneracy is redundant, that is not the case in the figure on the righthand side. That is, redundant constraints may cause degeneracy, but not all constraints causing degeneracy are redundant.

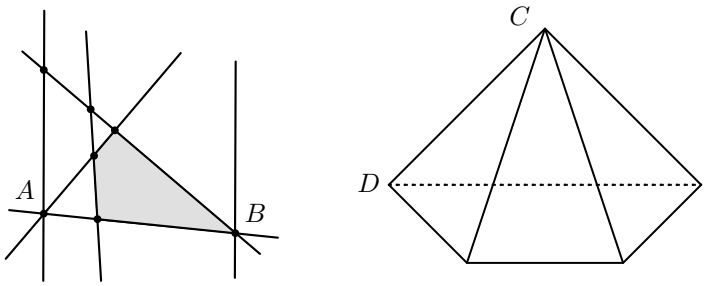


Figure 3.1: A is a degenerate basic solution, B and C are degenerate BFS, and D is a BFS.

As another example, in some cases, we may see that degeneracy is caused by the chosen representation of the problem. For example, consider the two equivalent sets:

$$P_1 = \{(x_1, x_2, x_3) : x_1 - x_2 = 0, x_1 + x_2 + 2x_3 = 2, x_1, x_3 \geq 0\} \text{ and } P_2 = P_1 \cap \{x_2 \geq 0\}.$$

The polyhedral set P_2 is equivalent to P_1 since $x_2 \geq 0$ is a redundant constraint. In that case, one can see that, while the point $(0, 0, 1)$ is not degenerate in P_1, it is in P_2, which illustrates the weak but existent relationship between redundancy and degeneracy. This is illustrated in Figure 3.2.

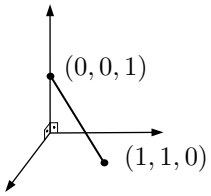


Figure 3.2: $(0, 0, 1)$ is degenerate if you add the constraint $x_2 \geq 0$.

In practice, degeneracy might cause issues related to the way we identify vertices. Because more than n active constraints form the vertex, and yet, we identify vertices by groups of n constraints to be active, it means that we might have a collection of adjacent bases that, in fact, are representing the same (vertex) point in space, meaning that we might be "stuck" for a while in the same position while scanning through adjacent bases. The numerical example below illustrates this phenomenon.

Let us consider again the example in (3.2).

$$\begin{bmatrix} 1 & 1 & 2 & 1 & 0 & 0 & 0 \\ 0 & 1 & 6 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} x = \begin{bmatrix} 8 \\ 12 \\ 4 \\ 6 \end{bmatrix}$$

Observe the following,

- let $I_B = \{1, 2, 3, 7\}$; this implies that $x = (4, 0, 2, 0, 0, 0, 6)$. There are 4 zeros (instead of $n - m = 3$) in x, which indicates degeneracy.
- Now, let $= I_B = \{1, 3, 4, 7\}$. This also implies that $x = (4, 0, 2, 0, 0, 0, 6)$. The two bases are adjacent yet represent the same point in $\mathbb{R}^7$.

As we will see, there are mechanisms that prevent the simplex method from becoming stuck on such vertices forever, an issue that is referred to as *cycling*.

3.2 Optimality of extreme points

Now that we have discussed how to algebraically represent extreme points and have seen a simple mechanism to iterate among their adjacent neighbours, the final element missing for us to be able to devise an optimisation method is to define the optimality conditions we wish to satisfy. In other words, we must define the conditions that, once satisfied, mean that we can stop the algorithm and declare the current solution optimal.

3.2.1 The existence of extreme points

First, let us define the condition that guarantees the existence of extreme points in a polyhedral set. Otherwise, there is no hope of finding an optimal solution.

Definition 3.4 (Existence of extreme points)**.** *A polyhedral set $P \subset \mathbb{R}^n$ contains a line if $P \neq \emptyset$ and there exists a nonzero vector $d \in \mathbb{R}^n$ such that $x + \lambda d \in P$ for all $\lambda \in \mathbb{R}$.*

Figure 3.3 illustrates the notion of containing a line and the existence of extreme points. Notice that if a set would have any "corner", then that would imply that the edges that form that corner prevent a direction (or a line) to be fully contained within the set. Furthermore, the said corner would be an extreme point.

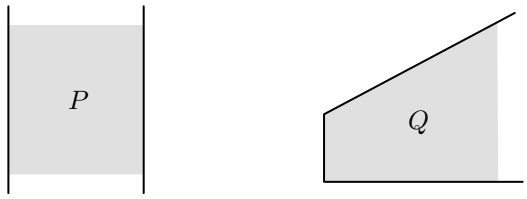


Figure 3.3: P contains a line (left) and Q does not contain a line (right)

We are now ready to pose the result that utilises Definition 3.4 to provide the conditions for the existence of extreme points.

Theorem 3.5 (Existence of extreme points)**.** *Let $P = \left\{x \in \mathbb{R}^n : a_i^\top x \geq b_i, i = 1, \ldots, m\right\} \neq \emptyset$ be a polyhedral set. Then the following are equivalent:*

(1) P has at least one extreme point;

(2) P does not contain a line;

(3) There exists n LI vectors within $\{a_i\}_{i=1}^m$.

Proof. We start with (2) $\Rightarrow$ (1), i.e., if P does not contain a line, then it must have a basic feasible solution and thus, cf. Theorem 2.15, an extreme point. Let $x \in \mathbb{R}^n$ be an element of P and let

$I = \{i : a_i^\top x = b_i s\}$. If n of the vectors a_i, $i \in I$, are linearly independent, then x is a basic feasible solution, cf. Definition 2.14.

If less than n vectors are linearly independent, then all vectors a_i, $i \in I$, lie in a proper subspace of $\mathbb{R}^n$ and, consequently, there exists a nonzero vector $d \in \mathbb{R}^n$ such that $a_i^\top d = 0$ for every $i \in I$. Consider the line consisting of all points of the form $y = x + \lambda d$, with $\lambda \in \mathbb{R}$. For $i \in I$, we have that

$$a_i^\top y = a_i^\top x + \lambda a_i^\top d = a_i^\top x = b.$$

Thus, the active constraints remain active at all points on the line. Now, by assumption, the polyhedral set contains no lines, and thus there will be values for λ for which some constraint will eventually be violated. Let $\overline{\lambda}$ be the point at which this new constraint $j \notin I$ becomes active, i.e., $a_j^\top x + \overline{\lambda} = b_j$.

We must show that a_j is not a linear combination of $\{a_i\}_{i \in I}$. We know that $a_j^\top x \neq b_j$, as $j \notin I$ and that $a_j^\top x + \overline{\lambda} = b_j$ by the definition of $\overline{\lambda}$. This implies that $a_j^\top d \neq 0$. Contrasting this with the fact that $a_i^\top d = 0$, $\forall i \in I$, we conclude that a_j is linearly independent of $\{a_i\}_{i \in I}$[2].

To conclude, we can see that by moving from x to $x + \overline{\lambda} d$, the number of linearly independent active constraints is increased by one. Repeating the see procedure as many times as needed, we will eventually reach a point having n linearly independent constraints, being thus a basic feasible solution, as we retain feasibility in the process.

To show that (1) $\Rightarrow$ (3), we simply need to notice that if P has an extreme point x, then it is also a basic feasible solution (cf. Theorem 2.15) and there are n active constraints corresponding to $\{a_i\}_{i \in I}$ linearly independent vectors.

Finally, let us show that (3) $\Rightarrow$ (2) by contradiction. Assume, without loss of generality, that $a_1, \ldots, a_n$ are the n linearly independent vectors. Suppose that P contains a line $x + \lambda d$ where $d \neq 0$. Therefore, we have that $a_i^\top x \geq b_i$, for all $i = 1, \ldots, m$ and for any λ. The only way this is possible is if $a_i^\top d = 0$ for all $i = 1, \ldots, m$, which implies that $d = 0$, reaching a contradiction that establishes that P does not contain a line. □

It turns out that linear programming problems in the standard form do not contain a line, meaning that they will always provide at least one extreme point (and, consequently, a basic feasible solution, cf. Theorem 2.15). More generally, bounded polyhedral sets do not contain a line, and neither does the positive orthant.

We are now to state the result that proves the intuition we had when analysing the plots in Chapter 1, which states that if a polyhedral set has at least one extreme point and at least one optimal solution, then there must be an optimal solution that is an extreme point.

Theorem 3.6 (Optimality of extreme points)**.** *Let $P = \{x \in \mathbb{R}^n : Ax \geq b\}$ be a polyhedral set and $c \in \mathbb{R}^n$. Consider the problem*

$$z = min. \ \{c^\top x : x \in P\}.$$

Suppose that P has at least one extreme point and that there exists an optimal solution. Then, there exists an optimal solution that is an extreme point of P.

Proof. Let $Q = \{x \in \mathbb{R}^n : Ax \geq b, c^\top x = z\}$ be the (nonempty) polyhedral set of all optimal solutions. Since $Q \subset P$ and P contains no line (cf. Theorem 3.5), Q contains no line either, and thus has an extreme point.

[2]To see that, notice that d is orthogonal to any linear combination of vectors $\{a_i\}_{i \in I}$ whilst it is not orthogonal to a_j and, thus, a_j cannot be expressed as a linear combination of $\{a_i\}_{i \in I}$.

Let $\overline{x}$ be an extreme point of Q. By contradiction, assume that $\overline{x}$ is not an extreme point of P. Then, there exist $y \neq \overline{x}$, $w \neq \overline{x}$, and $\lambda \in [0,1]$ such that $\overline{x} = \lambda y + (1-\lambda)w$. Then, $c^\top \overline{x} = \lambda(c^\top y) + (1-\lambda)c^\top w$. As $c^\top \overline{x} = z$ is optimal, we have that $z \leq c^\top y$ and $z \leq c^\top w$, and thus $z = c^\top y = c^\top w$.

Thus, $w \in Q$ and $y \in Q$, which contradicts that $\overline{x}$ is an extreme point. Thus, $\overline{x}$ must be an extreme point and, since we established that $\overline{x} \in Q$, it is also optimal. □

Theorem 3.6 is posed in a somewhat general way, which might be a source for confusion. First, recall that in the example in Chapter 1, we considered the possibility of the objective function level curve associated with the optimal value to be parallel to one of the edges of the feasible region, meaning that instead of a single optimal solution (a vertex), we would observe a line segment containing an infinite number of optimal solutions, of which exactly two would be extreme points.

This is important because we intend to design an algorithm that only inspects extreme points. This discussion guarantees that, even for the cases in which a whole set of optimal solutions exists, some elements in that set will be extreme points anyway and thus identifiable by our method.

In a more general case (with $n > 2$) it might be so that a whole facet of optimal solutions is obtained. That is precisely the polyhedral set of all optimal solutions Q in the proof. This polyhedral set will not contain a line and, therefore (cf. Theorem 3.5), have at least one extreme point.

Perhaps another important point is to notice that the result is posed assuming a minimisation problem, but it naturally holds for maximisation problems as well. Maximising a function $f(x)$ is the same as minimising $-f(x)$, with the caveat that, although the optimal value x^* is the same in both cases, the optimal values are symmetric in sign (because of the additional minus we included in the problem being originally maximised).

3.2.2 Finding optimal solutions

We now focus on the issue of being able to find and recognise extreme points as optimal solutions. In general, optimisation methods iterate the following steps:

1. Start from an initial (often feasible) solution;
2. Find a nearby solution with better value;
3. If none are available, return the best-known solution.

This very simple procedure happens to be the core idea of most optimisation methods. We will concentrate on how to identify directions of improvement and, as a consequence of their absence, how to identify optimality.

Starting from a point $x \in P$, we would like to move in a direction d that yields improvement while maintaining feasibility. Definition 3.7 provides a formalisation of this idea.

Definition 3.7 (Feasible directions)**.** *Let $x \in P$, where $P \subset \mathbb{R}^n$ is a polyhedral set. A vector $d \in \mathbb{R}^n$ is a feasible direction at x if there exists $\theta > 0$ for which $x + \theta d \in P$.*

Figure 3.4 illustrates the concept. Notice that at extreme points, the relevant feasible directions are those along the edges of the polyhedral set since those are the directions that can lead to other extreme points.

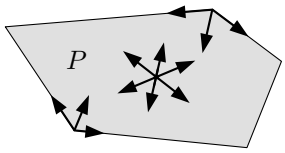


Figure 3.4: Feasible directions at different points of P

Let us now devise a way of identifying feasible directions algebraically. For that, let A be a $m \times n$ matrix, $I = [m]$ and $J = [n]$. Consider the problem

$$\text{min. } \left\{c^\top x : Ax = b, x \geq 0\right\}.$$

Let x be a basic feasible solution (BFS) with basis $B = [A_{B(1)}, \ldots, A_{B(m)}]$. Recall that the basic variables x_B are given by

$$x_B = (x_{B(i)})_{i \in I_B} = B^{-1}b, \text{ with } I_B = \{B(1), \ldots, B(m)\} \subset J,$$

and that the remaining nonbasic variables x_N are such that $x_N = (x_j)_{j \in I_N} = 0$, with $I_N = J \setminus I_B$.

Moving to a neighbouring solution can be achieved by simply moving between adjacent bases, which can be accomplished without a significant computational burden. This entails selecting a nonbasic variable x_j, with $j \in I_N$, and increasing it to a positive value θ.

Equivalently, we can define a *feasible direction* $d = [d_N, d_B]$, where d_N represent the components associated with nonbasic variables and d_B those associated with basic variables and move from the point x to the point $x + \theta d$. The components d_N associated with the nonbasic variables are thus defined as

$$d = \begin{cases} d_j = 1 \\ d_{j'} = 0, \text{for all } j' \neq j, \end{cases}$$

with $j, j' \in I_N$. Notice that, geometrically, we are moving along a line in the dimension represented by the nonbasic variable x_j.

Now, feasibility might become an issue if we are not careful to retain feasibility conditions. To retain feasibility, we must observe that $A(x + \theta d) = b$, implying that $Ad = 0$. This allows us to define the components d_B of the direction vector d that is associated with the basic variables x_j, with $j \in I_B$, since

$$0 = Ad = \sum_{j=1}^{n} A_j d_j = \sum_{i=1}^{m} A_{B(i)} d_{B(i)} + A_j = Bd_B + A_j$$

and thus $d_B = -B^{-1}A_j$ is the *basic direction* implied by the choice of the nonbasic variable x_j, $j \in I_N$, to become basic. The vector d_B can be thought of as the adjustments that must be made in the value of the other basic variables to accommodate the new variable becoming basic to retain feasibility.

Figure 3.5 provides a schematic representation of this process, showing how the change between adjacent basis can be seen as a movement between adjacent extreme points. Notice that it conveys a schematic representation of a $n = 5$ dimensional problem, in which we ignore all the m dimensions and concentrate on the $n-m$ dimensional projection of the feasibility set. This implies that the only constraints left are those associated with the nonnegativity of the variables $x \geq 0$, each associated with an edge of this alternative representation. Thus, when we set $n - m$ (nonbasic variables)

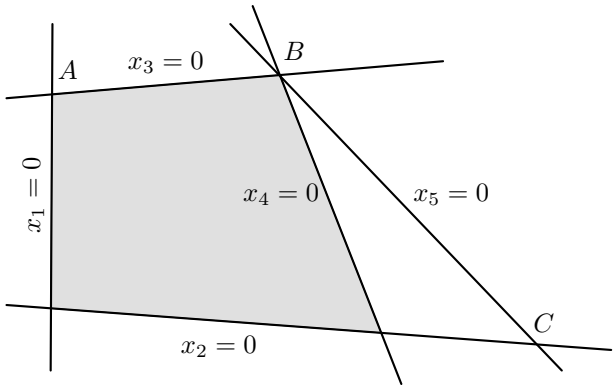


Figure 3.5: Example: $n = 5$ and $n - m = 2$. At A, $x_1 = x_3 = 0$ and $x_2, x_4, x_5 \geq 0$. Increasing x_1 while keeping x_3 zero leads to B. At B, suppose $I_N = \{3, 5\}$; by increasing x_3 while keeping x_5 zero would leads to C.

to zero, we identify an associated extreme point. As $n - m = 2$, we can plot this alternative representation on $\mathbb{R}^2$.

Overall feasibility, i.e., ensuring that $x \geq 0$, can only be retained if $\theta > 0$ is chosen appropriately small. This can be achieved if the following is observed:

1. All the other nonbasic variables remain valued at zero, that is, $x_{j'} = 0$ for $j' \in I_N \setminus \{j\}$.

2. If x is a *nondegenerate* extreme point, then all $x_B > 0$ and thus $x_B + \theta d_B \geq 0$ for appropriately small $\theta > 0$.

3. If x is a *degenerate* extreme point: $d_{B(i)}$ might not be feasible since, for some $B(i)$, if $d_{B(i)} < 0$ and $x_{B(i)} = 0$, any $\theta > 0$ will make $x_{B(i)} < 0$.

We will see later that we can devise a simply rule to define a value for θ that guarantees the above will be always observed. For now, we will put this discussion on hold and focus on the issue of how to guide the choice of which nonbasic variable x_j, $j \in I_N$, to select to become basic.

3.2.3 Moving towards improved solutions

A simple yet efficient way of deciding which nonbasic component $j \in I_N$ to make basic is to consider the immediate potential benefit that it would have for the objective function value.

Using objective function $c^\top x$, if we move along the feasible direction d as previously defined, we have that the objective function value changes by

$$c^\top d = c_B^\top d_B + c_j = c_j - c_B^\top B^{-1} A_j,$$

where $c_B = [c_{B(1)}, \ldots, c_{B(m)}]$. The quantity $c_j - c_B B^{-1} A_j$ can be used, for example, in a greedy fashion, meaning that we choose the nonbasic variable index $j \in I_N$ with greatest *potential of improvement*.

First, let us formally define this quantity, which is known as the *reduced cost*.

Definition 3.8 (Reduced cost)**.** *Let x be a basic solution associated with the basis B and let $c_B = [c_{B(1)}, \ldots, c_{B(m)}]$ be the objective function coefficients associated with the basis B. For each nonbasic variable x_j, with $j \in I_N$, we define the reduced cost $\overline{c}_j$ as*

$$\overline{c}_j = c_j - c_B^\top B^{-1} A_j.$$

The name reduced cost refers to the fact that it quantifies a cost change onto the reduced space of the basic variables. In fact, the reduced cost is calculating the change in the objective function caused by the increase in one unit of the nonbasic variable x_j elected to become basic (represented by the c_j component) and the associated change caused by the accommodation in the basic variable values to retain feasibility ($-c_B^\top B^{-1} A_j$). Therefore, the reduced cost can be understood as a *marginal value* of change in the objective function value associated with each nonbasic variable.

Let us demonstrate this with a numerical example. Consider the following linear programming problem

$$\begin{aligned} \text{min. } & 2x_1 + 1x_2 + 3x_3 + 2x_4 \\ \text{s.t.: } & x_1 + x_2 + x_3 + x_4 = 2 \\ & 2x_1 + 3x_3 + 4x_4 = 2 \\ & x_1, x_2, x_3, x_4 \geq 0. \end{aligned}$$

Let $I_B = \{1, 2\}$, yielding

$$B = \begin{bmatrix} 1 & 1 \\ 2 & 0 \end{bmatrix}.$$

Thus, $x_3 = x_4 = 0$ and $x = (1, 1, 0, 0)$. The basic direction d_B^3 for x_3 is given by

$$d_B^3 = -B^{-1} A_3 = -\begin{bmatrix} 0 & 1/2 \\ 1 & -1/2 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \end{bmatrix} = \begin{bmatrix} -3/2 \\ 1/2 \end{bmatrix}.$$

which gives $d^3 = (-3/2, 1/2, 1, 0)$. Analogously, for x_4, we have

$$d_B^4 = -B^{-1} A_4 = -\begin{bmatrix} 0 & 1/2 \\ 1 & -1/2 \end{bmatrix} \begin{bmatrix} 1 \\ 4 \end{bmatrix} = \begin{bmatrix} -2 \\ 1 \end{bmatrix}.$$

The (reduced) cost of moving along the direction given by d^3 is

$$c^\top d^3 = (c_1, c_2)^\top d_B^3 + c_3 = ((-3/2)2 + (1/2)1) + 3 = 0.5$$

while moving along d^4 has a cost of

$$c^\top d^4 = (c_1, c_2)^\top d_B^4 + c_4 = ((-2)2 + (1)1) + 2 = -1.$$

Thus, between d^3 and d^4, d^4 is a better direction since its reduced cost indicates a reduction of 1 unit of the objective function per unit of x_4. In contrast, the reduced cost associated with d^3 indicates an increase of 0.5 units of the objective function per unit of x_3, indicating that this is a direction to be avoided as we are minimising the problem. Clearly, the willingness to choose $x_{j'}$, with $j' \in I_N$, as the variable to become basic will depend on whether the scalar $c_{j'} - (c_j)_{j \in I_B}^\top d_B$ is negative (recall that we want to minimise the problem, so the smaller the total cost, the better). Another point is how large in module the reduced cost is. Recall that the reduced cost is, in fact, a measure of the marginal value associated with the increase in value of the nonbasic variable. Thus, the more negative it is, the quicker the objective function value will decrease per unit of increase of the nonbasic variable value. One interesting thing to notice is the reduced cost associated with basic variables. Recall that $B = [A_{B(1)}, \ldots, A_{B(m)}]$ and thus $B^{-1}[A_{B(1)}, \ldots, A_{B(m)}] = I$. Therefore $B^{-1} A_{B(i)}$ is the i^{th} column of I, denoted e_i, implying that

$$\overline{c}_{B(i)} = c_{B(i)} - c_B^\top B^{-1} A_{B(i)} = c_{B(i)} - c_B^\top e_i = c_{B(i)} - c_{B(i)} = 0.$$

3.2.4 Optimality conditions

Now that we have seen how to identify promising directions for improvement, we have incidentally developed a framework for identifying the optimality of a given basic feasible solution (BFS). That is, a BFS from which no direction of improvement can be found must be locally optimal. And, since local optimality implies global optimality in the presence of convexity (cf. Theorems 2.8 and 2.9), we can declare this BFS as an optimal solution.

Theorem 3.9 establishes the optimality of a BFS from which no improving feasible direction can be found without explicitly relying on the notion of convexity.

Theorem 3.9 (Optimality conditions)**.** *Consider the problem* P *: min.* $\{c^\top x : Ax = b, x \geq 0\}$*. Let* x *be the BFS associated with a basis* B *and let* $\overline{c}$ *be the corresponding vector of reduced costs.*

(1) if $\overline{c} \geq 0$*, then* x *is optimal.*

(2) if x *is optimal and nondegenerate, then* $\overline{c} \geq 0$*.*

Proof. To prove (1), assume that $\overline{c}_j \geq 0$, let y be a feasible solution to P, and $d = y - x$. We have that $Ax = Ay = b$ and thus $Ad = 0$. Equivalently:

$$Bd_B + \sum_{j \in I_N} A_j d_j = 0 \Rightarrow d_B = -\sum_{j \in I_N} B^{-1} A_j d_j,$$

implying that

$$c^\top d = c_B^\top d_B + \sum_{j \in I_N} c_j d_j = \sum_{j \in I_N} (c_j - c_B^\top B^{-1} A_j) d_j = \sum_{j \in I_N} \overline{c}_j d_j. \tag{3.3}$$

We have that $x_j = 0$ and $y_j \geq 0$ for $j \in I_N$. Thus, $d_j \geq 0$ and $\overline{c}_j d_j \geq 0$ for $j \in I_N$, which implies that $c^\top d \geq 0$ (cf. (3.3)). Consequently,

$$c^\top d \geq 0 \Rightarrow c^\top (y - x) \geq 0 \Rightarrow c^\top y \geq c^\top x,$$

i.e., x is optimal. To prove (2) by contradiction, assume that x is optimal with $\overline{c}_j < 0$ for some $j \in I_N$. Thus, we could improve on x moving along this j^{th} direction d, contradicting the optimality of x. □

A couple of remarks are worth making at this point. First, notice that, in the presence of degeneracy, it might be that x is optimal with $\overline{c}_j < 0$ for some $j \in I_N$. This is mostly caused by problems with multiple optima (i.e., the set of optimal solutions not being empty nor a singleton). Luckily, the simplex method manages to get around this issue in an effective manner, as we will see in the next chapter. Another point to notice is that, if $\overline{c}_j > 0$, $\forall j \in I_N$, then x is a *unique optimal*. Analogously, if $\overline{c} \geq 0$ with $c_j = 0$ for some $j \in I_N$, then it means that moving in that direction will cause no change in the objective function value, implying that both BFS are "equally optimal" and that the problem has multiple optimal solutions.

3.3 Exercises

Exercise 3.1: Properties of basic solutions

Prove the following theorem:

Theorem (Linear independence and basic solutions)**.** *Consider the constraints $Ax = b$ and $x \geq 0$, and assume that A has m LI rows $M = \{1, \ldots, m\}$. A vector $\overline{x} \in \mathbb{R}^n$ is a* basic solution *if and only if we have that $A\overline{x} = b$ and there exists indices $B(1), \ldots, B(m)$ such that*

(1) The columns $A_{B(1)}, \ldots, A_{B(m)}$ are LI

(2) If $j \neq B(1), \ldots, B(m)$, then $\overline{x}_j = 0$.

Note: the theorem is proved in the notes. Use this as an opportunity to revisit the proof carefully, and try to take as many steps without consulting the text as you can. This is a great exercise to help you internalise the proof and its importance in the context of the material. I strongly advise against blindly memorising it, as I suspect you will never (in my courses, at least) be requested to recite the proof literally.

Exercise 3.2: Basic solutions and extreme points

Consider the set $P = \{\mathbf{x} \in \mathbb{R}^2 : x_1 + x_2 \leq 6,\ x_2 \leq 3,\ x_1, x_2 \geq 0\}$.

(a) Enumerate all basic solutions, and identify those that are basic feasible solutions.

(b) Draw the feasible region, and identify the extreme point associated with each basic feasible solution.

(c) Consider a minimization problem with the cost vector $\mathbf{c}' = (c_1, c_2, c_3, c_4) = (-2, \frac{1}{2}, 0, 0)$. Compute the basic directions and the corresponding reduced costs of the nonbasic variables at the basic solution $\mathbf{x}' = (3, 3, 0, 0)$ with $\mathbf{x}'_B = (x_1, x_2)$ and $\mathbf{x}'_N = (x_3, x_4)$; either verify that $\mathbf{x}'$ is optimal, or move along a basic direction which leads to a better solution.

Exercise 3.3: Degeneracy - part 1

Given the linear program given below,

$$
\begin{array}{lrcl}
\max & 2x_1 + x_2 & & \\
\text{s.t.} & & & \\
& 2x_1 + 2x_2 & \leq & 9 \\
& 2x_1 - x_2 & \leq & 3 \\
& x_1 - x_2 & \leq & 1 \\
& 4x_1 - 3x_2 & \leq & 5 \\
& x_1 & \leq & 2.5 \\
& x_2 & \leq & 4 \\
& & & \\
& x_1, x_2 \geq 0 & &
\end{array}
$$

(a) Plot the constraints and find the degenerate basic feasible solutions.

(b) What are the bases forming the degenerate solutions?

Exercise 3.4: Feasible direction

Let x be a point in a polyhedron $P = \{x \in \mathbb{R}^n : Ax = b, x \geq 0\}$. Show that a vector $d \in \mathbb{R}^n$ is a feasible direction at $x \in P$ if and only if $Ad = 0$ and $d_i \geq 0$ for all i for which $x_i = 0$. A feasible direction of P at point x is a vector $d \neq 0$ such that $x + \theta d \in P$ for some $\theta > 0$.

Exercise 3.5: Optimality of extreme points

Prove the following theorem.

Theorem (Optimality of extreme points)**.** *Let $P = \{x \in \mathbb{R}^n : Ax \geq b\}$ be a polyhedral set and $c \in \mathbb{R}^n$. Consider the problem*

$$z = min. \left\{ c^\top x : x \in P \right\}.$$

Suppose that P has at least one extreme point and that there exists an optimal solution. Then, there exists an optimal solution that is an extreme point of P.

Note: see Exercise 3.1.

Chapter 4

The simplex method

4.1 Developing the simplex method

In Chapter 3, we discussed all the necessary technical aspects required to develop the simplex method. In this chapter, we will concentrate on operationalising the method from a computational standpoint.

4.1.1 Calculating step sizes

One discussion that we purposely delayed was that of how to define the value of the step size θ to be taken in the feasible direction d. Let $c \in \mathbb{R}^n$, A be a $m \times n$ full-rank matrix, b a nonnegative m-sized vector[1] and $J = [n]$. Consider the linear programming problem P in the standard form

$$(P) : \text{min. } \left\{c^\top x : Ax = b,\ x \geq 0\right\}.$$

Building upon the elements we defined in Chapter 3, employing the simplex method to solve P consists of the following set of steps:

1. Start from a nondegenerate basic feasible solution (BFS)
2. Find a negative reduced cost component $\overline{c}_j$. If $\overline{c} \geq 0$, return the current solution.
3. Move along the feasible direction $d = (d_B, d_N)$, where $d_j = 1$, $d_{N\setminus\{j\}} = 0$ and $d_B = -B^{-1}A_j$.

Moving along the feasible direction d towards $x + \theta d$ (with scalar $\theta > 0$) makes $x_j > 0$ (i.e., $j \in I_N$ enters the basis) while reducing the objective value at a rate of $\overline{c}_j$. Thus, one should move as far as possible (say, take a step of length $\overline{\theta}$) along the direction d, which incurs on an objective value change of $\overline{\theta}(c^\top d) = \overline{\theta}\overline{c}_j$.

Moving as far along the feasible direction d as possible while observing that feasibility is retained is equivalent to setting $\overline{\theta}$ as

$$\overline{\theta} = \max\left\{\theta \geq 0 : A(x + \overline{\theta}d) = b,\ x + \overline{\theta}d \geq 0\right\}.$$

Recall that, by construction, the feasible direction d, we have that $Ad = 0$ and thus $A(x + \overline{\theta}d) = Ax = b$. Therefore, the only feasibility condition that can be violated when setting $\overline{\theta}$ too large is the nonnegativity of all variables, i.e., $x + \overline{\theta}d \geq 0$.

[1] Notice that this can be assumed without loss of generality, by multiplying both sides by a constant, which does not change the constraint.

To prevent this from being the case, all basic variables $i \in I_B$ for which the component in the basic direction vector d_B is negative must be guaranteed to retain

$$x_i + \overline{\theta} d_i \geq 0 \Rightarrow \overline{\theta} \leq -\frac{x_i}{d_i}.$$

Therefore, the maximum value $\overline{\theta}$ is that that can be increased until the first component of x_B turns zero. Or, more precisely put,

$$\overline{\theta} = \min_{i \in I_B : d_i < 0} \left\{ -\frac{x_{B(i)}}{d_{B(i)}} \right\}.$$

Notice that we only need to consider those basic variables with components d_i, $i \in I_B$, that are negative. This is because, if $d_i \geq 0$, then $x_i + \overline{\theta} d_i \geq 0$ holds for any value of $\overline{\theta} > 0$. This means that the constraints associated with these basic variables (referring to the representation in Figure 3.5) do not limit the increase in value of the select nonbasic variable. Notice that this can lead to a pathological case in which none of the constraints limits the increase in value of the nonbasic variable, which indicates that the problem has an unbounded direction of decrease for the objective function. In this case, we say that the problem is *unbounded*.

Another important point is the assumption of a nondegenerate BFS. The nondegeneracy of the BFS implies that $x_{B(i)} > 0$, $\forall i \in I_B$, and, thus, $\overline{\theta} > 0$. In the presence of degeneracy, one can infer that the definition of the step size $\overline{\theta}$ must be done more carefully.

Let us consider the numerical example we used in Chapter 3 with a generic objective function.

$$\begin{aligned} \text{min. } & c_1 x_1 + c_2 x_2 + c_3 x_3 + c_4 x_4 \\ \text{s.t.: } & x_1 + x_2 + x_3 + x_4 = 2 \\ & 2x_1 + 3x_3 + 4x_4 = 2 \\ & x_1, x_2, x_3, x_4 \geq 0. \end{aligned}$$

Let $c = (2, 0, 0, 0)$ and $I_B = \{1, 2\}$. The reduced cost of the nonbasic variable x_3 is

$$\overline{c}_3 = c_3 - (c_1, c_2)^\top [-3/2, 1/2] = -3.$$

where $d_B = [-3/2, 1/2]$. As x_3 increases in value, only x_1 decreases, since $d_1 < 0$. Therefore, the largest $\overline{\theta}$ for which $x_1 \geq 0$ is $-(x_1/d_1) = 2/3$. Notice that this is precisely the value that makes $x_1 = 0$, i.e., nonbasic. The new basic variable is now $x_3 = 2/3$, and the new (adjacent, as we will see next) extreme point is

$$\overline{x} = x + \theta d = (1, 1, 0, 0) + (2/3)(-3/2, 1/2, 1, 0) = (0, 4/3, 2/3, 0).$$

4.1.2 Moving between adjacent bases

Once we have defined the optimal step size $\overline{\theta}$, we move to a new BFS $\overline{x}$. That new solution is such that, for the nonbasic variable $j \in I_N$ selected to be basic, we observe that $\overline{x}_j = \theta$. Now, let us define as l the index of the basic variable that first becomes zero, that is, the variable that defines the value of $\overline{\theta}$. More precisely put, let

$$l = \operatorname*{argmin}_{i \in I_B : d_i < 0} \left\{ -\frac{x_{B(i)}}{d_{B(i)}} \right\} \text{ and, thus, } \overline{\theta} = \left\{ -\frac{x_{B(l)}}{d_{B(l)}} \right\}. \tag{4.1}$$

By moving to the BFS $\overline{x}$ by making $\overline{x} = x + \overline{\theta}d$, we are in fact moving from the basis B to an adjacent basis $\overline{B}$, defined as

$$\overline{B} = \begin{cases} \overline{B}(i) = B(i), \text{ for } i \in I_B \setminus \{l\} \\ \overline{B}(i) = j, \text{ for } i = l. \end{cases}$$

Notice that the new basis $\overline{B}$ only has one pair of variables swapped between basic and nonbasic when compared against B. Analogously, the basic matrix associated with $\overline{B}$ is given by

$$\begin{bmatrix} | & & | & | & | & & | \\ A_{B(1)} & \dots & A_{B(l-1)} & A_j & A_{B(l+1)} & \dots & A_{B(m)} \\ | & & | & | & | & & | \end{bmatrix},$$

where the middle column representing that the column $A_{B(l)}$ has been replaced with A_j.

Theorem 4.1 provides the technical result that formalises our developments so far.

Theorem 4.1 (Adjacent bases)**.** *Let A_j be the column of the matrix A associated with the selected nonbasic variable index $j \in I_N$. And let l be defined as* (4.1)*, with $A_{B(i)}$ being its respective column in A. Then*

(1) The columns $A_{B(i)}$ and A_j are linearly independent. Thus, $\overline{B}$ is a basic matrix;

(2) The vector $\overline{x} = x + \overline{\theta}d$ is a BFS associated with $\overline{B}$.

Proof. We start by proving (1). By contradiction, assume that $\left\{A_{B(i)}\right\}_{i \in I_B \setminus \{l\}}$ and A_j are not linearly independent. Thus, there exist $\{\lambda_i\}_{i=1}^m$ (not all zeros) such that

$$\sum_{i=1}^{m} \lambda_i A_{\overline{B}(i)} = 0 \Rightarrow \sum_{i=1}^{m} \lambda_i B^{-1} A_{\overline{B}(i)} = 0,$$

making $B^{-1}A_{\overline{B}(i)}$ not linearly independent. However, $B^{-1}A_{\overline{B}(i)} = B^{-1}A_{B(i)}$ for $i \in I_B \setminus \{l\}$ and thus are all unit vectors e_i with the l^{th} component zero.

Now, $B^{-1}A_j = -d_B$, with component $d_{B(l)} \neq 0$, is linearly independent from $B^{-1}A_{B(i)} = B^{-1}A_{\overline{B}(i)}$. Thus, $\left\{A_{\overline{B}(i)}\right\}_{i \in I_{\overline{B}}} = \left\{A_{B(i)}\right\}_{i \in B \setminus \{l\}} \cup \{A_j\}$ are linearly independent, forming the contradiction.

Now we focus on proving (2). We have that $\overline{x} \geq 0$, $A\overline{x} = b$ and $\overline{x}_j = 0$ for $j \in I_{\overline{N}} = J \setminus I_{\overline{B}}$. This, combined with $\left\{\overline{B}(i)\right\}_{i \in I_{\overline{B}}}$ being linearly independent (cf. (1)), completes the proof. □

We have finally compiled all the elements we need to state the simplex method pseudocode, which is presented in Algorithm 1. One technical detail in the presentation of the algorithm is the use of the auxiliary vector u. This allows for the precalculation of the components of $d_B = -B^{-1}A_j$ (notice the changed sign) to test for unboundedness, that is, the lack of a constraint (and associated basic variable) that can limit the increase of the chosen nonbasic variable.

The last missing element is proving that Algorithm 1 eventually converges to an optimal solution, should one exist. This is formally stated in Theorem 4.2.

Theorem 4.2 (Convergence of the simplex method)**.** *Assume that P has at least one feasible solution and that all BFS are nondegenerate. Then, the simplex method terminates after a finite number of iterations, in one of the following states:*

(1) The basis B and the associated BFS are optimal; or

(2) d is such that $Ad = 0$, $d \geq 0$, and $c^\top d < 0$, with optimal value $-\infty$.

Proof. If the condition in Line 2 of Algorithm 1 is not met, then B and associated BFS are optimal, c.f. Theorem 3.9. Otherwise, if Line 4 condition is met, then d is such that $Ad = 0$ and $d \geq 0$, implying that $x + \theta d \in P$ for all $\theta > 0$, and a value reduction $\theta\overline{c}$ of $-\infty$.

Finally, notice that $\overline{\theta} > 0$ step sizes are taken along d satisfy $c^\top d < 0$. Thus, the value of successive solutions is strictly decreasing and no BFS can be visited twice. As there is a finite number of BFS, the algorithm must eventually terminate. □

Algorithm 1 Simplex method

1: **initialise.** Initial basis B, associated BFS x, and reduced costs $\overline{c}$.
2: **while** $\overline{c}_j < 0$ for some $j \in I_N$ **do**
3: Choose some j for which $\overline{c_j} < 0$. Calculate $u = B^{-1}A_j$.
4: **if** $u \leq 0$ **then**
5: **return** $z = -\infty$.
6: **else**
7: $\overline{\theta} = \min_{i \in I_B : u_i > 0} \left\{ \frac{x_{B(i)}}{u_i} \right\}$ and $l = \operatorname{argmin}_{i \in I_B : u_i > 0} \left\{ \frac{x_{B(i)}}{u_i} \right\}$
8: Set $x_j = \overline{\theta}$ and $x_B = x - \overline{\theta}u$. Form new basis $I_B = I_B \setminus \{l\} \cup \{j\}$.
9: Calculate $\overline{c}_j = c_j - c_B^\top B^{-1} A_j$ for all $j \in I_N$.
10: **end if**
11: **end while**
12: **return** optimal basis I_B and optimal solution x.

4.1.3 A remark on degeneracy

We now return to the issue related to degeneracy. As we discussed earlier, degeneracy is an important pitfall for the simplex method. To recognise that the method arrived at a degenerate BFS, one must observe how the values of the basic variables are changing. If, for $\overline{\theta}$, more than one basic variable becomes zero at $\overline{x} = x + \overline{\theta}d$, then $\overline{B}$ is degenerate.

Basically, if the current BFS is degenerate, $\overline{\theta} = 0$ can happen when $x_{B(l)} = 0$ and the component $d_{B(l)} < 0$. Notice that a step size of $\overline{\theta} = 0$ is the only option to prevent infeasibility in this case. Nevertheless, a new basis can still be defined by replacing $A_{B(l)}$ with A_j in B. However, $\overline{x} = x + \overline{\theta}d = x$. Sometimes, even though the method is effectively staying at the same extreme point, changing the basis on a degenerate solution might eventually expose a direction of improvement, a phenomenon that is called *stalling*. In an extreme case, it might be so that the selection of the next basic variable is such that the same extreme point is recovered over and over again, which is called *cycling*. The latter can be prevented by a specific technique for carefully selecting the variable that will enter the basis.

Figure 4.1 illustrates an example of stalling. In that, a generic problem with five variables is illustrated, with any given basis being formed by three variables. For example, at y, $I_B = 3, 4, 5$, with $x_3 > 0$, $x_4 > 0$, and $x_5 > 0$. Notice that there are multiple bases representing x. Suppose we have $I_B = \{1, 2, 3\}$. Notice that in this case, we have $x_2 = x_3 = 0$ even though x_2 and x_3 are basic variables. Now, suppose we perform one simplex iteration and move to the adjacent basis $I_B = \{1, 3, 4\}$). Even though the extreme point is the same (x), this new basis exposes

the possibility of moving to the nondegenerate basis $I_B = \{3, 4, 5\}$, i.e., y. Assuming that we are minimising, any direction with a negative component in the y-axis direction will represent an improvement in the objective function (notice the vector c, which is parallel to the y-axis and points upwards). Thus, the method is only stalled by the degeneracy of x but does not cycle.

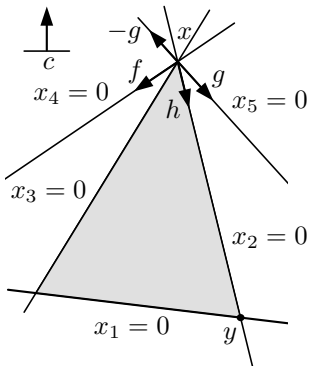


Figure 4.1: $I_N = \{4, 5\}$ for x; f ($x_5 > 0$) and g ($x_4 > 0$) are basic directions. Making $I_N = \{2, 5\}$ lead to new basic directions h ($x_4 > 0$) and $-g$ ($x_2 > 0$).

4.2 Implementing the simplex method

We now focus on some specific details related to alternative simplex method implementations. In a general sense, implementations of the simplex method vary in terms of how the selection of the nonbasic variables with $\overline{c}_j < 0$ that enter the basis is made. Different ways of selecting the basic variable $l = \text{argmin}_{i \in B | d_i < 0} \left\{ -x_{B(i)}/d_{B(i)} \right\}$ to leave the basis in case of ties might be of interest, especially in the attempt of preventing cycling.

Another important aspect related to implementations of the simplex method is how matrices are represented, its consequences on memory utilisation, and how the operations related to matrix inversion are carried out.

4.2.1 Pivot or variable selection

The rules utilised for making choices regarding entering and leaving variables are generally referred to as *pivoting rules*. However, the term most commonly used to refer to the selection of nonbasic variables to enter the basis is *(simplex) pricing rules*. Recall that these are applied to choose between those variables that have favourable (negative, for minimisation) reduced costs.

- *Greedy selection* (or Dantzig's rule): choose x_j, $j \in I_N$, with largest $|\overline{c}_j|$. Prone to cycling.
- *Index-based order* (or Bland's rule): choose x_j, $j \in I_N$, with smallest j. It prevents cycling but is computationally inefficient.
- *Reduced cost pricing*: calculate $\overline{\theta}$ for all (or some) $j \in N$ and pick smallest (i.e., largest in module with negative sign) $\overline{\theta}\overline{c}_j$. Calculating the actual observed change for all nonbasic variables is too computationally expensive. Partial pricing refers to only considering a subset of the nonbasic variables to calculate $\overline{\theta}\overline{c}_j$.

- *Devex*[2] and *steepest-edge*[3]: most commonly used by modern implementations of the simplex method, available in professional-grade solvers.

4.2.2 The revised simplex method

The central element in the simplex method is the calculation of the matrix $B^{-1}A_j$, from which the reduced cost vector $\bar{c}_j$, $j \in I_N$, the basic feasible direction vector d_B and the step size $\overline{\theta}$ can be easily computed.

First, let us consider a more natural way of implementing the simplex method so then we can point out how the method can be revised to be more computationally efficient. We will refer to this version as the "naive simplex". The main differences between the naive and its revised version will be how $B^{-1}A_j$ is computed and the amount of information being carried over between iterations.

A somewhat natural way to implement the simplex method would be to store the term $p^\top = c_B^\top B^{-1}$ in an auxiliary variable by solving the linear system $p^\top B = c_B^\top$. These terms are important, as we will see later, and they are often referred to as the *simplex multipliers*.

Once the simplex multipliers p are available, the reduced cost c_j associated with the nonbasic variable index $j \in I_N$ is simply

$$\bar{c}_j = c_j - p^\top A_j.$$

Once the column A_j is selected, we can then solve a second linear system $Bu = A_j$ to determine $u = B^{-1}A_j$.

The key observation that can yield computational savings is that we do not need to solve two linear systems. As one can notice, there is a common term between the two, the inverse of the basic matrix B^{-1}. If this matrix can be made available at the beginning of each iteration, then the terms $c_B^\top B^{-1}$ and $B^{-1}A_j$ can be easily and more cheaply (computationally) obtained.

For that to be possible, we need an efficient method of updating the matrix B^{-1} after each iteration. To see how that can be accomplished, recall that

$$\begin{aligned} B &= [A_{B(1)}, \ldots, A_{B(m)}], \text{ and} \\ \overline{B} &= [A_{B(1)}, \ldots, A_{B(l-1)}, A_j, A_{B(l+1)} \ldots, A_{B(m)}], \end{aligned}$$

where the l^{th} column $A_{B(l)}$ is precisely how the adjacent bases B and $\overline{B}$ differ, with A_j replacing $A_{B(l)}$ in $\overline{B}$.

We can devise an efficient manner to update B^{-1} into $\overline{B}^{-1}$ utilising *elementary row operations*. First, let us formally define the concept.

Definition 4.3 (Elementary row operations)**.** *Adding a constant multiple of one row to the same or another row is called an elementary row operation.*

Defining elementary row operations is equivalent to devising a matrix $Q = I + D$ to premultiply B, where

$$D = \begin{cases} D_{ij} = \beta, \\ D_{i'j'} = 0 \text{ for all } i', j' \neq i, j. \end{cases}$$

[2] P. M. J. Harris (1973), Pivot Selection Methods in the Devex LP Code, Math. Prog., 57, 341–374.
[3] J. Forrest & D. Goldfarb (1992), Steepest-Edge Simplex Algorithms for LP, Math. Prog., 5, 1–28.

Calculating $QB = (I + D)B$ is the same as having the j^{th} row of B multiplied by a scalar β and then having the resulting j^{th} row added to the i^{th} row of B. Before we continue, let us utilise a numerical example to clarify this procedure. Let

$$B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

and suppose we would like to multiply the third row by 2 and have it then added to the first row. That means that $D_{13} = 2$ and that $Q = I + D$ would be

$$Q = \begin{bmatrix} 1 & & 2 \\ & 1 & \\ & & 1 \end{bmatrix}$$

Then premultiplying B by Q yields

$$QB = \begin{bmatrix} 11 & 14 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}.$$

As a side note, we have that Q^{-1} exists since $\mathbf{det}(Q) = 1$. Furthermore, sequential elementary row operations $\{1, 2, \ldots, k\}$ can be represented as $Q = Q_1 Q_2, \ldots Q_k$.

Going back to the purpose of updating B^{-1} into $\overline{B}^{-1}$, notice the following. Since $B^{-1}B = I$, each term $B^{-1}A_{B(i)}$ is the i^{th} unit vector e_i (the i^{th} column of the identity matrix). That is,

$$B^{-1}\overline{B} = \begin{bmatrix} | & & | & | & | & & | \\ e_1 & \ldots & e_{l-1} & u & e_{l+1} & \ldots & e_m \\ | & & | & | & | & & | \end{bmatrix} = \begin{bmatrix} 1 & & u_1 & & \\ & \ddots & \vdots & & \\ & & u_l & & \\ & & \vdots & \ddots & \\ & & u_m & & 1 \end{bmatrix},$$

where $u = B^{-1}A_j$. We want to define an elementary row operation matrix Q such that $QB^{-1} = \overline{B}^{-1}$, or $QB^{-1}\overline{B} = I$. Therefore Q will be such that the elementary row operations turn $B^{-1}\overline{B}$ into an identity matrix, i.e., that turn the vector u into the unit vector e_l.

The main trick is that we do not need matrix multiplication to achieve it, considerably decreasing the computational burden. Instead, we can simply apply the elementary row operations, focusing only on the column u and the operations required to turn it into the unit vector e_l. This can be achieved by:

1. First, for each $i \neq l$, multiply the l^{th} row by $-\frac{u_i}{u_l}$ and add to the i^{th} row. This replaces u_i with zero for all $i \in I \setminus \{l\}$.

2. Then, divide the l^{th} row by u_l. This replaces u_l with one.

We can restate the simplex method in its revised form. This is presented in Algorithm 2.

Notice that in Algorithm 2, apart from the initialisation step, no linear systems are directly solved. Instead, elementary row operations (ERO) are performed, leading to computational savings.

Algorithm 2 Revised simplex method

initialise. Initial basis B, associated BFS x, and B^{-1} are available.
Calculate $p^\top = c_B^\top B^{-1}$ and $\overline{c}_j = c_j - p^\top A_j$ for $j \in I_N$.
while $\overline{c}_j < 0$ for some $j \in N$ **do**
    Choose some j for which $\overline{c_j} < 0$. Calculate $u = B^{-1}A_j$.
    if $u \leq 0$ **then**
        return $z = -\infty$.
    else
        $\overline{\theta} = \min_{i \in I_B : u_i > 0} \left\{ \frac{x_{B(i)}}{u_i} \right\}$ and $l = \operatorname{argmin}_{i \in I_B : u_i > 0} \left\{ \frac{x_{B(i)}}{u_i} \right\}$
        Set $x_j = \overline{\theta}$ and $x_B = x - \overline{\theta} u$.
        Form the matrix $[B^{-1} \,|\, u]$ and perform EROs to convert it to $[\overline{B}^{-1} \,|\, e_l]$.
        Make $I_B = I_B \setminus \{l\} \cup \{j\}$ and $B^{-1} = \overline{B}^{-1}$.
        Calculate $p^\top = c_B^\top B^{-1}$ and $\overline{c}_j = c_j - p^\top A_j$ for all $j \in I_N = J \setminus I_B$.
    end if
end while
return optimal basis I_B and optimal solution x.

The key feature of the revised simplex method is a matter of representation and, thus, memory allocation savings. Algorithm 2 only requires keeping in memory a matrix of the form

$$\begin{bmatrix} p & | & p^\top b \\ B^{-1} & | & u \end{bmatrix}$$

which, after each series of elementary row operations, yields not only $\overline{B}^{-1}$ but also the updated simplex multipliers $\overline{p}$ and $\overline{p}^\top b = c_B^\top \overline{B}^{-1} b = c_B^\top \overline{x}_B$, which represents the objective function value of the new basic feasible solution $\overline{x} = [\overline{x}_B, \overline{x}_N]$. These savings will become obvious once we discuss the tabular (or non-revised) version of the simplex method.

Three main issues arise when considering the efficiency of implementations of the simplex method, namely, matrix (re)inversion, representation in memory, and the use of matrix decomposition strategies.

- *Reinversion:* localised updates such as EROs have the side effect of accumulating truncation and round-off error. To correct this, solvers typically rely on periodically recalculating B^{-1}, which, although costly, can avoid numerical issues.
- *Representation:* A sparse representation of $Q_n = Q_1 Q_2 \dots Q_{k-1}$ can be kept instead of updating B^{-1}. Recall that $u = \overline{B}^{-1} A_j = Q B^{-1} A_j$. For larger problems, this means a trade-off between memory allocation and the number of (sparse) matrix-matrix multiplications.
- *Decomposition:* Decomposed (e.g., LU decomposition) forms of B are used to improve efficiency in storage and the solution of the linear systems to exploit the typical sparsity of linear programming problems.

4.2.3 Tableau representation

The tableau representation of the simplex method is useful as a concept presentation tool. It consists of a naive memory-intensive representation of the problem elements as they are iterated

between each basic feasible solution. However, it is a helpful representation from a pedagogical standpoint and will be useful for explaining further concepts in the upcoming chapters.

In contrast to the revised simplex method, instead of updating only B^{-1}, we consider the complete matrix

$$B^{-1}[A \mid b] = \left[B^{-1}A_1, \ldots, B^{-1}A_n \mid B^{-1}b\right].$$

Furthermore, we adjoint a row representing the reduced cost vector $\overline{c}^\top = c^\top - c_B^\top B^{-1}A$ and the negative of the objective function value for the current basis, $-c_B x_B = -c_B^\top B^{-1}b$, a row often referred to as the *zeroth row*. The reason why we consider the negative sign is that it allows for a simple updating rule for the zeroth row, by performing elementary row operations to make the j^{th} element, associated with the nonbasic variable in B that becomes basic in $\overline{B}$, zero.

The full tableau representation is given by

$$\begin{array}{|c|c|}\hline c^\top - c_B^\top B^{-1}A & -c_B^\top B^{-1}b \\ \hline B^{-1}A & B^{-1}b \\ \hline \end{array} \Rightarrow \begin{array}{|ccc|c|}\hline \overline{c}_1 & \cdots & \overline{c}_n & -c_B x_B \\ \hline | & & | & x_{B(1)} \\ B^{-1}A_1 & \cdots & B^{-1}A_n & \vdots \\ | & & | & x_{B(m)} \\ \hline \end{array}$$

In this representation, we say that the j^{th} column associated with the nonbasic variable to become basic is the *pivot column* u. Notice that, since the tableau exposes the reduced costs $\overline{c}_j$, $j \in I_N$, it allows for trivially applying the greedy pricing strategy (by simply choosing the variables with a negative reduced cost with the largest absolute value).

The l^{th} row associated with the basic variable selected to leave the basis is the *pivot row*. Again, the tableau representation facilitates the calculation of the ratios used in choosing the basic variable l to leave the basis since it amounts to simply calculating the ratios between the elements on the rightmost column and those in the pivot column, disregarding those that present entries less or equal than zero and the zeroth row. The row with the minimal ratio will be the row associated with the current basic variable leaving the basis.

Once a pivot column and a pivot row have been defined, it is a matter of performing elemental row operations utilising the pivot row to turn the pivot column into the unit vector e_l and turn to zero the respective zeroth element (recall that basic variables have zero reduced costs). This is the same as using elementary row operations using the pivot row to turn all elements in the pivot column zero, except for the *pivot element* u_l, which is the intersection of the pivot row and the pivot column, that must be turned into one. The above highlights the main purpose of the tableau representation, which is to facilitate calculation by hand.

Notice that, as we have seen before, performing elementary row operations to convert the pivot column u into e_l converts $B^{-1}[A \mid b]$ into $\overline{B}^{-1}[A \mid b]$. Analogously, turning the entry associated with the pivot column u in the zeroth row to zero converts $[c^\top - c_B^\top B^{-1}A \mid -c_B^\top B^{-1}b]$ into $[c^\top - c_{\overline{B}}^\top \overline{B}^{-1}A \mid -c_{\overline{B}}^\top \overline{B}^{-1}b]$.

Let us return to the paint factory example from Section 1.2.1, which in its standard form can be

written as

$$\begin{aligned} \text{max. } & z = 5x_1 + 4x_2 && (4.2)\\ \text{s.t.: } & 6x_1 + 4x_2 + x_3 = 24 && (4.3)\\ & x_1 + 2x_2 + x_4 = 6 && (4.4)\\ & x_2 - x_1 + x_5 = 1 && (4.5)\\ & x_2 + x_6 = 2 && (4.6)\\ & x_1, \ldots, x_6 \geq 0. && (4.7) \end{aligned}$$

The sequence of tableaus for this problem is given by

	x_1	x_2	x_3	x_4	x_5	x_6	RHS
z	-5	-4	0	0	0	0	0
x_3	**6**	4	1	0	0	0	24
x_4	1	2	0	1	0	0	6
x_5	-1	1	0	0	1	0	1
x_6	0	1	0	0	0	1	2

	x_1	x_2	x_3	x_4	x_5	x_6	RHS
z	0	-2/3	5/6	0	0	0	20
x_1	1	2/3	1/6	0	0	0	4
x_4	0	**4/3**	-1/6	1	0	0	2
x_5	0	5/3	1/6	0	1	0	5
x_6	0	1	0	0	0	1	2

	x_1	x_2	x_3	x_4	x_5	x_6	RHS
z	0	0	3/4	1/2	0	0	21
x_1	1	0	1/4	-1/2	0	0	3
x_2	0	1	-1/8	3/4	0	0	3/2
x_5	0	0	3/8	-5/4	1	0	5/2
x_6	0	0	1/8	-3/4	0	1	1/2

The bold terms in the tableau represent the pivot elements at each iteration, i.e., the intersection of the pivot column and row. From the last tableau, we see that the optimal solution is $x^* = (3, 3/2)$. Notice that we applied a change in signs in the objective function coefficients, turning it into a minimisation problem; also notice that this makes the values of the objective function in the RHS column appear as positive, although it should be negative as we are in fact minimising $-5x_1 - 4x_2$, for which $x^* = (3, 3/2)$ evaluates as -21. As we have seen, the tableau shows $-c_B x_B$, hence why the optimal tableau displays 21 in the first row of the RHS column.

4.2.4 Generating initial feasible solutions (two-phase simplex)

We now consider the issue of converting general linear programming problems into the standard form they are assumed to be for the simplex method. As we mentioned before, problems with constraints of the form $Ax \leq b$ can be converted to standard form by simply adding nonnegative *slack variables* $s \geq 0$ (recall that $b \geq 0$ can be assumed without loss of generality). In addition, we can trivially obtain an initial basic feasible solution (BFS) with $(x, s) = (0, b)$, with $B = I$, as

$$Ax \leq b \Rightarrow Ax + s = b.$$

Notice that this is equivalent to assuming all original problem variables (i.e., those that are not slack variables) to be initialised as zero (i.e., nonbasic) since this is a trivially available initial feasible solution. However, this approach does not work for constraints of the form $Ax \geq b$, as in this case, the transformation would take the form

$$Ax \geq b \Rightarrow Ax - s = b \Rightarrow Ax - s = b.$$

Notice that making the respective slack variable basic would yield an initial value of $-b$, making the basic solution not feasible.

For more general problems, however, this might not be possible since simply setting the original problem variables to zero might not yield a feasible solution that can be used as a BFS. To circumvent that, we rely on *artificial variables* to obtain a BFS.

Let $P : \text{min. } \{c^\top x : Ax = b, x \geq 0\}$, which can be achieved with appropriate transformation (i.e., adding nonnegative slack to the inequality constraints) and assumed (without loss of generality) to have $b \geq 0$. Then, finding a BFS for P amounts to finding a zero-valued optimal solution to the *auxiliary problem*

$$\begin{aligned} (AUX) : \text{min. } & \sum_{i=1}^{m} y_i \\ \text{s.t.: } & Ax + y = b \\ & x, y \geq 0. \end{aligned}$$

The auxiliary problem AUX is formed by including one artificial variable for each constraint in P, represented by the vector y of so-called *artificial variables*. Notice that the problem is represented in a somewhat compact notation, in which we assume that all slack variables used to convert inequalities into equalities have already been incorporated in the vector x and matrix A, with the artificial variables y playing the role of "slacks" in AUX that can be assumed to be basic and trivially yield an initial BFS for AUX. In principle, one does not need artificial variables for the rows in which there is a positive signed slack variable (i.e., an originally less-or-equal-than constraint), but this representation allows for compactness.

Solving AUX to optimality consists of trying to find a BFS in which the value of the artificial variables is zero since, in practice, the value of the artificial variables measures a degree of infeasibility of the current basis in the context of the original problem P. This means that a BFS in which the artificial variable plays no roles was found and can be used as an initial BFS for solving P. On the other hand, if the optimal for AUX is such that some of the artificial variables are nonzero, then this implies that there is no BFS for AUX in which the artificial variables are all zero, or, more specifically, there is no BFS for P, indicating that the problem P is *infeasible*.

Assuming that P is feasible and $\overline{y} = 0$, two scenarios can arise. The first is when the optimal basis B for AUX is composed only of columns A_j of the original matrix A, with no columns associated with the artificial variables. Then B can be used as an initial starting basis without any issues.

The second scenario is somewhat more complicated. Often, AUX is a degenerate problem and the optimal basis B may contain some of the artificial variables y. This then requires an additional preprocessing step, which consists of the following:

(1) Let $A_{B(1)}, \ldots, A_{B(k)}$ be the columns A in B, which are linearly independent. We know from earlier (c.f. Theorem 2.4) that, being A full-rank, we can choose additional columns $A_{B(k+1)}, \ldots, A_{B(m)}$ that will span $\mathbb{R}^m$.

(2) Select the l^{th} artificial variable $y_l = 0$ and select a component j in the l^{th} row with nonzero $B^{-1}A_j$ and use elementary row operations to include A_j in the basis. Repeat this $m - k$ times.

The procedure is based on several ideas we have seen before. Since $\sum_{i=1}^{m} y_i$ is zero at the optimal, there must be a BFS in which the artificial variables are nonbasic (which is what (1) is referring to). Thus, step (2) can be repeated until a basis B is formed and includes none of the artificial variables.

Some interesting points are worth highlighting. First, notice that $B^{-1}A_{B(i)} = e_i$, $i = 1, \ldots, k$. Since $k < l$, the l^{th} component of each of these vectors is zero and will remain so after performing the elementary row operations. In turn, the l^{th} entry of $B^{-1}A_j$ is not zero, and thus A_j is linearly independent to $A_{B(1)}, \ldots, A_{B(k)}$.

However, it might be so that we find zero elements in the l^{th} row. Let g be the l^{th} row of $B^{-1}A$ (i.e., the entries in the tableau associated with the original problem variables). If g is the null vector, then g_l is zero, and the procedure fails. However, note that $g^\top A = 0 = g^\top Ax = g^\top b$, implying that $g^\top Ax = g^\top b$ is *redundant* can be removed altogether.

This process of generating initial BFS is often referred to as *Phase I* of the two-phase simplex method. *Phase II* consists of employing the simplex method as we developed it, utilising the BFS found in Phase I as a starting basis.

4.3 Column geometry of the simplex method

Let us try to develop a geometrical intuition on why it is so that the simplex method is remarkably efficient in practice. As we have seen in Theorem 4.2, although the simplex method is guaranteed to converge, the total number of steps the algorithm might need to take before convergence grows exponentially with the number of variables and constraints, since the number of steps depends on the number of vertices of the polyhedral set that represents the feasible region of the problem.

However, it turns out that, in practice, the simplex method typically requires $O(m)$ iterations (recall that m is the number of rows in the matrix A), being one of the reasons why it has experienced tremendous success and is one of the most mature and reliable methods when it comes to optimisation.

To develop a geometrical intuition on why this is the case, let us first consider an equivalently reformulated problem P:

$$\begin{aligned} P : \text{min. } & z \\ \text{s.t.: } & Ax = b \\ & c^\top x = z \\ & \sum_{j=1}^{n} x_j = 1 \\ & x \geq 0. \end{aligned}$$

In this reformulation, we make the objective function an auxiliary variable, so it can be easily represented on a real line at the expense of adding an additional constraint $c^\top x = z$. Furthermore, we normalise the decision variables so they add to one (notice that this implies a bounded feasible

set). Notice that problem P can be equivalently represented as

$$\begin{aligned} P: \min.\ & z \\ \text{s.t.:}\ & x_1 \begin{bmatrix} A_1 \\ c_1 \end{bmatrix} + x_2 \begin{bmatrix} A_2 \\ c_2 \end{bmatrix} + \cdots + x_n \begin{bmatrix} A_n \\ c_n \end{bmatrix} = \begin{bmatrix} b \\ z \end{bmatrix} \\ & \sum_{j=1}^{n} x_j = 1 \\ & x \geq 0. \end{aligned}$$

This second formulation exposes one interesting interpretation of the problem. Solving P is akin to finding a set of weights x that makes a convex combination (c.f. Definition 2.7) of the columns of A such that it constructs (or matches) b in a way that the resulting combination of the respective components of the vector c is minimised. Now, let us define some nomenclature that will be useful in what follows.

Definition 4.4 (k-dimensional simplex)**.** *A collection of vectors $y_1, \ldots, y_{k+1}$ are affinely independent if $k \leq n$ and $y_1 - y_{k+1}, \ldots, y_k - y_{k+1}$ are linearly independent. The convex hull of $k+1$ affinely independent vectors is a k-dimensional simplex.*

Definition 4.4 is precisely the inspiration for the name of the simplex method. We know that only $m+1$ components of x will be different than zero since that is the number of constraints we have and, thus, the size of a basis in this case. Thus, a BFS is formed by $m+1$ $(A_i, 1)$ columns, which in turn are associated with (A_i, c_i) basic points.

Figure 4.2 provides an illustration of the concept. In this, we have that $m = 2$, so each column A_j represents a point in a two-dimensional plane. Notice that a basis requires three points (A_i, c_i) and forms a 3-simplex. A BFS is a selection of three points (A_i, c_i) such that b, also illustrated in the picture, can be formed by a convex combination of the (A_i, c_i) forming the basis. This will be possible if b happens to be inside the 3-simplex formed by these points. For example, in Figure 4.2, the basis formed by columns $\{2, 3, 4\}$ is a BFS, while the basis $\{1, 2, 3\}$ is not.

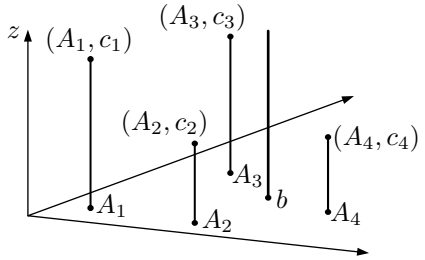


Figure 4.2: A solution x is a convex combinations of (A_i, c_i) such that $Ax = b$.

We now can add a third dimension to the analysis representing the value of z. For that, we will use Figure 4.3. As can be seen, each selection of basis creates a tilt in the three-dimensional simplex, such that the point b is met precisely at the height corresponding to its value in the z axis. This allows us to compare bases according to their objective function value. And, since we are minimising, we would like to find the basis that has its respective simplex crossing b at the lowermost point.

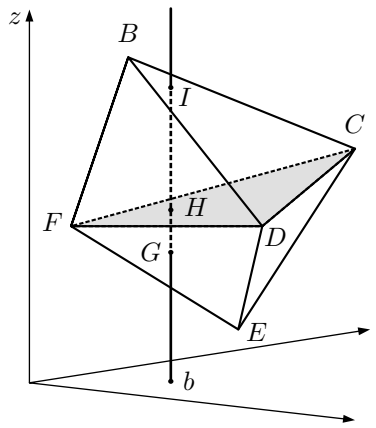


Figure 4.3: A solution x is a convex combinations of (A_i, c_i) such that $Ax = b$.

Notice that in Figure 4.3, although each facet is a basic simplex, only three are feasible (BCD, CDF, and DEF). We can also see what one iteration of the simplex method does under this geometrical interpretation. Moving between adjacent basis means that we are replacing one vertex (say, C) with another (say, E) considering the potential for a decrease in value in the z axis (represented by the difference between points H and G onto the z axis). You can also see the notion of pivoting: since we are moving between adjacent bases, two successive simplexes share an edge in common. Consequently, they pivot around that edge (think about the movement of the edge C moving to the point E while the edge $\overline{DF}$ remains fixed).

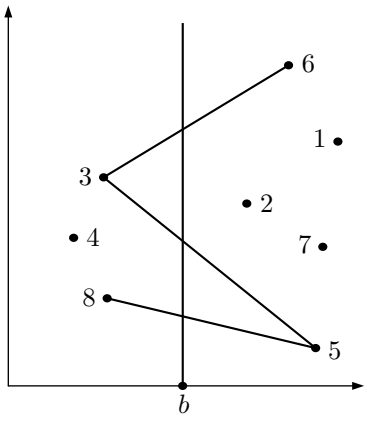


Figure 4.4: Pivots from initial basis $[A_3, A_6]$ to $[A_3, A_5]$ and to the optimal basis $[A_8, A_5]$

Now we are ready to provide an insight into why the simplex method is often so efficient. The main reason is associated with the ability that the method possesses of skipping bases in favour of those with most promising improvement. To see that, consider Figure 4.4, which is a 2-dimensional schematic projection of Figure 4.3. By using the reduced costs to guide the choice of the next basis, we tend to choose the steepest of the simplexes that can provide reductions in the objective function value, which has the side effect of allowing for skipping several bases that would have to be otherwise considered. This creates a "sweeping effect", which allows the method to find optimal

solutions in fewer pivots than vertices. Clearly, this can be engineered to be prevented, as there are examples purposely constructed to force the method to consider every single vertex, but the situation illustrated in Figure 4.4 is by far the most common in practice.

4.4 Exercises

Exercise 4.1: Properties of the simplex algorithms

Consider the simplex method applied to a standard form minimization problem, and assume that the rows of the matrix A are linearly independent. For each of the statements that follow, give either a proof or a counter example.

(a) An iteration of the simplex method might change the feasible solution while leaving the cost unchanged.

(b) A variable that has just entered the basis cannot leave in the very next iteration.

(c) If there is a non-degenerate optimal basis, then there exists a unique optimal basis.

Exercise 4.2: The simplex method

Consider the problem

$$\begin{aligned} \text{max. } & 40x_1 + 60x_2 \\ \text{s.t.: } & 2x_1 + x_2 \leq 7 \\ & x_1 + x_2 \leq 4 \\ & x_1 + 3x_2 \leq 9 \\ & x_1, x_2 \geq 0. \end{aligned}$$

A feasible point for this problem is $(x_1, x_2) = (0, 3)$. Formulate the problem as a minimisation problem in standard form and verify whether or not this point is optimal. If not, solve the problem by using the simplex method.

Exercise 4.3: Solving a tableau

Consider a linear programming problem in standard form, described in terms of the following tableau:

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	RHS
z	0	0	0	δ	3	γ	ξ	0
x_2	0	1	0	α	1	0	3	β
x_3	0	0	1	-2	2	η	-1	2
x_1	1	0	0	0	-1	2	1	3

The entries α, β, γ, δ, η and ξ in the tableau are unknown parameters. Furthermore, let B be the basis matrix corresponding to having x_2, x_3, and x_1 (in that order) be the basic variables. For each one of the following statements, find the ranges of values of the various parameters that will make the statement to be true.

(a) Phase II of the Simplex method can be applied using this as an initial tableau.

(b) The corresponding basic solution is feasible, but we do not have an optimal basis.

(c) The corresponding basic solution is feasible and the first Simplex iteration indicates that the optimal cost is $-\infty$.

(d) The corresponding basic solution is feasible, x_6 is a candidate for entering the basis, and when x_6 is the entering variable, x_3 leaves the basis.

(e) The corresponding basic solution is feasible, x_7 is a candidate for entering the basis, but if it does, the objective value remains unchanged.

Exercise 4.4: Two-phase simplex method

Solve the problem below using the two-phase simplex method. What is your conclusion about the feasibility of the problem? Verify your results by drawing the feasible region.

$$\begin{aligned} \text{max. } & 5x_1 + x_2 \\ \text{s.t.: } & 2x_1 + x_2 \geq 5 \\ & x_2 \geq 1 \\ & 2x_1 + 3x_2 \leq 12 \\ & x_1, x_2 \geq 0. \end{aligned}$$

Chapter 5

Linear Programming Duality - Part I

5.1 Linear programming duals

In this chapter, we will discuss the notion of duality in the context of linear programming problems. Duality can be understood as a toolbox of results that makes available a collection of techniques and features that can be exploited to both further understand characteristics of the optimal solution and to devise specialised methods for solving linear programming problems.

5.1.1 Motivation

Let us return to the paint factory example. Recall its formulation, given by

$$\begin{aligned}
\text{max. } z = & 5x_1 + 4x_2 && (5.1)\\
\text{s.t.: } & 6x_1 + 4x_2 \leq 24 && (5.2)\\
& x_1 + 2x_2 \leq 6 && (5.3)\\
& x_2 - x_1 \leq 1 && (5.4)\\
& x_2 \leq 2 && (5.5)\\
& x_1, x_2 \geq 0. && (5.6)
\end{aligned}$$

To motivate our developments, suppose we do not have an LP solver at our disposal but have found (perhaps by trial and error) a feasible solution $(3, 1)$ with an objective value of 19 and want to know how good of a solution this is. If we know nothing else about the problem, 19 might be a very bad solution compared to the optimal solution. On the other hand, if we can somehow determine an upper bound for the solution, showing that this problem simply cannot have a solution with an objective value better than, say, 25, we might consider the solution to be "good enough" even if we cannot show that it is optimal.

Notice that analysing the objective function (5.1) and the first resource constraint together (5.2) imply

$$z = 5x_1 + 4x_2 \leq 6x_1 + 4x_2 \leq 24.$$

The first inequality comes from the fact that the coefficients of the nonnegative decision variables x are greater or equal on the left-hand side. The first constraint (5.2) thus tells us that the optimal solution value cannot be more than 24, because of the availability of resources. This upper bound might be higher than the optimal value, but it allows us to conclude that our solution with a value of 19 cannot be improved by more than 26%, even if this theoretical upper bound could be reached. The second constraint (5.3) does not fulfil the requirement of coefficients being at least equal to

those in the objective function, but multiplying the constraint by 5 on both sides achieves just that. This gives us

$$z = 5x_1 + 4x_2 \leq 5x_1 + 10x_2 \leq 30,$$

which is a worse bound than 24.

Finally, we notice that two constraints can be combined to obtain a new constraint. Multiplying the first constraint by 0.75 and the second by 0.5, we can add the results together to obtain:

$$(0.75 \times 6 + 0.5 \times 1)x_1 + (0.75 \times 4 + 0.5 \times 2)x_2 \leq 0.75 \times 24 + 0.5 \times 6,$$

or

$$5x_1 + 4x_2 \leq 21.$$

Turns out, this upper bound 21 is also the optimal solution, though we cannot know it from this analysis only.

Let us formalise these ideas. What we want to obtain is a constraint giving us an upper bound for the original problem, hereinafter referred to as the primal problem P. This constraint is obtained as a weighted combination of the primal constraints. Denote the weights as p_i for each constraint $i \in [m]$. In the case of a primal problem of the form

$$\begin{aligned} (P): \text{min.} \quad & c^\top x \\ \text{s.t.:} \quad & Ax \leq b \\ & x \geq 0, \end{aligned}$$

the weights p result in a constraint:

$$\sum_{i=1}^{m} p_i \sum_{j=1}^{n} A_{ij} x_j \leq \sum_{i=1}^{m} p_i b_i,$$

where the coefficient of x_j is $\sum_{i=1}^{m} p_i A_{ij}$. For this to give an upper bound of the objective function $\sum_{j=1}^{n} c_j x_j$, the condition

$$\sum_{i=1}^{m} p_i A_{ij} \geq c_j$$

must be satisfied for each $j \in [n]$. In matrix form, this becomes:

$$A^\top p \geq c.$$

Note that in order for us to be able to combine the constraints into one, the weights p must be nonnegative, as negative weights would flip the inequalities in the corresponding primal constraints. Finally, we want the upper bound to be as tight as possible, so we minimise the right-hand side $b^\top p$ of this combined constraint, resulting in the dual problem:

$$\begin{aligned} (D): \text{min.} \quad & b^\top p \\ \text{s.t.:} \quad & A^\top p \geq c \\ & p \geq 0. \end{aligned}$$

Next, we formally devise the technical framework behind these "bounding" problems and derive the results regarding the equivalence between problems P and D.

5.1.2 Formulating duals

Let us define the notation we will be using throughout the next chapters. As before, let $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, $A \in \mathbb{R}^{m \times n}$, and P be the standard form linear programming problem

$$\begin{aligned}(P) : \text{min.} \quad & c^\top x \\ \text{s.t.:} \quad & Ax = b \\ & x \geq 0,\end{aligned}$$

which we will refer to as the *primal* problem. In mathematical programming, we say that a constraint has been *relaxed* if it has been removed from the set of constraints. With that in mind, let us consider a *relaxed* version of P, where $Ax = b$ is replaced with a *violation penalty* term $p^\top(b - Ax)$. This leads to the following problem:

$$g(p) = \min_{x \geq 0} \left\{ c^\top x + p^\top (b - Ax) \right\},$$

which has the benefit of not having equality constraints explicitly represented, but only implicit by means of a penalty term. This term is used to penalise the infeasibility of the constraints in the attempt to steer the solution of the relaxed problem towards the solution to P. Recalling that our main objective is to solve P, we are interested in the values (or prices, as they are often called) for $p \in \mathbb{R}^m$ that make P and $g(p)$ equivalent.

Let $\overline{x}$ be the optimal solution to P. Notice that, for any $p \in \mathbb{R}^m$, we have that

$$g(p) = \min_{x \geq 0} \left\{ c^\top x + p^\top (b - Ax) \right\} \leq c^\top \overline{x} + p^\top (b - A\overline{x}) = c^\top \overline{x},$$

i.e., $g(p)$ is a *lower bound* on the optimal value $c^\top \overline{x}$. The inequality holds because, although $\overline{x}$ is optimal for P, it might not be optimal for $g(p)$ for an arbitrary vector p. The rightmost equality is a consequence of $\overline{x} \in P$, i.e., the feasibility of $\overline{x}$ implies that $A\overline{x} = b$.

We can use an optimisation-based approach to try to find an optimal lower bound, i.e., the tightest possible lower bound for P. This can be achieved by solving the *dual problem* D formulated as

$$(D) : \max_{p} \; g(p).$$

Notice that D is an unconstrained problem to which a solution proves the *tightest* lower bound on P (say, at $\overline{p}$). Also, notice how the function $g(p) : \mathbb{R}^m \to \mathbb{R}$ has embedded on its evaluation the solution of a linear programming problem with $x \in \mathbb{R}^n$ as decision variables for a fixed p, which is the argument given to the function g.

We will proceed in this chapter developing the analytical framework that allows us to pose the key result in duality theory, which states that

$$g(\overline{p}) = c^\top \overline{x}.$$

That is, we will next develop the results that guarantee the equivalence between primal and dual representations. This will be useful for interpreting properties associated with the optimal primal solution $\overline{x}$ from the associated optimal prices $\overline{p}$. Furthermore, we will see in later chapters that linear programming duality can be used as a framework for replacing constraints with equivalent representations, which is a useful procedure in many settings, including developing alternative solution strategies also based on linear programming.

5.1.3 General form of duals

Now, let us focus on developing a formulation for dual problems that is based on linear programming as well. Using the definition of D, we notice that

$$\begin{aligned} g(p) &= \min_{x \geq 0} \left\{ c^\top x + p^\top (b - Ax) \right\} \\ &= p^\top b + \min_{x \geq 0} \left\{ c^\top x - p^\top Ax \right\} \\ &= p^\top b + \min_{x \geq 0} \left\{ (c^\top - p^\top A)x \right\}. \end{aligned}$$

As $x \geq 0$, the rightmost problem can only be bounded if $(c^\top - p^\top A) \geq 0$. This gives us a linear constraint that can be used to enforce the existence of a solution for

$$\min_{x \geq 0} \left\{ (c^\top - p^\top A)x \right\}.$$

With that in mind, we can equivalently reformulate D as

$$\begin{aligned} (D) : \text{max.} \quad & p^\top b \\ \text{s.t.:} \quad & p^\top A \leq c^\top. \end{aligned}$$

Notice that D is a linear programming problem with m variables (one per constraint of the primal problem P) and n constraints (one per variable of P). As you might suspect, if you were to repeat the analysis, looking at D as the "primal" problem, you would end with a dual that is exactly P. For this to become more apparent, let us first define more generally the rules that dictate what kind of dual formulations are obtained for different types of primal problems in terms of their original (i.e., not in standard) form.

In the more general case, let P be defined as

$$\begin{aligned} (P) : \text{min.} \quad & c^\top x \\ \text{s.t.:} \quad & Ax \geq b. \end{aligned}$$

Notice that the problem P can be equivalently reformulated as

$$\begin{aligned} (P) : \text{min.} \quad & c^\top x \\ \text{s.t.:} \quad & Ax - s = b \\ & s \geq 0. \end{aligned}$$

Let us focus on the constraints in the reformulated version of P, which can be written as

$$[A \mid -I] \begin{bmatrix} x \\ s \end{bmatrix} = b.$$

We will apply the same procedure as before, being our constraint matrix $[A \mid -I]$ in place of A and $[x \mid s]^\top$ our vector of variables, in place of x. Using analogous arguments, we now require that $c^\top - p^\top A = 0$, so $g(p)$ is finite. Notice that this is a slight deviation from before, but in this case, we have that $x \in \mathbb{R}^n$, so $c^\top - p^\top A = 0$ is the only condition that allows the inner problem in $g(p)$ to have a finite solution. Then, we obtain the following conditions to be imposed to our dual linear programming formulation

$$\begin{aligned} & p^\top [A \mid -I] \leq [c^\top \mid 0^\top] \\ & c^\top - p^\top A = 0, \end{aligned}$$

Combining them all and redoing the previous steps for obtaining a dual formulation, we arrive at

$$\begin{aligned}(D): \max.\ & p^\top b \\ \text{s.t.: } & p^\top A = c^\top \\ & p \geq 0.\end{aligned}$$

Notice how the change in the type of constraints in the primal problem P lead to additional nonnegative constraints in the dual variables p. Similarly, the absence of explicit nonnegativity constraints in the primal variables x lead to equality constraints in the dual problem D, as opposed to inequalities.

Table 5.1 provides a summary which allows one to identify the resulting formulation of the dual problem based on the primal formulation, in particular regarding its type (minimisation or maximisation), constraint types and variable domains.

Primal (dual)	**Dual (primal)**
minimise	maximise
Independent terms	Obj. function coef.
Obj. function coef.	Independent terms
i-th row of constraint coef.	i-th column of constraint coef.
i-th column of constraint coef.	i-th row of constraint coef.
Constraints	**Variables**
$\geq$	≥ 0
$\leq$	≤ 0
$=$	$\in \mathbb{R}$
Variables	**Constraints**
≥ 0	$\leq$
≤ 0	$\geq$
$\in \mathbb{R}$	$=$

Table 5.1: Primal-dual conversion table

For converting a minimisation primal problem into a (maximisation) dual, one must read the table from left to right. That is, the independent terms (b) become the objective function coefficients, greater or equal constraints become nonnegative variables, and so forth. However, if the primal problem is a maximisation problem, the table must be read from right to left. For example, in this case, less-or-equal-than constraints would become nonnegative variables instead, and so forth. It takes a little practice to familiarise yourself with this table, but it is a really useful resource to obtain dual formulations from primal problems.

One remark to be made at this point is that, as is hopefully clearer now, the conversion of primal problems into duals is symmetric, meaning that reapplying the rules in Table 5.1 would take you from the obtained dual back to the original primal. This is a property of linear programming problems called being *self dual*. Another remark is that equivalent reformulations made in the primal lead to equivalent duals. Specifically, transformations that replace variables $x \in \mathbb{R}$ with $x^+ - x^-$, where $x^+, x^- \geq 0$, introduce nonnegative slack variables, or remove redundant constraints all lead to equivalent duals.

For example, recall that the dual formulation for the primal problem

$$\begin{aligned}(P): \text{min.} \quad & c^\top x \\ \text{s.t.:} \quad & Ax \geq b \\ & x \in \mathbb{R}^n\end{aligned}$$

is given by

$$\begin{aligned}(D): \text{max.} \quad & p^\top b \\ \text{s.t.:} \quad & p \geq 0 \\ & p^\top A = c^\top.\end{aligned}$$

Now suppose we equivalently reformulate the primal problem to become

$$\begin{aligned}(P'): \text{min.} \quad & c^\top x + 0^\top s \\ \text{s.t.:} \quad & Ax - s = b \\ & x \in \mathbb{R}^n, s \geq 0.\end{aligned}$$

Then, using Table 5.1, we would obtain the following dual formulation, which is equivalent to D

$$\begin{aligned}(D'): \text{max.} \quad & p^\top b \\ \text{s.t.:} \quad & p \in \mathbb{R}^m \\ & p^\top A = c^\top \\ & -p \leq 0.\end{aligned}$$

Analogously, suppose we were to equivalently reformulate P as

$$\begin{aligned}(P''): \text{min.} \quad & c^\top x^+ - c^\top x^- \\ \text{s.t.:} \quad & Ax^+ - Ax^- \geq b \\ & x^+ \geq 0, x^- \geq 0.\end{aligned}$$

Then, the dual formulation for P'' would be

$$\begin{aligned}(D''): \text{max.} \quad & p^\top b \\ \text{s.t.:} \quad & p \geq 0 \\ & p^\top A \leq c^\top \\ & -p^\top A \leq -c^\top,\end{aligned}$$

which is also equivalent to D.

5.2 Duality theory

We will now develop the technical results associated with duality that will be the kingpin for its use as a framework for devising solution methods and interpreting optimal solution properties.

5.2.1 Weak duality

Weak duality is the property associated with the bounding nature of dual feasible solutions. This is stated in Theorem 5.1.

Theorem 5.1 (Weak duality)**.** *Let x be a feasible solution to (P) : min. $\{c^\top x : Ax = b, x \geq 0\}$ and p be a feasible solution to (D) : max. $\{p^\top b : p^\top A \leq c^\top\}$, the dual problem of P. Then $c^\top x \geq p^\top b$.*

Proof. Let $I = [m]$ and $J = [n]$. For any x and p, define

$$u_i = p_i(a_i^\top x - b_i) \text{ and } v_j = (c_j - p^\top A_j)x_j.$$

where a_i is the i^{th} row and A_j is the j^{th} column of A. Notice that $u_i \geq 0$ for $i \in I$ and $v_j \geq 0$ for $j \in J$, since each pair of terms will have the same sign (you can see that from Table 5.1 and assuming x_j to be the dual variable associated with $p^\top A \leq c^\top$). Thus, we have that

$$0 \leq \sum_{i \in I} u_i + \sum_{j \in J} v_j = [p^\top Ax - p^\top b] + [c^\top x - p^\top Ax] = c^\top x - p^\top b. \qquad \square$$

Let us also pose some results that are direct consequences of Theorem 5.1, which are summarised in Corollary 5.2.

Corollary 5.2 (Consequences of weak duality)**.** *The following are immediate consequences of Theorem 5.1:*

(1) If the optimal value of P is $-\infty$ (i.e., P is unbounded), then D must be infeasible;

(2) if the optimal value of D is ∞ (i.e., D is unbounded), then P must be infeasible;

(3) let x and p be feasible to P and D, respectively. Suppose that $p^\top b = c^\top x$. Then x is optimal to P and p is optimal to D.

Proof. By contradiction, suppose that P has optimal value $-\infty$ and that D has a feasible solution p. By weak duality, $p^\top b \leq c^\top x = -\infty$, i.e., a contradiction. Part (2) follows a symmetric argument.

Part (3): let $\overline{x}$ be an alternative feasible solution to P. From weak duality, we have $c^\top \overline{x} \geq p^\top b = c^\top x$, which proves the optimality of x. The optimality of p follows a symmetric argument. $\square$

Notice that Theorem 5.1 provides us with a bounding technique for any linear programming problem. That is, for a given pair of primal and dual feasible solutions, $\overline{x}$ and $\overline{p}$, respectively, we have that

$$\overline{p}^\top b \leq c^\top x^* \leq c^\top \overline{x},$$

where $c^\top x^*$ is the optimal objective function value.

Corollary 5.2 also provides an alternative way of identifying infeasibility by means of linear programming duals. One can always use the unboundedness of a given element of a primal-dual pair to state the infeasibility of the other element in the pair. That is, an unbounded dual (primal) implies an infeasible primal (dual). However, the reverse statement is not as conclusive. Specifically, an infeasible primal (dual) does not necessarily imply that the dual (primal) is unbounded, but does imply it to be *either* infeasible or unbounded.

5.2.2 Strong duality

This bounding property can also be used as a certificate of optimality, in case they match in value. This is precisely the notion of *strong duality*, a property that is inherent to linear programming problems. This is stated in Theorem 5.3.

Theorem 5.3 (Strong duality)**.** *If* $(P) : min.\ \left\{c^\top x : Ax = b, x \geq 0\right\}$ *has an optimal solution, so does its dual* $(D) : max.\ \left\{p^\top b : p^\top A \leq c^\top\right\}$ *and their respective optimal values are equal.*

Proof. Assume P is solved to optimality, with optimal solution x and basis B. Let $x_B = B^{-1}b$. At the optimal, the reduced costs are $c^\top - c_B^\top B^{-1}A \geq 0$. Let $p^\top = c_B^\top B^{-1}$. We then have $p^\top A \leq c^\top$, which shows that p is feasible to D. Moreover,

$$p^\top b = c_B^\top B^{-1} b = c_B^\top x_B = c^\top x, \tag{5.7}$$

which, in turn, implies the optimality of p (cf. Corollary 5.2 (3)). □

The proof of Theorem 5.3 reveals something remarkable relating the simplex method and the dual variables p. As can be seen in (5.7), for any primal feasible solution x, an associated dual (not necessarily) feasible solution p can be immediately recovered. If the associated dual solution is also feasible, then Theorem 5.3 guarantees that optimality ensues.

This means that we can interchangeably solve either a primal or a dual form of a given problem, considering aspects related to convenience and computational ease. This is particularly useful in the context of the simplex method since the prices p are readily available as the algorithm progresses. In the next section, we will discuss several practical uses of this relationship in more detail. For now, let us halt this discussion for a moment and consider a geometrical interpretation of duality in the context of linear programming.

5.3 Geometric interpretation of duality

Linear programming duality has an interesting geometric interpretation that stems from the much more general framework of Lagrangian duality (of which linear programming duality is a special case) and its connection to optimality conditions. For now, let us focus on how linear programming duality can be interpreted in the context of "balancing forces".

First, let $\overline{x}$ be the optimal solution of primal problem P in the form

$$\begin{aligned}(P) : \min.\ \ & c^\top x \\ \text{s.t.: } & a_i^\top x \geq b_i,\ \forall i \in I.\end{aligned}$$

Imagine that there is a particle within the polyhedral set representing the feasible region of P and that this particle is subjected to a force represented by the vector $-c$. Notice that this is equivalent to minimising the function $z = c^\top x$ within the polyhedral set $\left\{a_i^\top x \geq b_i\right\}_{i \in I}$ representing the feasible set of P. Assuming that the feasible set of P is bounded in the direction $-c$, this particle will eventually come to a halt after hitting the "walls" of the feasible set, at a point where the pulling force $-c$ and the reaction of these walls reach an equilibrium. We can think of $\overline{x}$ as this stopping point. This is illustrated in Figure 5.1.

We can then think of the dual variables p as the multipliers applied to the normal vectors associated with the hyperplanes (i.e., the walls) that are in contact with the particle to achieve this

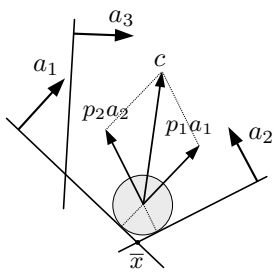


Figure 5.1: A geometric representation of duality for linear programming problems

equilibrium. Hence, these multipliers p will be such that

$$c = \sum_{i \in I} p_i a_i, \text{ for some } p_i \geq 0, i \in I,$$

which is precisely the dual feasibility condition (i.e., constraint) associated with the dual of P, given by

$$D : \max. \left\{ p^\top b : p^\top A = c, p \geq 0 \right\}.$$

And, dual feasibility, as we seen before, implies the optimality of $\overline{x}$.

5.3.1 Complementary slackness

One point that must be noticed is that, for the constraints that are not active at the optimal point $\overline{x}$ (i.e., the walls that are not exerting resistance to the particle at the equilibrium point), the multipliers p must be set to zero. That is, we have that

$$p^\top b = \sum_{i \in I} p_i b_i = \sum_{i \in I} p_i (a_i^\top \overline{x}) = c^\top \overline{x},$$

which again implies the optimality of p (cf. Corollary 5.2 (3)). This geometrical insight leads to another key result for linear programming duality, which is the notion of *complementary slackness*.

Theorem 5.4 (Complementary slackness)**.** *Let x be a feasible solution for*

$$(P) : \textit{min.} \left\{ c^\top x : Ax = b, x \geq 0 \right\}$$

and p be a feasible solution for

$$(D) : \textit{max.} \left\{ p^\top b : p^\top A \leq c^\top \right\}.$$

The vectors x and p are optimal solutions to P and D, respectively, if and only if $p_i(a_i^\top x - b_i) = 0, \forall i \in I$, and $(c_j - p^\top A_j)x_j = 0$, $\forall j \in J$.

Proof. From the proof of Theorem 5.1 and with Theorem 5.3 holding, we have that

$$p_i(a_i^\top x - b_i) = 0, \forall i \in I, \text{ and } (c_j - p^\top A_j)x_j = 0, \forall j \in J.$$

In turn, if these hold, then x and p are optimal (cf. Corollary 5.2 (3)). □

Theorem 5.4 exposes an important feature related to optimal solutions. Notice that $a_i^\top x - b_i$ represents the slack value of constraint $i \in I$. Thus, under the assumption of nondegeneracy, $p_i(a_i^\top x - b_i) = 0, \forall i \in I$, implies that for each constraint $i \in I$, either the dual variable p_i associated with constraint is zero, or the associated slack value $a_i^\top x - b_i$ is zero.

For nondegenerate basic feasible solutions (BFS) (i.e., $x_j > 0, \forall j \in I_B$, where I_B is the set of basic variable indices), complementary slackness determines a *unique* dual solution. That is

$$(c_j - p^\top A_j)x_j = 0, \text{ which yields } c_j = p^\top A_j, \forall j \in I_B,$$

which has a unique solution $p^\top = c_B^\top B^{-1}$, as the columns A_j of be are assumed to be linearly independent. In the presence of degeneracy, this is not the case anymore, typically implying that a degenerate optimal BFS will have multiple associated feasible dual variables.

5.3.2 Dual feasibility and optimality

Combining what we have seen so far, the conditions for a *primal-dual pair* (x, p) to be optimal to their respective primal (P) and dual (D) problems are given by

$$a_i^\top x \geq b_i, \forall i \in I \qquad \text{(primal feasibility)} \tag{5.8}$$

$$p_i = 0, \forall i \notin I^0 \qquad \text{(complementary conditions)} \tag{5.9}$$

$$\sum_{i \in I} p_i^\top a_i = c \qquad \text{(dual feasibility I)} \tag{5.10}$$

$$p_i \geq 0, \qquad \text{(dual feasibility II)} \tag{5.11}$$

where $I^0 = \left\{i \in I, a_i^\top x = b_i\right\}$ are the active constraints. From (5.8)–(5.11), we see that the optimality of the primal-dual pair has two main requirements. The first is that x must be (primal) feasible. The second, expressed as

$$\sum_{i \in I^0} p_i a_i = c, \; p_i \geq 0,$$

is equivalent to requiring c to be expressed as a nonnegative linear combination (also known as a conic combination) of the active constraints. This has a nice geometrical interpretation: dual feasibility can be interpreted as having the vector c inside the "cone" formed by the normal vectors of the active constraints, which in turn is a necessary condition for the existence of an equilibrium, as described in Figure 5.1. Figure 5.2 illustrates this fact.

Notice how in Figure 5.2 neither points A or B are dual feasible, while C represents a dual feasible point, being thus the optimal for the problem depicted. One interesting point to notice is D. Although not feasible, it allows us to see an important effect that degeneracy may cause. Assume for a moment that D is feasible. Then, dual feasibility becomes dependent on the basis representing the vertex. That is, while the bases $I_B = \{1, 5\}$ and $I_B = \{1, 6\}$ are dual feasible, the basis $I_B = \{5, 6\}$ is not. As we will see, just as it is the case with the simplex method, the dual simplex method, which we will discuss in the next section, might be subject to stalling and cycling from the presence of primal degeneracy, which in turn may also leads to multiple dual feasible (primal optimal) solutions.

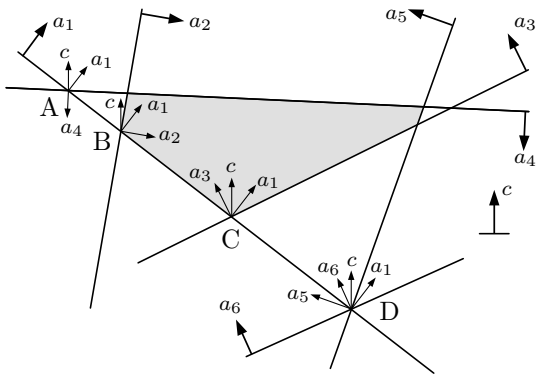


Figure 5.2: A is both primal and dual infeasible; B is primal feasible and dual infeasible; C is primal and dual feasible; D is degenerate.

5.4 The dual simplex method

In general, solution methods in mathematical programming can be either *primal methods*, in which primal feasibility of an initial solution is maintained while seeking for dual feasibility (i.e., primal optimality); or *dual methods*, where dual feasibility is maintained while seeking for primal feasibility (i.e., dual optimality).

As we have seen in Chapter 4, the original (or primal) simplex method iterated from an initial basic feasible solution (BFS) until the optimality condition

$$\overline{c} = c^\top - c_B B^{-1} A \geq 0$$

was observed. Notice that this is precisely the dual feasibility condition $p^\top A \leq c^\top$.

Being a dual method, the dual version of the simplex method, or the *dual simplex method*, considers conditions in reverse order. That is, it starts from an initial dual feasible solution and iterates in a manner that the primal feasibility condition $B^{-1}b \geq 0$ is *sought* for, while $\overline{c} \geq 0$, or equivalently, $p^\top A \leq c$, is maintained.

To achieve that, one must revise the pivoting of the primal simplex method such that the variable to leave the basis is some $i \in I_B$, with $x_{B(i)} < 0$, while the variable chosen to enter the basis is some $j \in I_N$, such that $\overline{c} \geq 0$ is maintained.

Consider the l^{th} simplex tableau row for which $x_{B(i)} < 0$ of the form $[v_1, \ldots, v_n, x_{B(i)}]$; i.e., v_j is the l^{th} component of $B^{-1}A_j$.

For each $j \in I_N$ for which $v_j < 0$, we pick

$$j' = \arg\min_{j \in I_N : v_j < 0} \frac{\overline{c}_j}{|v_j|}.$$

Pivoting is performed by employing elemental row operations to replace $A_{B(i)}$ with A_j in the basis. This implies that $\overline{c}_j \geq 0$ is maintained, since

$$\frac{\overline{c}_j}{|v_j|} \geq \frac{\overline{c}_{j'}}{|v_{j'}|} \Rightarrow \overline{c}_j - |v_j|\frac{\overline{c}_{j'}}{|v_{j'}|} \geq 0 \Rightarrow \overline{c}_j + v_j\frac{\overline{c}_{j'}}{|v_{j'}|} \geq 0, \, \forall j \in J.$$

Notice that it also justifies why we must only consider for entering the basis those variables for which $v_j < 0$. Analogously to the case in the primal simplex method, if we observe that $v_j \geq 0$

for all $j \in J$, then no limiting condition is imposed in terms the increase in the nonbasic variable (i.e., an unbounded dual, which, according to Corollary 5.2 (2), implies the original problem is infeasible).

Assuming that the dual is not unbounded, the termination of the dual simplex method is observed when $B^{-1}b \geq 0$ is achieved, and primal-dual optimal solutions have been found, with $x = (x_B, x_N) = (B^{-1}b, 0)$ (i.e., the primal solution) and $p = (p_B, p_N) = (0, c_B^\top B^{-1})$ (dual). Algorithm 3 presents a pseudocode for the dual simplex method.

Algorithm 3 Dual simplex method

initialise. Initial basis B and associated basic solution x.
while $x_B = B^{-1}b < 0$ for some component $i \in I_B$ **do**
    Choose some l for which $x_{B(l)} < 0$. Make v the the l^{th} row of $B^{-1}A$.
    if $u \geq 0$ **then**
        return $z = +\infty$.
    else
        Form new basis $B = B \setminus \{l\} \cup \{j'\}$ where $j' = \arg\min_{j \in I_N : v_j < 0} \frac{\overline{c}_j}{|v_j|}$
        Calculate $x_B = B^{-1}b$.
    end if
end while
return optimal basis B and optimal solution x.

To clarify some of the previous points, let us consider a numerical example. Consider the problem

$$\begin{aligned} \text{min. } & x_1 + x_2 \\ \text{s.t.: } & x_1 + 2x_2 \geq 2 \\ & x_1 \geq 1 \\ & x_1, x_2 \geq 0. \end{aligned}$$

The first thing we must do is convert the greater-or-equal-than inequalities into less-or-equal-than inequalities and add the respective slack variables. This allows us to avoid the inclusion of artificial variables, which are not required anymore since we can allow for primal infeasibility. This leads to the equivalent standard form problem

$$\begin{aligned} \text{min. } & x_1 + x_2 \\ \text{s.t.: } & -x_1 - 2x_2 + x_3 = -2 \\ & -x_1 + x_4 = -1 \\ & x_1, x_2, x_3, x_4 \geq 0. \end{aligned}$$

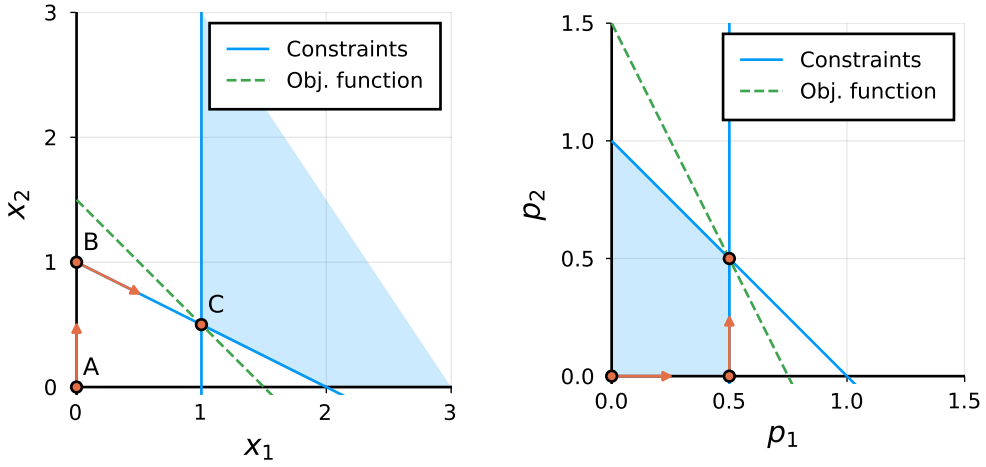
Below is the sequence of tableaus after applying the dual simplex method to solve the problem. The terms in bold font represent the pivot element (i.e., the intersection between the pivot row and pivot column).

	x_1	x_2	x_3	x_4	RHS
z	1	1	0	0	0
x_3	-1	**-2**	1	0	-2
x_4	-1	0	0	1	-1

	x_1	x_2	x_3	x_4	RHS
z	1/2	0	1/2	0	-1
x_2	1/2	1	-1/2	0	1
x_4	**-1**	0	0	1	-1

	x_1	x_2	x_3	x_4	RHS
z	0	0	1/2	1/2	-3/2
x_2	0	1	-1/2	1/2	1/2
x_1	1	0	0	-1	1

Figure 5.3 illustrates the progress of the algorithm both in the primal (Figure 5.3a) and in the dual (Figure 5.3b) variable space. Notice how in the primal space the solution remains primal infeasible until a primal feasible solution is reached, that being the optimal for the problem. Also, notice that the coordinates of the dual variables can be extracted from the zeroth row of the simplex tableau.



(a) The primal-variable space

(b) The dual-variable space

Figure 5.3: The progress of the dual simplex method in the primal and dual space.

Some interesting features related to the progress of the dual simplex algorithm are worth highlighting. First, notice that the objective function is monotonically increasing in this case, since $x_{B(l)} \frac{\bar{c}_{j'}}{|v_{j'}|}$ is added to $-c_B B^{-1} b$ and $x_{B(l)} < 0$, meaning that the dual cost increases (recall the convention of having a minus sign so that the zeroth row correctly represent the objective function value, given by the negative of the value displayed in the rightmost column). This illustrates how the solution becomes gradually worse as it compromises optimality in the search for (primal) feasibility. For a nondegenerate problem, this can also be used as an argument for eventual convergence since the dual objective value can only increase and is bounded by the primal optimal value. However, in the presence of dual degeneracy, that is, $\bar{c}_j = 0$ for some $j \in I_N$ in the optimal solution, the algorithm can suffer from cycling. As we have seen before, that is an indication that the primal problem has multiple optima.

The dual simplex method is often the best choice of algorithm, because it typically precludes the need for a Phase I type of method as it is often trivial to find initial dual feasible solutions

(the origin, for example, is typically dual feasible in minimisation problems with nonnegative coefficients; similar trivial cases are also well known).

Moreover, dual simplex is the algorithm of choice for resolving a linear programming problem when after finding an optimal solution, you modify the feasible region. Turns out that this procedure is in the core of the methods used to solve integer programming problems, as well as in the Benders decomposition, both topics we will explore later on. The dual simplex method is also more successful than its primal counterpart in combinatorial optimisation problems, which are typically plagued with degeneracy. As we have seen, primal degeneracy simply means multiple dual optima, which are far less problematic under an algorithmic standpoint.

Most professional implementations of the simplex method use by default the dual simplex version. This has several computational reasons, in particular related to more effective Phase I and pricing methods for the dual counterpart.

5.5 Exercises

Exercise 5.1: Duality in the transportation problem

Recall the transportation problem from Chapter 1. Answer the following questions based on the interpretation of the dual price.

We would like to plan the production and distribution of a certain product, taking into account that the transportation cost is known (e.g., proportional to the distance travelled), the factories (or source nodes) have a supply capacity limit, and the clients (or demand nodes) have known demands. Table 5.2 presents the data related to the problem.

	Clients			
Factory	NY	Chicago	Miami	Capacity
Seattle	2.5	1.7	1.8	350
San Diego	3.5	1.8	1.4	600
Demands	325	300	275	-

Table 5.2: Problem data: unit transportation costs, demands and capacities

Additionally, we consider that the arcs (routes) from factories to clients have a maximum transportation capacity, assumed to be 250 units for each arc. The problem formulation is then

$$\begin{aligned}
\text{min. } z = & \sum_{i \in I} \sum_{j \in J} c_{ij} x_{ij} \\
\text{s.t.: } & \sum_{j \in J} x_{ij} \leq C_i, \ \forall i \in I \\
& \sum_{i \in I} x_{ij} \geq D_j, \ \forall j \in J \\
& x_{ij} \leq A_{ij}, \ \forall i \in I, j \in J \\
& x_{ij} \geq 0, \forall i \in I, j \in J,
\end{aligned}$$

where C_i is the supply capacity of factory i, D_j is the demand of client j and A_{ij} is the transportation capacity of the arc between i and j.

(a) What price would the company be willing to pay for increasing the supply capacity of a given factory?

(b) What price would the company be willing to pay for increasing the transportation capacity of a given arc?

Exercise 5.2: Dual simplex

(a) Solve the problem below by using the dual simplex method. Report both the primal and dual optimal solutions x and p associated with the optimal basis.

(b) Write the dual formulation of the problem and use strong duality to verify that x and p are optimal.

$$
\begin{aligned}
\text{min. } & 2x_1 + x_3 \\
\text{s.t.: } & -1/4x_1 - 1/2x_2 \leq -3/4 \\
& 8x_1 + 12x_2 \leq 20 \\
& x_1 + 1/2x_2 - x_3 \leq -1/2 \\
& 9x_1 + 3x_2 \geq -6 \\
& x_1, x_2, x_3 \geq 0.
\end{aligned}
$$

Exercise 5.3: Unboundedness and duality

Consider the standard-form linear programming problem:

$$
\begin{aligned}
(P): \text{min. } & c^\top x \\
\text{s.t.: } & Ax = b \\
& x \geq 0,
\end{aligned}
$$

where $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$. Show that if P has a finite optimal solution, then the new problem $\overline{P}$ obtained from P by replacing the right-hand side vector b with another one $\bar{b} \in \mathbb{R}^m$ cannot be unbounded no matter what value the components of $\bar{b}$ can take.

Exercise 5.4: Dual in matrix form

Consider the linear programming problem:

$$
\begin{aligned}
(P): \text{min. } & {c^1}^\top x^1 + {c^2}^\top x^2 + {c^3}^\top x^3 \\
\text{s.t.} & \\
& A^1x^1 + A^2x^2 + A^3x^3 \leq b^1 \quad (y^1) \\
& A^4x^1 + A^5x^2 + A^6x^3 \leq b^2 \quad (y^2) \\
& A^7x^1 + A^8x^2 + A^9x^3 \leq b^3 \quad (y^3) \\
& x^1 \leq 0 \\
& x^2 \geq 0 \\
& x^3 \in \mathbb{R}^{|x^3|},
\end{aligned}
$$

where $A^{1,\ldots,9}$ are matrices, $b^{1,\ldots,3}$, $c^{1,\ldots,3}$ are column vectors, and $y^{1,\ldots,3}$ are the dual variables associated to each constraint.

(a) Write the dual problem in matrix form.

(b) Compute the dual optimum for the case in which

$A^1 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$; $A^2 = \begin{bmatrix} 5 & 1 \\ 0 & 0 \end{bmatrix}$; $A^3 = \begin{bmatrix} 6 \\ 0 \end{bmatrix}$; $A_4 = \begin{bmatrix} 1 & 1 \end{bmatrix}$; $A^5 = \begin{bmatrix} 0 & 1 \end{bmatrix}$; $A^6 = \begin{bmatrix} 1 \end{bmatrix}$;
$A^7 = \begin{bmatrix} 0 & 2 \end{bmatrix}$; $A^8 = \begin{bmatrix} 0 & 0 \end{bmatrix}$; $A^9 = \begin{bmatrix} 3 \end{bmatrix}$; $c^1 = \begin{bmatrix} 3 \\ 9 \end{bmatrix}$; $c^2 = \begin{bmatrix} 4 \\ 2 \end{bmatrix}$; $c^3 = \begin{bmatrix} 1 \end{bmatrix}$; $b^1 = \begin{bmatrix} 5 \\ 10 \end{bmatrix}$;
$b^2 = \begin{bmatrix} 3 \end{bmatrix}$; $b^3 = \begin{bmatrix} 6 \end{bmatrix}$.

Exercise 5.5: Primal-dual conversion and complementary slackness

Recall the transportation problem in Exercise 5.1.

(a) Construct the dual of the problem and solve both the original problem and its dual.

(b) Use complementary slackness to verify that the primal and dual solutions are optimal.

Chapter 6

Linear Programming Duality - Part II

The most direct application of duality in linear programming problems is the interpretation of dual variable values as marginal values associated with constraints, with important economic implications.

We will also consider the notion of solution stability and restarting the simplex method, once new variables or constraints are added to the problem post optimality. This will have an important consequence for the development of efficient solution methods for integer programming problems, which we will discuss in detail in later chapters.

6.1 Sensitivity analysis

We are interested in analysing aspects associated with the *stability* of the optimal solution $\overline{x}$ in terms of how it changes with the inclusion of new decision variables and constraints or with changes in the input data. Both cases are somewhat motivated by the realisation that problems typically emerge from dynamic settings. Thus, one must assess how *stable* a given plan (represented by $\overline{x}$) is or how it can be adapted in the face of changes in the original problem setting. This kind of analysis is generally referred to as *sensitivity analysis* in the context of linear programming.

First, we will consider the inclusion of new variables or new constraints *after* the optimal solution $\overline{x}$ is obtained. This setting represents, for example, the inclusion of a new product or a new production plant (referring to the context of resource allocation and transportation problems, as discussed in Chapter 1) or the consideration of additional constraints imposing new (or previously disregarded) requirements or conditions. The techniques we consider here will also be relevant in the following chapters. We will then discuss specialised methods for large-scale problems and solution techniques for integer programming problems, both topics that heavily rely on the idea of iteratively incrementing linear programming problems with additional constraints (or variables).

The second group of cases relates to changes in the input data. When utilising linear programming models to optimise systems performance, one must bear in mind that there is inherent uncertainty associated with the input data. Be it due to measurement errors or a lack of complete knowledge about the future, one must accept that the input data of these models will, by definition, embed some measure of error. One way of taking this into account is to try to understand the consequences to the optimality of $\overline{x}$ in case of eventual changes in the input data, represented by the matrix A, and the vectors c and b. We will achieve this by studying the ranges within which variations in these terms do not compromise the optimality of $\overline{x}$.

6.1.1 Adding a new variable

Assume that we have solved to optimality the problem P given as

$$\begin{aligned} P: \text{min. } & c^\top x \\ \text{s.t.: } & Ax = b \\ & x \geq 0. \end{aligned}$$

Let us consider that a new variable x_{n+1} with associated column (that is, respective objective function and constraint coefficients) (c_{n+1}, A_{n+1}) is added to P. This leads to a new augmented problem P' of the form

$$\begin{aligned} P': \text{min. } & c^\top x + c_{n+1}x_{n+1} \\ \text{s.t.: } & Ax + A_{n+1}x_{n+1} = b \\ & x \geq 0, x_{n+1} \geq 0. \end{aligned}$$

We need to determine if, after the inclusion of this new variable, the current basis B is still optimal. Making the newly added variable nonbasic yields the basic feasible solution (BFS) $x = (\overline{x}, 0)$. Moreover, we know that the optimality condition $c^\top - c_B^\top B^{-1} b \geq 0$ held before the inclusion of the variable, so we know that all the other reduced costs associated with the nonbasic variables $j \in I_N$ were nonnegative.

Therefore, the only check that needs to be done is whether the reduced cost associated with x_{n+1} also satisfies the optimality condition, i.e., if

$$\overline{c}_{n+1} = c_{n+1} - c_B^\top B^{-1} A_{n+1} \geq 0.$$

If the optimality condition is satisfied, the new variable does not change the optimal basis, and the solution $x = (\overline{x}, 0)$ is optimal. Otherwise, one must perform a new simplex iteration, using B as a starting BFS. Notice that in this case, primal feasibility is trivially satisfied, while dual feasibility is not observed (that is, $\overline{c}_{n+1} < 0$). Therefore, primal simplex can be employed, *warm started* by B. This is often a far more efficient strategy than resolving P' from scratch.

Let us consider a numerical example. Consider the problem

$$\begin{aligned} \text{min. } & -5x_1 - x_2 + 12x_3 \\ \text{s.t.: } & 3x_1 + 2x_2 + x_3 = 10 \\ & 5x_1 + 3x_2 + x_4 = 16 \\ & x_1, \ldots, x_4 \geq 0. \end{aligned}$$

The tableau associated with its optimal solution is given by

	x_1	x_2	x_3	x_4	RHS
z	0	0	2	7	12
x_1	1	0	-3	2	2
x_2	0	1	5	-3	2

Suppose we include a variable x_5, for which $c_5 = -1$ and $A_5 = (1, 1)$. The modified problem then

becomes

$$\begin{aligned} \text{min. } & -5x_1 - x_2 + 12x_3 - x_5 \\ \text{s.t.: } & 3x_1 + 2x_2 + x_3 + x_5 = 10 \\ & 5x_1 + 3x_2 + x_4 + x_5 = 16 \\ & x_1, \ldots, x_5 \geq 0. \end{aligned}$$

We have that the reduced cost of the new variable is given by $\bar{c}_5 = c_5 - c_B^\top B^{-1} A_5 = -4$ and $B^{-1}A_5 = (-1, 2)$. The tableau for the optimal basis B considering the new column associated with x_5 is thus

	x_1	x_2	x_3	x_4	x_5	RHS
z	0	0	2	7	-4	12
x_1	1	0	-3	2	-1	2
x_2	0	1	5	-3	2	2

Notice that this tableau now shows a primal feasible solution that is not optimal and can be further iterated using primal simplex.

6.1.2 Adding a new constraint

We now focus on the inclusion of additional constraints. Let us assume that a general constraint of the form ${a_{m+1}}^\top x \geq b_{m+1}$ is added to P after it has been solved. We assume it to be an inequality, but notice that P was originally in the standard form.

The first thing to observe is that, if the optimal solution $\overline{x}$ to P satisfy ${a_{m+1}}^\top \overline{x} \geq b_{m+1}$, then nothing changes. Otherwise, we need to rewrite the new constraint accordingly by including a slack variable, obtaining

$${a_{m+1}}^\top x - x_{n+1} = b_{m+1}.$$

Notice that doing so changes the matrix A of the original problem P, which becomes

$$\overline{A} = \begin{bmatrix} A & 0 \\ a_{m+1}^\top & -1 \end{bmatrix}.$$

We can reuse the optimal basis B to form a new basis $\overline{B}$ for the problem. This will have the form

$$\overline{B} = \begin{bmatrix} B & 0 \\ a^\top & -1 \end{bmatrix},$$

where a are the respective components of a_{m+1} associated with the columns from A that formed B. Now, since we have that ${\overline{B}}^{-1}\overline{B} = I$, we must have that

$$\overline{B}^{-1} = \begin{bmatrix} B^{-1} & 0 \\ a^\top B^{-1} & -1 \end{bmatrix}.$$

Notice however, that the basic solution $(\overline{x}, {a_{m+1}}^\top \overline{x} - b_{m+1})$ associated with $\overline{B}$ is not feasible, since we assumed that $\overline{x}$ did not satisfy the newly added constraint, i.e., ${a_{m+1}}^\top \overline{x} < b_{m+1}$.

The reduced costs considering the new basis $\overline{B}$ then become

$$[c^\top \;\; 0] - [c_B^\top \;\; 0] \begin{bmatrix} B^{-1} & 0 \\ a^\top B^{-1} & -1 \end{bmatrix} \begin{bmatrix} A & 0 \\ a_{m+1}^\top & -1 \end{bmatrix} = [c^\top - c_B^\top B^{-1} A \;\; 0].$$

Notice that the new slack variable has a null component as a reduced cost, meaning that it does not violate dual feasibility conditions. Thus, after adding a constraint that makes $\overline{x}$ infeasible, we still have a dual feasible solution that can be immediately used by the dual simplex method, again allowing for warm starting the solution of the new problem.

To build an initial solution in terms of the tableau representation of the simplex method, we must simply add an extra new row, which leads to a new tableau with the following structure

$$\overline{B}^{-1} A = \begin{bmatrix} B^{-1}A & 0 \\ a^\top B^{-1} A - a_{m+1}^\top & 1 \end{bmatrix}.$$

Let us consider a numerical example again. Consider the same problem as the previous example, but we instead include the additional constraint $x_1 + x_2 \geq 5$, which is violated by the optimal solution $(2, 2, 0, 0)$. In this case, we have that $a_{m+1} = (1, 1, 0, 0)$ and $a^\top B^{-1} A - a_{m+1}^\top = [0, 0, 2, -1]$. This modified problem then looks like

$$\begin{aligned} \text{min.} \;& -5x_1 - x_2 + 12x_3 \\ \text{s.t.:} \;& 3x_1 + 2x_2 + x_3 = 10 \\ & 5x_1 + 3x_2 + x_4 = 16 \\ & -x_1 - x_2 + x_5 = -5 \\ & x_1, \ldots, x_5 \geq 0 \end{aligned}$$

with associated tableau

	x_1	x_2	x_3	x_4	x_5	RHS
z	0	0	2	7	0	12
x_1	1	0	-3	2	0	2
x_2	0	1	5	-3	0	2
x_5	0	0	2	-1	1	-1

Notice that this tableau indicates that we have a dual feasible solution that is not primal feasible and thus suitable to be solved using dual simplex.

A final point to note is that these operations are related to each other in terms of equivalent primal-dual formulations. That is, consider dual of P, which is given by

$$\begin{aligned} D : \text{max.} \;& p^\top b \\ \text{s.t.:} \;& p^\top A \leq c. \end{aligned}$$

Then, adding a constraint of the form $p^\top A_{n+1} \leq c_{n+1}$ is equivalent to adding a variable to P, exactly as discussed in Section 6.1.1.

6.1.3 Changing input data

We now consider how changes in the input data can influence the optimality of a given basis. Specifically, we consider how to predict whether changes in the vector of independent terms b and objective coefficients c will affect the optimality of the problem. Notice that variations in the coefficient matrix A are left aside.

Optimal dual variables as marginal costs

As before, assume that we have solved P to optimality. As we have seen in Chapter 4, the optimal solution $\overline{x}$ with associated basis B satisfies the following optimality conditions: it is a BFS and, therefore (i) $B^{-1}b \geq 0$; and (ii) all reduced costs are nonnegative, i.e., $c^\top - c_B^\top B^{-1}b \geq 0$.

Now, assume that we cause a marginal perturbation on the vector b, represented by a vector d. That is, assume that we have $B^{-1}(b+d) > 0$, assuming that nondegeneracy is retained.

Recall that the optimality condition $\overline{c} = c^\top - c_B^\top B^{-1}A \geq 0$ is not influenced by such a marginal perturbation. In other words, for a small change d, the optimal basis (i.e., the selection of basic variables) is not disturbed. On the other hand, the optimal value of the basic variables, and consequently, the optimal value, becomes

$$c_B^\top B^{-1}(b+d) = p^\top(b+d).$$

Notice that $p^\top = c_B^\top B^{-1}$ is optimal for the (respective) dual problem. Thus, a change d causes a change of $p^\top d$ in the optimal value, meaning that the components p_i represent a *marginal value/cost* associated with the independent term b_i, for $i \in I$.

This has important implications in practice, as it allows for *pricing* the values of the resources associated with constraints. For example, suppose the dual value (or price) p_i is associated with a resource whose requirement is given by b_i. In that case, any opportunity to remove units of b_i for less than p_i should be seized since it costs p_i to satisfy any additional unit in the requirement b_i. A similar interpretation can be made in the context of less-or-equal-than constraints, in which p_i would indicate benefits (or losses, if $p_i > 0$) in increasing the availability of b_i.

For a numerical example, let us consider our paint factory problem once again.

The following tableau represents the optimal solution

	x_1	x_2	x_3	x_4	x_5	x_6	RHS
z	0	0	3/4	1/2	0	0	21
x_1	1	0	1/4	-1/2	0	0	3
x_2	0	1	-1/8	3/4	0	0	3/2
x_5	0	0	3/8	-5/4	1	0	5/2
x_6	0	0	1/8	-3/4	0	1	1/2

where x_3 and x_4 were the slack variables associated with raw material M1 and M2, respectively. In this case, we have that

$$B^{-1} = \begin{bmatrix} 1/4 & -1/2 & 0 & 0 \\ -1/8 & 3/4 & 0 & 0 \\ 3/8 & -5/4 & 1 & 0 \\ 1/8 & -3/4 & 0 & 1 \end{bmatrix} \text{ and } p = c_B^\top B^{-1} = \begin{bmatrix} -5 \\ -4 \\ 0 \\ 0 \end{bmatrix}^\top \begin{bmatrix} 1/4 & -1/2 & 0 & 0 \\ -1/8 & 3/4 & 0 & 0 \\ 3/8 & -5/4 & 1 & 0 \\ 1/8 & -3/4 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -3/4 \\ -1/2 \\ 0 \\ 0 \end{bmatrix}$$

Notice that these are values of the entries in the z-row below the slack variables $x_3, \ldots, x_6$, except the minus sign. This is because the z-rows contain the entries for $c - p^\top B^{-1}$ and we have an objective function coefficient of zero for all slack variables, i.e., $c_j = 0$, for $j = 3, \ldots, 6$. Also, recall that the paint factory problem is a maximisation problem, so p represents the decrease in the objective function value. In this, we see that removing one unit of M1 would decrease the objective function by 3/4 and removing one unit of M2 would similarly decrease the objective value by 1/2. Analogous, increasing M1 or M2 availability by one unit would increase the objective function value by 3/4 and 1/2, respectively.

Changes in the vector b

Suppose that some component b_i changes and becomes $b_i + \delta$, with $\delta \in \mathbb{R}$. We are interested in the range for δ within which the basis B remains optimal.

First, we must notice that optimality conditions $\bar{c} = c^\top - c_B^\top B^{-1} A \geq 0$ are not directly affected by variation in the vector b. This means that the choice of variable indices $j \in I_B$ to form the basis B will, in principle, be *stable* unless the change in b_i is such that B is rendered *infeasible*. Thus, we need to study the conditions in which feasibility is retained, or, more specifically, if (recall the e_i is the vector of zeros except for the i^{th} component being 1)

$$B^{-1}(b + \delta e_i) \geq 0.$$

Let $g = (g_{1i}, \ldots, g_{mi})$ be the i^{th} column of B^{-1}. Thus

$$B^{-1}(b + \delta e_i) \geq 0 \Rightarrow x_B + \delta g \geq 0 \Rightarrow x_{B(j)} + \delta g_{ji} \geq 0,\ j = 1, \ldots, m.$$

Notice that this is equivalent to having δ within the range

$$\max_{j: g_{ji} > 0} \left(-\frac{x_{B(j)}}{g_{ji}} \right) \leq \delta \leq \min_{j: g_{ji} < 0} \left(-\frac{x_{B(j)}}{g_{ji}} \right).$$

In other words, changing b_i will incur changes in the value of the basic variables, and thus, we must determine the range within which all basic variables remain nonnegative (i.e., feasible).

Let us consider a numerical example. Once again, consider the problem from Section 6.1.1. The optimal tableau was given by

	x_1	x_2	x_3	x_4	RHS
z	0	0	2	7	12
x_1	1	0	-3	2	2
x_2	0	1	5	-3	2

Suppose that b_1 will change by δ in the constraint $3x_1 + 2x_2 + x_3 = 10$. Notice that the first column of B^{-1} can be directly extracted from the optimal tableau and is given by $(-3, 5)$. The optimal basis will remain feasible if $2 - 3\delta \geq 0$ and $2 + 5\delta \geq 0$, and thus $-2/5 \leq \delta \leq 2/3$.

Notice that this means that we can calculate the change in the objective function value as a function of $\delta \in [-2/5, 2/3]$. Within this range, the optimal cost changes as

$$c_B^\top (b + \delta e_i) = p^\top b + \delta p_i,$$

where $p^\top = c_B^\top B^{-1}$ is the optimal dual solution. In case the variation falls outside that range, this means that some of the basic variables will become negative. However, since the dual feasibility conditions are not affected by changes in b_i, one can still reutilise the basis B using dual simplex to find a new optimal solution.

Changes in the vector c

We now consider the case where variations are expected in the objective function coefficients. Suppose that some component c_j becomes $c_j + \delta$. In this case, optimality conditions become a concern. Two scenarios can occur. First, it might be that the changing coefficient is associated

with a variable $j \in J$ that happens to be nonbasic ($j \in I_N$) in the optimal solution. In this case, we have that optimality will be retained as long as the nonbasic variable remains "not attractive", i.e., the reduced cost associated with j remains nonnegative. More precisely put, the basis B will remain optimal if

$$(c_j + \delta) - c_B B^{-1} A_j \geq 0 \Rightarrow \delta \geq -\overline{c}_j.$$

The second scenario concerns changes in variables that are basic in the optimal solution, i.e., $j \in I_B$. In that case, the optimality conditions are directly affected, meaning that we have to analyse the range of variation for δ within which the optimality conditions are maintained, i.e., the reduced costs remain nonnegative.

Let c_j is the coefficient of the l^{th} basic variable, that is $j = B(l)$. In this case, c_B becomes $c_B + \delta e_l$, meaning that all optimality conditions are simultaneously affected. Thus, we have to define a range for δ in which the condition

$$(c_B + \delta e_l)^\top B^{-1} A_i \leq c_i, \forall i \neq j$$

holds. Notice that we do not need to consider j since x_j is a basic variable, and thus, its reduced costs are assumed to remain zero.

Considering the tableau representation, we can use the l^{th} row and examine the conditions for which $\delta q_{li} \leq \overline{c}_i, \forall i \neq j$, where q_{li} is the i^{th} entry in the l^{th} row of $B^{-1}A$.

Let us once again consider the previous example, with optimal tableau

	x_1	x_2	x_3	x_4	RHS
z	0	0	2	7	12
x_1	1	0	-3	2	2
x_2	0	1	5	-3	2

First, let us consider variations in the objective function coefficients of variables x_3 and x_4. Since both variables are nonbasic in the optimal basis, the allowed variation for them is given by

$$\delta_3 \geq -\overline{c}_3 = -2 \text{ and } \delta_4 \geq -\overline{c}_4 = -7.$$

Two points to notice. First, notice that both intervals are one-sided. This means that one should only be concerned with variations that decrease the reduced cost value since increases in their value can never cause any changes in the optimality conditions. Second, notice that the allowed variation is trivially the negative value of the reduced cost. For variations that turn the reduced costs negative, the current basis can be utilised as a starting point for the primal simplex.

Now, let us consider a variation in the basic variable x_1. Notice that in this case, we have to analyse the impact in all reduced costs, except for x_1 itself. Using the tableau, we have that $q_l = [1, 0, -3, 2]$ and thus

$$\begin{aligned}
&\delta_1 q_{12} \leq \overline{c}_2 \Rightarrow 0 \leq 0 \\
&\delta_1 q_{13} \leq \overline{c}_3 \Rightarrow \delta_1 \geq -2/3 \\
&\delta_1 q_{14} \leq \overline{c}_4 \Rightarrow \delta_1 \leq 7/2,
\end{aligned}$$

implying that $-2/3 \leq \delta_1 \leq 7/2$. Like before, for a change outside this range, primal simplex can be readily employed.

6.2 Cones and extreme rays

We now change the course of our discussion towards some results that will be useful in identifying two non-ordinary situations when employing the simplex method: unboundedness and infeasibility. Typically, these are consequences of issues related to the data and/or with modelling assumptions and are challenging in that they prevent us from obtaining a solution from the model. As we will see, infeasibility and unboundedness can be identified using duality and are all connected by the notion of cones, which we formally give in Definition 6.1.

Definition 6.1 (Cones)**.** *A set $C \subset \mathbb{R}^n$ is a cone if $\lambda x \in C$ for all $\lambda \geq 0$ and all $x \in C$.*

A cone C can be understood as a set formed by the nonnegative scaling of a collection of vectors $x \in C$. Notice that it implies that $0 \in C$. Often, it will be the case that $0 \in C$ is an extreme point of C and in that case, we say that C is *pointed*. As one might suspect, in the context of linear programming, we will be mostly interested in a specific type of cone, those known are *polyhedral cones*. Polyhedral cones are sets of the form

$$P = \{x \in \mathbb{R}^n : Ax \geq 0\}.$$

Figure 6.1 illustrates a polyhedral cone in $\mathbb{R}^3$ formed by the intersection of three half-spaces.

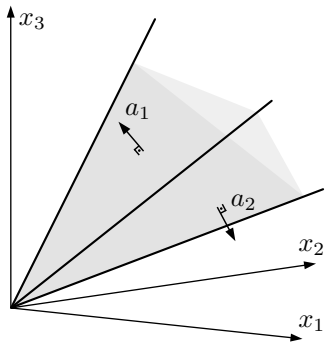


Figure 6.1: A polyhedral cone in $\mathbb{R}^3$ formed by 3 half-spaces. The third (with normal vector a_3 perpendicular to the page plan) cannot be seen

Some interesting properties can be immediately concluded regarding polyhedral cones. First, they are convex sets, since they are polyhedral sets (cf. Theorem 2.8). Also, the origin is an extreme point, and thus, polyhedral cones are always pointed. Furthermore, just like general polyhedral sets, a cone $C \in \mathbb{R}^n$ will always be associated with a collection of n linearly independent vectors. Corollary 6.2 summarises these points. Notice we pose it as a corollary because these are immediate consequences of Theorem 3.5.

Corollary 6.2. *Let $C \subset \mathbb{R}^n$ be a polyhedral cone defined by constraints $\left\{a_i^\top x \geq 0\right\}_{i=1,\ldots,m}$. Then the following are equivalent*

1. *0 is an extreme point of C;*
2. *C does not contain a line;*
3. *There exists n vectors in $a_1, \ldots, a_m$ that are LI.*

Proof. The proof of Theorem 3.5 verbatim, with $P = C$. □

Notice that $0 \in C$ is the unique extreme point of the polyhedral cone C. To see that, let $0 \neq x \in C$, $x_1 = (1/2)x$ and $x_2 = (3/2)x$. Note that $x_1, x_2 \in C$, and $x \neq x_1 \neq x_2$. Setting $\lambda_1 = \lambda_2 = 1/2$, we have that $\lambda_1 x_1 + \lambda_2 x_2 = x$ and thus, x is not an extreme point (cf. Definition 2.11).

6.2.1 Recession cones and extreme rays

We now focus on a specific type of cone, called *recession cones*. In the context of linear optimisation, recession cones are useful for identifying directions of unboundedness. Let us first formally define the concept.

Definition 6.3 (Recession cone)**.** *Consider the polyhedral set $P = \{x \in \mathbb{R}^n : Ax \geq b\}$. The recession cone at $\overline{x} \in P$, denoted* $\mathbf{recc}(P)$*, is defined as*

$$\mathbf{recc}(P) = \{d \in \mathbb{R}^n : A(\overline{x} + \lambda d) \geq b, \lambda \geq 0\} \text{ or } \{d \in \mathbb{R}^n : Ad \geq 0\}.$$

Notice that the definition states that a recession cone comprises all directions d along which one can move from $\overline{x} \in P$ without ever leaving P. However, notice that the definition does not depend on $\overline{x}$, meaning that the recession cone is unique for the polyhedral set P, regardless of its "origin". Furthermore, notice that Definition 6.3 implies that recession cones of polyhedral sets are polyhedral cones.

We say that any directions $d \in \mathbf{recc}(P)$ is a *ray*. Thus, bounded polyhedra can be alternatively defined as polyhedral sets that do not contain rays.

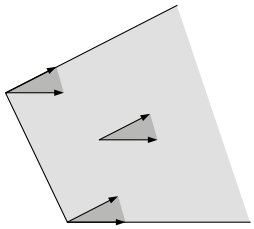


Figure 6.2: Representation of the recession cone of a polyhedral set

Figure 6.2 illustrates the concept of recession cones. Notice that the cone is purposely placed in several places to illustrate the independence of the point $\overline{x} \in P$.

Finally, the recession cone for a standard form polyhedral set $P = \{x \in \mathbb{R}^n : Ax = b, x \geq 0\}$ is given by

$$\mathbf{recc}(P) = \{d \in \mathbb{R}^n : Ad = 0, d \geq 0\}.$$

6.2.2 Unbounded problems

To identify unboundedness in linear programming problems, we must check for the existence of *extreme rays*. Extreme rays are analogous to extreme points, but defined with a "loose" degree of freedom. Definition 6.4 provides a technical definition of extreme rays.

Definition 6.4 (Extreme ray)**.** *Let $C \subset \mathbb{R}^n$ be a nonempty polyhedral cone. A nonzero $x \in C$ is an extreme ray if there are $n - 1$ linearly independent active constraints at x.*

Notice that we are interested in extreme rays of the recession cone $\mathbf{recc}(P)$ of the polyhedral set P. However, it is typical to say that they are extreme rays of P. Figure 6.3 illustrates the concept of extreme rays in polyhedral cones.

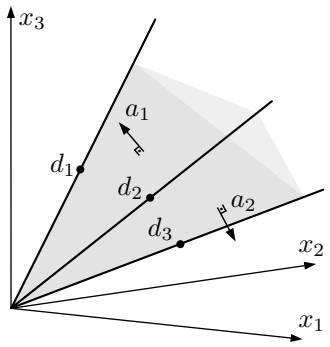


Figure 6.3: A polyhedral cone formed by the intersection of three half-spaces (the normal vector a_3 is perpendicular to the plane of the picture and cannot be seen). Directions d_1, d_2, and d_3 represent extreme rays.

Notice that, just like extreme points, the number of extreme rays is finite by definition. In fact, we say that two extreme rays are equivalent if they are positive multiples corresponding to the same $n-1$ linearly independent active constraints.

The existence of extreme rays can be used to verify unboundedness in linear programming problems. The mere existence of extreme rays does not suffice since unboundedness is a consequence of the extreme ray being a direction of improvement for the objective function. To demonstrate this, let us first describe unboundedness in polyhedral cones, which we can then use to show the unboundedness in polyhedral sets.

Theorem 6.5 (Unboundedness in polyhedral cones)**.** *Let P : min. $\{c^\top x : x \in C\}$, with $C = \{a_i^\top x \geq 0, i \in [m]\}$. The optimal value is equal to $-\infty$ if and only if some extreme ray $d \in C$ satisfies $c^\top d < 0$.*

Proof. If $c^\top d < 0$, then P is unbounded, since $c^\top x \to -\infty$ along d. Also, there exists some $x \in C$ for which $c^\top x < 0$ can be scaled to -1.

Let $P = \{x \in \mathbb{R}^n : a_i^\top x \geq 0, i = 1, \ldots, m, c^\top x = -1\}$. Since $0 \in C$, P has at least one extreme point $\{a_i\}_{i=1}^m$ and thus span $\mathbb{R}^n$ (cf. Theorem 3.5). Let d be one of those. As we have n linearly-independent active constraints at d, $n-1$ of the constraints $\{a_i^\top x \geq 0\}_{i=1}^m$ must be active (plus $c^\top x = -1$), and thus d is an extreme ray. □

We can now expand the result to general polyhedral sets.

Theorem 6.6 (Unboundedness in polyhedral sets)**.** *Let P : min. $\{c^\top x : x \in X\}$ with $X = \{x \in \mathbb{R}^n : Ax \geq b\}$ and assume that the feasible set has at least one extreme point. Then, the optimal value is $-\infty$ if and only if $c^\top d < 0$.*

Proof. As before, if $c^\top d < 0$, then P is unbounded, since $c^\top x \to -\infty$ along d. Now, let D : max. $\{p^\top b : p^\top A = c^\top, p \geq 0\}$ be the dual of P. Recall that, if P is unbounded, then D is infeasible, and so must be D^0 : max. $\{p^\top 0 : p^\top A = c^\top, p \geq 0\}$. This implies that the primal P^0 : min. $\{c^\top x : Ax \geq 0\}$ is unbounded (as 0 is feasible).

The existence of at least one extreme point for P implies that the rows $\{a_i\}_{i=1,\dots,m}$ of A span $\mathbb{R}^n$ and $\mathbf{recc}(X)=\{x \in \mathbb{R}^n : Ax \geq 0\}$ is pointed. Thus, by Theorem 6.5 there exists d such that $c^\top d < 0$. □

We now focus on how this can be utilised in the context of the simplex method. It turns out that once unboundedness is identified in the simplex method, one can extract the extreme ray causing the said unboundedness. In fact, most professional-grade solvers are capable of returning extreme (or unbounded) rays, which is helpful in the process of understanding the causes for unboundedness in the model. We will also see in the next chapter that these extreme rays are also used in the context of specialised solution methods.

To see that is possible, let P : min. $\left\{c^\top x : x \in X\right\}$ with $X = \{x \in \mathbb{R}^n : Ax = b, x \geq 0\}$ and assume that, for a given basis B, we conclude that the optimal value is $-\infty$, that is, the problem is unbounded. In the context of the simplex method, this implies that we found a nonbasic variable x_j for which the reduced cost $\bar{c}_j < 0$ and the j^{th} column of $B^{-1}A_j$ has no positive coefficient. Nevertheless, we can still form the feasible direction $d = [d_B \; d_N]$ as before, with

$$d_B = -B^{-1}A_j \text{ and } d_N = \begin{cases} d_j = 1 \\ d_i = 0, \forall i \in I_N \setminus \{j\}. \end{cases}$$

This direction d is precisely an extreme ray for P. To see that, first, notice that $Ad = 0$ and $d \geq 0$, thus $d \in \mathbf{recc}(X)$. Moreover, there are $n-1$ active constraints at d: m in $Ad = 0$ and $n-m-1$ in $d_i = 0$ for $i \in I_N \setminus \{j\}$. The last thing to notice is that $\bar{c}_j = c^\top d < 0$, which shows the unboundedness in the direction d.

6.2.3 Farkas' lemma

We now focus on the idea of generating certificates of infeasibility for linear programming problems. That is, we show that if a problem P is infeasible, then there is a structure that can be identified to certify the infeasibility. To see how this works, consider the two polyhedral sets

$$X = \{x \in \mathbb{R}^n : Ax = b, x \geq 0\} \text{ and}$$
$$Y = \left\{p \in \mathbb{R}^m : p^\top Ax \geq 0, p^\top b < 0\right\}.$$

If there exists any $p \in Y$, then there is no $x \in X$ for which $p^\top Ax = p^\top b$ (and in turn $Ax = b$) holds. Thus, X must be empty. Notice that this can be used to infer that a problem P with a feasibility set represented by X prior to solving P itself, by means of solving the *feasibility problem* of finding a vector $p \in Y$.

We now pose this relationship more formally via a result generally known as the Farkas' lemma.

Theorem 6.7 (Farkas' lemma)**.** *Let A be a $m \times n$ matrix and $b \in \mathbb{R}^m$. Then, exactly one of the following statements hold*

(1) There exists some $x \geq 0$ such that $Ax = b$;

(2) there exists some vector p such that $p^\top A \geq 0$, $p^\top b < 0$.

Proof. Assume that (1) is satisfied. If $p^\top A \geq 0$, then $p^\top b = p^\top Ax \geq 0$, which violates (2).

Now, consider the primal-dual pair P : min. $\left\{0^\top x : Ax = b, x \geq 0\right\}$ and D : max. $\left\{p^\top b : p^\top A \geq 0\right\}$. Being P infeasible, D must be unbounded (instead of infeasible) since $p = 0$ is feasible for D. Thus, $p^\top b < 0$ for some $p \neq 0$. □

The Farkas' lemma has a nice geometrical interpretation that represents the mutually exclusive relationship between the two sets. For that, notice that we can think of b as being a conic combination of the columns A_j of A, for some $x \geq 0$. If that cannot be the case, then there exists a hyperplane that separates b and the cone formed by the columns of A, $C = \{y \in \mathbb{R}^m : y = Ax\}$. This is illustrated in Figure 6.4. Notice that the separation caused by such a hyperplane with normal vector p implies that $p^\top Ax \geq 0$ and $p^\top b < 0$, i.e., Ax and b are on the opposite sides of the hyperplane.

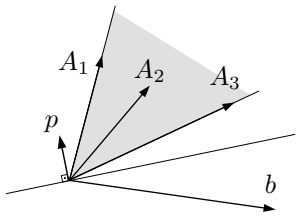


Figure 6.4: Since $b \notin X$, $p^\top x = 0$ separates them

6.3 Exercises

Exercise 6.1: Sensitivity analysis in the RHS

Consider the following linear programming problem and its optimal tableau below:

$$\begin{aligned} \text{min. } & -2x_1 - x_2 + x_3 \\ \text{s.t.: } & x_1 + 2x_2 + x_3 \leq 8 \\ & -x_1 + x_2 - 2x_3 \leq 4 \\ & 3x_1 + x_2 \leq 10 \\ & x_1, x_2, x_3 \geq 0. \end{aligned}$$

	x_1	x_2	x_3	x_4	x_5	x_6	RHS
z	0	0	1.2	0.2	0	0.6	-7.6
x_1	1	0	-0.2	-0.2	0	0.4	2.4
x_2	0	1	0.6	0.6	0	-0.2	2.8
x_5	0	0	-2.8	-0.8	1	0.6	3.6

(a) If you were to choose between increasing in 1 unit the right-hand side of any constraints, which one would you choose, and why? What is the effect of the increase on the optimal cost?

(b) Perform a sensitivity analysis on the model to discover what is the range of alteration in the RHS in which the same effect calculated in item (a) can be expected. *HINT*: JuMP (from version 0.21.6) includes the function "lp_sensitivity_report()" that you can use to help performing the analysis.

Exercise 6.2: Extreme points and extreme rays

(a) Let $P = \{(x_1, x_2) : x_1 - x_2 = 0,\ x_1 + x_2 = 0\}$. What are the extreme points and the extreme rays of P?

(b) Let $P = \{(x_1, x_2) : 4x_1 + 2x_2 \geq 0,\ 2x_1 + x_2 \leq 1\}$. What are the extreme points and the extreme rays of P?

(c) For the polyhedron of part (b), is it possible to express each one of its elements as a convex combination of its extreme points plus a nonnegative linear combination of its extreme rays? Is this compatible with the Resolution Theorem?

Exercise 6.3: From Farkas' lemma to duality

Use the Farkas' lemma to prove the duality theorem for a linear programming problem involving constraints of the form $a_i'x = b_i, a_i'x \geq b_i$, and nonegativity constraints for some of the variables x_j. *Hint*: Start by deriving the form of the set of feasible directions at an optimal solution.

Exercise 6.4: Adding a constraint

Consider the linear programming problem below with optimal basis $[x_1, x_2, x_5, x_6]$ and dual variables $p_1, \ldots, p_4$.

$$\begin{aligned} \text{max. } & 2x_1 + x_2 \\ \text{s.t.: } & 2x_1 + 2x_2 \leq 9 \quad (p_1) \\ & 2x_1 - x_2 \leq 3 \quad (p_2) \\ & x_1 \leq 3 \quad (p_3) \\ & x_2 \leq 4 \quad (p_4) \\ & x_1, x_2 \geq 0. \end{aligned}$$

(a) Find the primal and dual optimal solutions. *HINT*: You can use complementary slackness, once having the primal optimum, to find the dual optimal solution.

(b) Suppose we add a new constraint $6x_1 - x_2 \leq 6$, classify the primal and dual former optimal points stating if they: (i) remain optimal; (ii) remain feasible but not optimal; or (iii) become infeasible.

(c) Consider the new problem from item (b) and find the new dual optimal point through one dual simplex iteration. After that, find the primal optimum.

Chapter 7

Barrier Method for Linear Programming

7.1 Barrier methods

In this chapter, we look into barrier methods as an alternative for solving linear programming problems. Barrier methods stem from early developments of methods for solving constrained nonlinear programming problems which were mainly characterised by the *strict* satisfaction of the constraints throughout the method. Because of this feature, these methods became generally known as interior point methods. This is however not the case anymore, and most implementations of interior point methods benefit from features that allow the search to "leave the interior" of the feasible region. Hence, it became common to refer to these methods with the more general name of barrier methods.

A subclass of barrier methods called *primal-dual methods* distinguishes itself as an efficient method, with practical performance surpassing in many cases that of the simplex method. Currently, most professional-grade solvers have built-in implementations of barrier methods.

In essence, barrier methods are the method of choice of many nonlinear *local solvers*, which are targeted towards nonlinear optimisation problems. The term local refers to the fact that solutions found can only be guaranteed to be locally optimal. Of course, for convex optimisation problems, this is not an issue, as a local solution is globally optimal.

In general, the simplex methods often perform better in small and medium problems, whilst barrier methods typically perform better on large-scale problems. This is largely because, as we will see, the main operation in a barrier method is solving (large) linear systems of equations, which can be done rather efficiently and in ways that exploit the structure of the problem (e.g., matrix sparsity can be exploited by factorisation techniques) to reap computational performance improvements.

7.2 Newton's method with equality constraints

In essence, barrier methods employ a variant of Newton's method to solve the optimality conditions of optimisation problems. In the context of linear programming problems, this is equivalent to finding solutions that are both primal and dual feasible[1].

We consider a version of Newton's method called the Newton-Raphson (NR) method, which was originally conceived for finding the roots of vector functions. Let $f : \mathbb{R}^n \to \mathbb{R}^n$, with $f_i : \mathbb{R}^n \to \mathbb{R}$ differentiable for $i \in [n]$.

[1] Recall that the satisfaction of complementarity conditions in the linear case is a consequence of primal and dual feasibility, cf. Theorem 5.4.

We wish to find a solution x^* that is a root for f, i.e., $f(x^*) = 0$. For that, we must solve the system of equations given by

$$f(x) = \begin{bmatrix} f_1(x) \\ \vdots \\ f_n(x) \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}.$$

The NR method starts from an initial guess x^k for x^* and iterates by finding the root for a *linear* (i.e., first-order Taylor) approximation of f at x^k. Under suitable conditions, including having a starting point x^0 that is within a neighbourhood of the root of f, the sequence $\left\{x^k\right\}_{k\to\infty}$ converges to x^*.

Let us clearly state how the method iterates. At a given x^k, the first-order approximation of $f(x)$ is given by

$$f(x^k + d) = f(x^k) + \nabla f(x^k)^\top d,$$

where $\nabla f(x^k)$ is the *Jacobian* of $f(x)$, which is defined as

$$\nabla f(x^k) = \begin{bmatrix} \nabla f_1(x^k)^\top \\ \vdots \\ \nabla f_n(x^k)^\top \end{bmatrix}.$$

The algorithm proceeds by finding the step to be taken from x^k to reach the root of the first-order approximation of f at x^k. This means that we want to obtain a Newton direction d such that it solves the first-order approximation of $f(x^k + d) = 0$. Thus, we must solve

$$f(x^k) + \nabla f(x^k)^\top d = 0 \Rightarrow d = -\nabla f(x^k)^{-1} f(x^k).$$

Once the vector d is obtained, the root for linear approximation is given by $x^{k+1} = x^k + d$. This process repeats until convergence is achieved. In this case, convergence is measured as the norm of the resulting vector d at iteration k, denoted by d^k. The procedure is stopped when for a given $\epsilon > 0$, we have that $\|d^k\|_2 \leq \epsilon$.

Consider the following numerical example. Suppose we would like to find the root of f with $x^0 = (1, 0, 1)$, where f is given by

$$f(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \\ f_3(x) \end{bmatrix} = \begin{bmatrix} x_1^2 + x_2^2 + x_3^2 - 3 \\ x_1^2 + x_2^2 - x_3 - 1 \\ x_1 + x_2 + x_3 - 3 \end{bmatrix}.$$

The Jacobian of f is given by

$$\nabla f(x) = \begin{bmatrix} 2x_1 & 2x_2 & 2x_3 \\ 2x_1 & 2x_2 & -1 \\ 1 & 1 & 1 \end{bmatrix}.$$

The algorithm starts by calculating d^0, which is given by

$$d^0 = -[\nabla f(x^0)]^{-1} f(x^0) = -\begin{bmatrix} 2 & 0 & 2 \\ 2 & 0 & -1 \\ 1 & 1 & 1 \end{bmatrix}^{-1} \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix} = \begin{bmatrix} 1/2 \\ 1/2 \\ 0 \end{bmatrix}.$$

Thus, the first point is

$$x^1 = x^0 + d^0 = \begin{bmatrix} 3/2 & 1/2 & 1 \end{bmatrix}.$$

To infer that the method has converged, we can either check whether $f(x^k) \approx 0$ or whether $||x^{k+1} - x^k||_2 = ||d^k||_2 \approx 0$. As $||x^1 - x^0||_2 = ||d^0||_2 \approx 0.7$, and $f(x^1) = [1/2, 1/2, 0]$, the point x^1 is not a root, and the method carries on until we find that $||d^k|| < \epsilon$. If we adopt a numerical tolerance of $\epsilon = 0.01$, meaning that any number below this threshold is deemed acceptably close to zero, then $x^* = (1, 1, 1)$ is reached after approximately 20 iterations.

7.3 Interior point methods linear programming problems

We start by focusing on the primal-dual interior point method, as originally proposed. Then, we will focus on the modifications that make it capable of iterating through some not necessarily interior solutions in a strict sense.

As usual, let $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, $A \in \mathbb{R}^{m \times n}$, and $J = [n]$. We start by considering our linear programming problem in standard form

$$\begin{aligned} (P): \text{ min. } & c^\top x \\ \text{s.t.: } & Ax = b, \\ & x \geq 0, \end{aligned}$$

and its associated dual, which is stated as

$$\begin{aligned} (D): \text{ max. } & b^\top p \\ \text{s.t.: } & A^\top p + u = c, \\ & u \geq 0. \end{aligned}$$

Notice that the dual is also posed in a form where the inequalities (originally $A^\top p \leq c$) are converted to equalities requiring the additional variable u. Thus, this makes $u \in \mathbb{R}^n$ an n-sized vector, while $p \in \mathbb{R}^m$ is, as before, a m-sized vector. An alternative way to think about u is as if it were the dual variable associated with the nonnegativity constraints $x \geq 0$.

Recall that the optimality conditions for problem P can be expressed as

$$Ax = b, \quad x \geq 0, \tag{7.1}$$

$$A^\top p + u = c, \quad u \geq 0, \tag{7.2}$$

$$u^\top x = 0. \tag{7.3}$$

The first two conditions are primal (7.1) and dual (7.2) feasibility conditions, while (7.3) is an alternative way of stating complementarity conditions on the nonnegativity constraints $x \geq 0$.

One initial idea could be simply, from a purely methodological standpoint, to employ NR to solve the above system of equations. The caveat, however, is that one must observe and retain the nonnegativity conditions $x > 0$ and $u > 0$, which are an important complicating factor in this setting. Indeed, departing from a feasible solution (x, p, u), one could employ NR to find a solution for (7.1)-(7.3) while controlling the steps taken in each Newton direction such that the nonnegativity conditions $x > 0$ and $u > 0$ are retained, in a similar fashion to how it is done in the simplex method. However, it turns out that Newton directions obtained from successive iterations of (7.1)-(7.3) typically require the steps to be considerably small so that the nonnegativity conditions are retained, which renders the algorithm less useful from a practical standpoint.

This is precisely when the notion of an interior solution plays a role. An *interior point* is defined as a point that satisfies primal and dual feasibility conditions strictly, that is $Ax = b$ with $x > 0$,

and $A^\top p + u = c$ with $u > 0$, implying that the complementarity conditions (7.3) are *violated*. This notion of interior points is useful in that it allows for the definition of less "aggressive" Newton directions, which aim towards directions that reduce the amount of violation in the complementarity conditions.

An alternative way to think about the interior point method is to consider that it iterates through primal and dual feasible solutions that do not form a primal-dual pair satisfying strong duality at first, gradually progressing in the direction of satisfying it (cf. Theorem 5.3). This can be noticed in the primal and dual feasibility conditions

$$\begin{aligned} & A^\top p + u = c \Leftrightarrow \\ & p^\top (Ax) + u^\top x = c^\top x \Leftrightarrow \\ & u^\top x = c^\top x - p^\top b, \end{aligned} \tag{7.4}$$

that is, for $u^\top x > 0$, we have that $c^\top x > p^\top b$ implying that weak duality (cf. Theorem 5.1) holds, but not strong duality.

The algorithm alternatively considers the system with *relaxed* complementarity conditions in which we define the scalar $\mu > 0$ and restate the optimality conditions as

$$\begin{aligned} & Ax = b, \quad x \geq 0, \\ & A^\top p + u = c, \quad u \geq 0, \\ & u_j x_j = \mu, \ \forall j \in J. \end{aligned}$$

Then, we use these relaxed optimality conditions to obtain Newton directions, while, simultaneously, gradually making $\mu \to 0$. By doing so, one can obtain not only directions that can be explored with larger step sizes, thus making more progress per iteration but also yield methods with better numerical properties.

To see how that can be made operational, let us first define some helpful notation. Let $X \in \mathbb{R}^{n \times n}$ and $U \in \mathbb{R}^{n \times n}$ be defined as

$$X = \mathbf{diag}(x) = \begin{bmatrix} \ddots & & \\ & x_i & \\ & & \ddots \end{bmatrix} \text{ and } U = \mathbf{diag}(u) = \begin{bmatrix} \ddots & & \\ & u_i & \\ & & \ddots \end{bmatrix}$$

and let $e = [1, \ldots, 1]^\top$ be a vector of ones of suitable dimension. We can rewrite the optimality conditions (7.1)-(7.3) in matrix form as

$$\begin{aligned} & Ax = b, \ x > 0, \\ & A^\top p + u = c, \ u > 0, \\ & XUe = 0, \end{aligned}$$

and, analogously, state their relaxed version (i.e., with relaxed complementarity conditions) as

$$Ax = b, \ x > 0, \tag{7.5}$$

$$A^\top p + u = c, \ u > 0, \tag{7.6}$$

$$XUe = \mu e. \tag{7.7}$$

We start from a feasible solution (x^k, p^k, u^k). We can then employ NR to solve the system (7.5)-(7.7) for a given value of μ. That amounts to finding the Newton direction d that solves $f(x^k) + \nabla f(x^k)^\top d = 0$, in which

$$f(x^k) = \begin{bmatrix} Ax^k - b \\ A^\top p^k + u^k - c \\ X^k U^k e - \mu e \end{bmatrix}, \ \nabla f(x^k) = \begin{bmatrix} A & 0 & 0 \\ 0 & A^\top & I \\ U^k & 0 & X^k \end{bmatrix} \text{ and } d = (d_x^k, d_p^k, d_u^k) = \begin{bmatrix} x - x^k \\ p - p^k \\ u - u^k \end{bmatrix}.$$

Once the direction $d^k = (d_x^k, d_p^k, d_u^k)$ is obtained, we must calculate step sizes that can be taken in the direction d^k that also retain feasibility conditions, meaning that they do not violate $x > 0$ and $u > 0$ at the new point. One simple idea is to follow the same procedure as the simplex method. That is, let θ_p^k and θ_d^k be the iteration k step sizes for x and (p, u), respectively. Then they can be set as

$$\theta_p^k = \min_{i: d_{x_i}^k < 0} -\frac{x_i^k}{d_{x_i}^k} \text{ and } \theta_d^k = \min_{i: d_{u_i}^k < 0} -\frac{u_i^k}{d_{u_i}^k},$$

where $d_{x_i}^k$ and $d_{u_i}^k$ are the i-th component of the vectors d_x^k and d_u^k, respectively, and $i \in J$. In practice, there are alternative, and arguably more efficient, ways to set step sizes, but they all are such that their maximum sizes are θ_p and θ_d as above and always strictly smaller than one (notice that our constraints are such that x and u are strictly positive).

Once we calculated the appropriate step sizes, we can then make

$$(x^{k+1}, p^{k+1}, u^{k+1}) = (x^k + \theta_p^k d_x^k, p^k + \theta_d^k d_p^k, u^k + \theta_d^k d_u^k).$$

Then, we proceed by updating μ as $\mu = \beta\mu$, with $\beta \in (0, 1)$ and repeat the same procedure until convergence, i.e., until μ is sufficiently small.

Before discussing in more detail the practicalities of the method, let us take an alternative perspective on showing how the notion of interior plays a role in the design of the algorithm, which requires us to consider the notion of barrier functions. This will also serve as a justification for the name "barrier methods".

7.4 Barrier methods

Barrier methods are a class of optimisation methods designed to handle inequality-constrained problems of the form

$$\begin{aligned} (P): \min. \ & f(x) \\ \text{s.t.: } & g_i(x) \leq 0, \ i = 1, \ldots, m \\ & Ax = b. \end{aligned}$$

These methods use the notion of *barrier functions* to remove inequality constraints and have them represented implicitly as an objective function term. This allows for the use of NR as the underpinning numerical method of a solution algorithm.

To see how the method works, we can think of a barrier function as a surrogate for a *feasibility indicator function* I that reacts to infeasibility in $g_i(x) \leq 0$, $\forall i \in [m]$[2]. That is, $I : \mathbb{R} \to \mathbb{R}$ is such that

$$I(y) = \begin{cases} 0, & \text{if } y \leq 0 \\ \infty, & \text{if } y > 0. \end{cases}$$

[2]The technical name of this function is the *characteristic function*, which is arguably a less informative name.

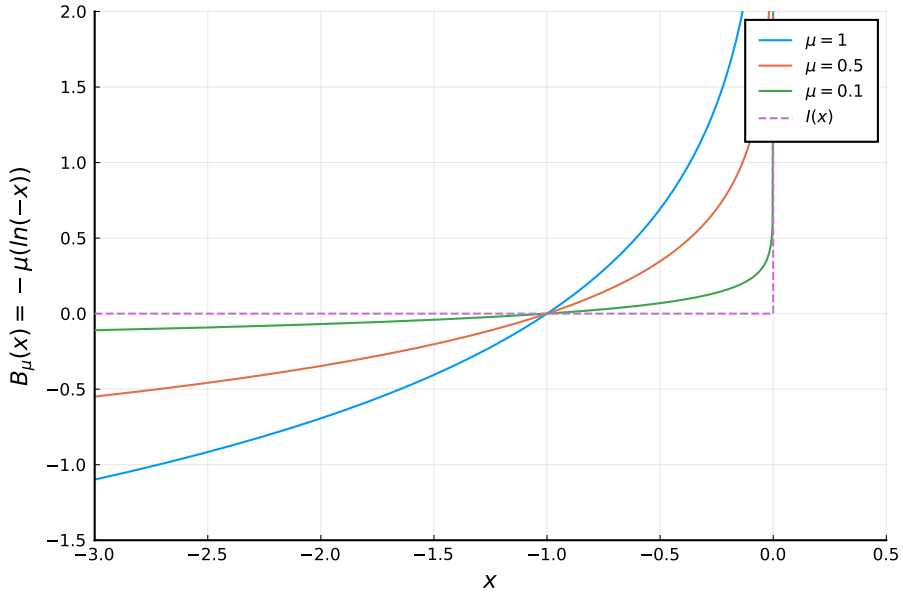


Figure 7.1: Alternative barrier functions for different values of μ. The dashed line represents the indicator function $I(x)$ going to infinity when $x > 0$

With that definition for I at hand, we can then reformulate our problem as

$$\begin{aligned} \text{min.} \quad & f(x) + \sum_{i=1}^{m} I(g_i(x)) \\ \text{s.t.:} \quad & Ax = b. \end{aligned}$$

The issue, however, is that I is not numerically favourable due to its nature of "shooting to infinity" (including the discontinuity it creates) whenever a solution x is infeasible to the original problem. To circumvent that, we can use barrier functions instead, which are chosen to mimic the behaviour of I whilst retaining more favourable numerical properties, in particular differentiability. The most widespread choice for barrier functions is the logarithmic barrier, which is given by

$$B_\mu(y) = -\mu \ln(-y)$$

where $\mu > 0$ sets the accuracy of the barrier term $B_\mu(y)$. Figure 7.1 illustrates the influence of μ in the shape of the barrier function for the unidimensional case. Notice how, as μ decreases, the logarithmic barrier more closely resembles the indicator function I.

Using B_μ as the barrier function, the *barrier problem* P_μ can be formulated as

$$\begin{aligned} (P_\mu): \text{min.} \quad & f(x) - \mu \sum_{i=1}^{m} \ln(-g_i(x)) \\ \text{s.t.:} \quad & Ax = b. \end{aligned}$$

This formulation has a number of important benefits. First, notice how this problem is such that if one were to apply NR to solve its optimality conditions, one would be faced with solving linear

systems of equations, regardless of the nature of the constraints $g_i(x) \leq 0$, $i = [m]$ (recall that NR solves first-order approximations of the original system of equations). This creates a bridge between linear algebra techniques and a wide range of nonlinear optimisation problems.

Moreover, as discussed earlier, one can also gradually decrease μ by making $\mu^{k+1} = \beta\mu$ with $\beta \in (0, 1)$. This method is an idea generally known as Sequential Unconstrained Minimisation Technique (SUMT). As one may suspect, as $\mu \to 0$, we have that $x^*(\mu) \to x^*$, where $x^*(\mu)$ and x^* are the optimal values for problems P_μ and P, respectively. However, for small values of μ, the barrier problem becomes challenging numerically, which does not come as a surprise as the barrier resembles more and more the indicator function I and all its associated numerical issues.

Let us illustrate the above with an example. Consider the nonlinear problem

$$P : \text{min. } \left\{ f(x) = (x+1)^2 : x \geq 0 \right\}.$$

Notice that the unconstrained optimum would be $x = -1$ but the constrained optimum is $x^* = 0$. For this example, the barrier problem P_μ is

$$P_\mu : \text{min. } f(x) + B_\mu(x) = (x+1)^2 - \mu \ln(x).$$

Because this is a univariate convex function, we know that the optimum is attained where the derivative $(f(x) + B_\mu(x))' = 0$. Thus, we have that

$$\begin{aligned} f\,'(x) + B'_\mu(x) &= 0 \\ 2(x+1) - \frac{\mu}{x} &= 2x^2 + 2x - \mu = 0. \end{aligned}$$

The positive solution (since we must have that $x > 0$) of $2x^2 + 2x - \mu = 0$ is given by

$$x^*(\mu) = \frac{-2 + \sqrt{4 + 8\mu}}{4}.$$

Notice that $\lim_{\mu \to 0} x^*(\mu) = 0$, which is indeed the optimal x^* for P. Figure 7.2 plots the function $f(x) + B_\mu(x)$ for alternative values of μ indicating their minima (found by substituting the value of μ in the expression for $x^*(\mu)$).

The notion of interiority

There is an interesting link between barrier methods and the notion of interior points, which is, in a way, the reason why the two ideas are equivalent. Before we show they are indeed equivalent, let us explore this notion of interiority.

For a large enough μ, the solution of the barrier problem P_μ is close to the *analytic centre* of the polyhedral feasibility set. The analytic centre of a polyhedral set is the point at which the distance to all of the hyperplanes forming the set is maximal. More precisely, consider the polyhedral set $S = \{x \in \mathbb{R}^n : Ax \leq b\}$. The analytic centre of S is given by the optimal solution x^* of the following problem

$$\begin{aligned} \max_x \; & \prod_{i=1}^{m} (b_i - a_i^\top x) \\ \text{s.t.: } & x \in S. \end{aligned}$$

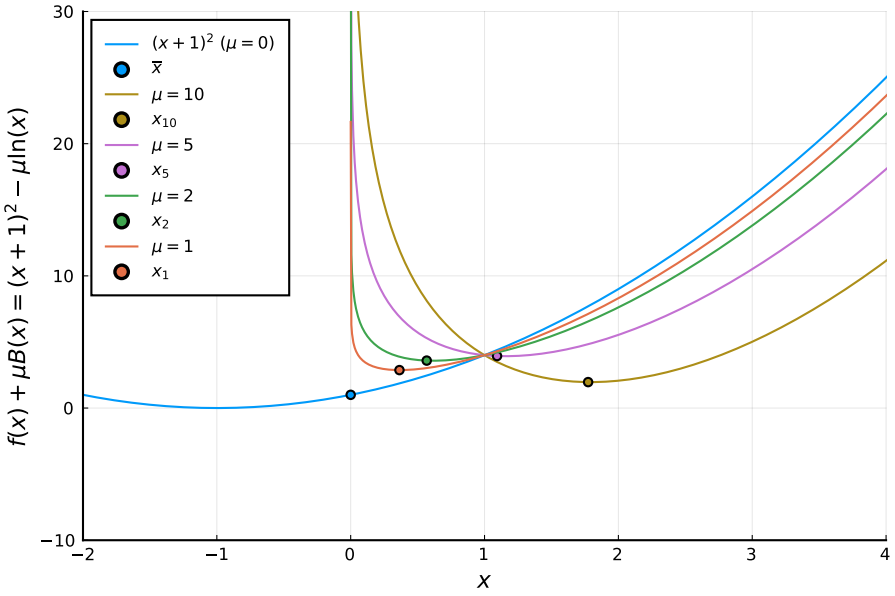


Figure 7.2: The optimal values of P_μ for different values of μ. Notice the trajectory formed by the points $x^*(\mu)$ as $\mu \to 0$.

Notice that we obtain an equivalent problem, that is, a problem would return the same optimal solution x^*, if we take the logarithm of the objective function, this would lead to

$$\begin{aligned} \min_x \ & \sum_{i=1}^{m} -\ln(b_i - a_i^\top x) \\ \text{s.t.: } & x \in S. \end{aligned}$$

This allows us to infer something about the behaviour of the barrier method. For larger values of μ, the optimal solution of $x^*(\mu)$ will lie close to the analytical centre of the feasible region. On the other hand, as μ diminishes, the "pull" towards the centre slowly decays whilst the pull caused by the objective function (that is, by its gradient $-\nabla f(x)$) slowly becomes more prevalent and steers the solution towards x^*.

7.5 Barrier methods for linear programming problems

Let us consider again our linear programming problem in standard form

$$(P) : \text{min. } \left\{ c^\top x : Ax = b, x \geq 0 \right\}$$

to which we want to devise a barrier problem. The barrier problem for P is

$$\begin{aligned} (P_\mu) : \ \text{min. } \ & c^\top x - \mu \sum_{i=1}^{n} \ln(x_j) \\ \text{s.t.: } & Ax = b \ \ (\text{and } \ x > 0). \end{aligned}$$

By looking at this formulation, and relating to the previous discussion, a barrier method for a linear programming problem operates such that in earlier iterations, or for larger μ values, the optimal solution $x(\mu)^*$ of P_μ tends to be pushed towards the centre of the feasible region, where $x > 0$. As $\mu \to 0$, the influence of $c^\top x$ gradually takes over, steering the solution towards the optimal x^* of P.

The most remarkable result, which ties together interior point methods and barrier methods for linear programming problems, is the following. If we derive the Karush-Kuhn-Tucker optimality conditions[3] for P_μ, we obtain the following set of conditions

$$\begin{aligned} &Ax = b,\ x > 0 \\ &A^\top p = c - \mu \left(\frac{1}{x_1}, \ldots, \frac{1}{x_n} \right). \end{aligned}$$

Using the same notation as before (i.e., defining X and U), these can be equivalently rewritten as

$$\begin{aligned} &Ax = b,\ x > 0 \\ &A^\top p + u = c \\ &u = \mu X^{-1} e \ \Rightarrow\ XUe = \mu e, \end{aligned}$$

which are exactly (7.5)-(7.7). Thus, it becomes clear that relaxing the complementarity conditions in the way that we have done before is equivalent to imposing logarithmic barriers to the nonnegativity constraints of x.

There are many important consequences for the analysis of the method once this link is established. One immediate consequence relates to the convergence of the method, meaning that as primal and dual feasibility conditions are satisfied throughout the iterations of the method, as $\mu \to 0$, complementarity conditions are satisfied in the limit, thus converging to the solution of the original linear problem P.

Indeed, the trajectory formed by successive solutions $\{(x(\mu_k), p(\mu_k), u(\mu_k))\}_{k=1,2,\ldots}$ is called the *central path*, which is a consequence of the interiority forced by the barrier function. This property is the reason why barrier methods are sometimes called "path-following" methods.

Recall from (7.4) that $u^\top x$ provides us with an estimate of the current *duality gap*, which can be used to estimate how far we are from the optimal solution. Moreover, notice that

$$u^\top x = \sum_{i=1}^{n} u(\mu)_i x(\mu)_i = n\mu$$

meaning that the term μ indicates the average amount of violation per x and u variable pairs at (x^k, p^k, u^k).

A practical implementation of the barrier method

The version of the barrier method most often used has a few additional ideas incorporated into the algorithm. One of the main ideas is using a *single Newton step* for each value of μ. That effectively means that the iterates do not delineate exactly the central path formed by successively smaller values of μ, but rather follow it only approximately.

[3]The Karush-Kuhn-Tucker, or KKT conditions, are the conditions that a solution must satisfy to be a local optimal solution for a constrained optimisation problem. If the optimisation problem is convex and regularity conditions on the constraints are met, then the KKT conditions are sufficient for global optimality

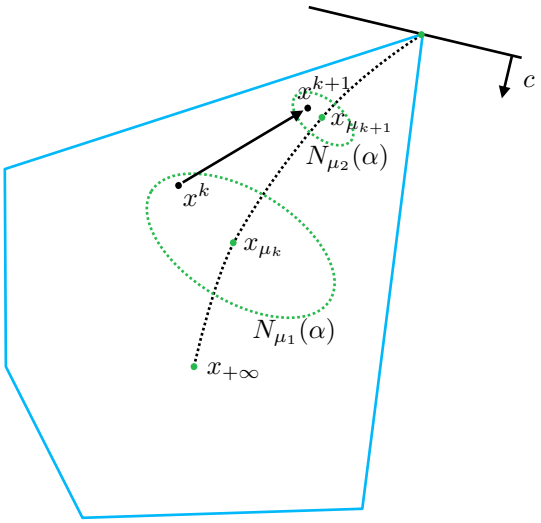


Figure 7.3: An illustrative representation of the central path and how the method follows it approximately

More precisely, assume we start with $\mu_k > 0$ and a (x^k, p^k, u^k) close to $(x(\mu_k), p(\mu_k), u(\mu_k))$. Then, for a small $\beta \in (0,1)$, a Newton step with $\mu_{k+1} = \beta\mu_k$ leads to $(x^{k+1}, p^{k+1}, u^{k+1})$ close to $x(\mu_{k+1}), p(\mu_{k+1}), u(\mu_{k+1})$. Now, to be more precise in terms of what is meant by close, we must refer to the notion of *central paths neighbourhoods*.

A common neighbourhood often used in convergence analysis of the barrier method is

$$N_{\mu_k}(\alpha) = \{(x, p, u) : \|XUe - \mu_k e\|_2 \leq \alpha\mu_k\},$$

which essentially consists of a bound on the difference between the expected value of the complementarity condition violation μ and that observed by the current solution (x^k, p^k, u^k). The parameter $\alpha \in (0, 1]$ is an arbitrary scalar that dictates the amount of such difference tolerated and effectively controls how wide the neighbourhood is. Then, by setting values for β and α that satisfy $\beta = 1 - \frac{\sigma}{\sqrt{n}}$ for some $\sigma \in (0,1)$ (e.g., $\alpha = \sigma = 0.1$), one can guarantee that the complementarity violation of the next iteration to be bounded such that $(x^{k+1}, p^{k+1}, u^{k+1}) \in N_{\mu_{k+1}}(\alpha)$ [3]. Figure 7.3 illustrate this idea, depicting two successive iterates of the method remaining within the shrinking neighbourhoods $N_{\mu_k}(\alpha)$.

Another important aspect of the method relates to the feasibility requirements of each step. Current implementations of the method can be shown to converge under particular conditions even if the feasibility requirements are eased in the Newton system. In this case, it needs to be shown that the Newton direction is also such that the amount of infeasibility in the system decreases as the algorithm progresses.

The infeasible version of the algorithm is such that the so-called perturbed (or relaxed) system

$$\begin{bmatrix} A & 0 & 0 \\ 0 & A^\top & I \\ U^k & 0 & X^k \end{bmatrix} \begin{bmatrix} d_x^{k+1} \\ d_p^{k+1} \\ d_u^{k+1} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \mu_{k+1}e - X^kU^ke \end{bmatrix}$$

becomes

$$\begin{bmatrix} A & 0 & 0 \\ 0 & A^\top & I \\ U^k & 0 & X^k \end{bmatrix} \begin{bmatrix} d_x^{k+1} \\ d_p^{k+1} \\ d_u^{k+1} \end{bmatrix} = - \begin{bmatrix} Ax^k - b \\ A^\top p^k + u^k - c \\ X^k U^k e - \mu_{k+1} e \end{bmatrix}, \tag{7.8}$$

where $\mu_{k+1} = \beta \mu_k$.

The primal residual $r_p(x) = Ax - b$ and dual residual $r_d(p, u) = A^\top p + u - c$ allow the method to iterate through solutions (x^k, p^k, u^k) that do not satisfy primal and/ or dual feasibility. A caveat though is that the convergence analysis of this variant is more involved and requires additional assumptions on the parameterisation of the algorithm for convergence.

To see how the residuals decay at each iteration, notice the following. Let $r(w) = r(x, p, u) = (r_p(x), r_d(p, u))$. The optimality conditions (7.1)-(7.3) require the residuals to vanish, that is, $r(w) = 0$. Now, let us consider the first-order approximation for r at w^k for a Newton step d_w. This amounts to the following:

$$r(w^k + d_w) \approx r(w^k) + Dr(w^k) d_w,$$

where $Dr(w^k)$ is the Jacobian of r evaluated at w^k. One can notice that, in this notation, the solution d_w^{k+1} to (7.8) is such that

$$Dr(w^k) d_w^{k+1} = -r(w^k).$$

Now, let us consider the directional derivative of the norm of the updated residues in the direction d_w^{k+1}. That leads to the following conclusion

$$\begin{aligned} \frac{d}{dt} \left\| r(w^k + t d_w^{k+1}) \right\|_2^2 \Big|_{t=0} &= 2 r(w^k + t d_w^{k+1}) Dr(w^k + t d_w^{k+1}) d_w^{k+1} \\ &= 2 r(w^k)^\top Dr(w^k) d_w^{k+1} \\ &= -2 \|r(w^k)\|_2^2 < 0. \end{aligned}$$

This shows that the direction d_w^{k+1} is a descent direction for the norm of the residues at w^k which leads to their eventual vanishing.

We are finally ready to pose the pseudocode of a working version of the barrier method, which is displayed in Algorithm 4.

Algorithm 4 Barrier method for LP

1: **initialise** (x^0, p^0, u^0), $\epsilon > 0$, $\mu_0 = \mu_1 > 0$, $\beta \in (0, 1)$, $k = 0$.
2: **while** $n\mu_k > \epsilon$ **do**
3: compute $d^{k+1} = (d_x^{k+1}, d_p^{k+1}, d_u^{k+1})$ using (7.8)
4: compute appropriate step size $\theta^{k+1} = (\theta_p^{k+1}, \theta_d^{k+1})$
5: $(x^{k+1}, p^{k+1}, u^{k+1}) = (x^k, p^k, u^k) + \theta^{k+1} d^{k+1}$
6: $k = k + 1$
7: $\mu_{k+1} = \beta \mu_k$
8: **end while**
9: **return** (x^k, p^k, u^k).

Some final remarks are worth making. Barrier methods are perhaps the only class of methods that are known for being strong contenders to the dominance of simplex method-based approaches for

linear programming problems. This stems from both a theoretical and an experimental perspective. The convergence analysis available in [3] shows that the number of iterations of the barrier method, for suitable parameterisation, is $O(\sqrt{n}\ln(1/\epsilon))$, whilst, for the simplex method, a similar analysis would bound the number of iterations to be of the order $O(2^n)$. A conclusion stemming from these results would be that barrier methods, being polynomial complexity algorithms, should outperform the simplex method.

But in practice, this is not necessarily the case. Despite its complexity analysis, the simplex method in practice often requires $O(m)$ iterations (i.e., a typically modest multiple of m), where m is the number of columns in the basis. On the other hand, practical observation has shown barrier methods to typically require $\sqrt{n}$ iterations or less and appear to be somewhat insensitive to growth in the number of variables further to a certain point. However, one iteration of the simplex method is rather inexpensive from a computational standpoint, while one iteration of a barrier method is a potentially computationally demanding operation (as it requires the solution of a large linear system of equations).

In general, simplex-method-based solvers are faster on problems of small to medium dimensions, while barrier methods are competitive, and often faster, on large problems. However, this is not a rule, as the performance is dependent on the structure of the particular application. In the end, it all boils down to how effectively the underlying liner system solver can exploit particular structural features that allow using dedicated numerical algebra methods (e.g., sparsity and the use of Cholesky decomposition combined with triangular solves).

Moreover, barrier methods are generally not able to take full advantage of any prior knowledge about the solution, such as an estimate of the solution itself or some suboptimal basis. Hence, barrier methods are less useful than simplex approaches in situations in which "warm-start" information is available. One situation of this type involves branch-and-bound algorithms for solving integer programs, where each node in the branch-and-bound tree requires the solution of a linear program that differs only slightly from one already solved in the parent node. In other situations, we may wish to solve a sequence of linear programs in which the data is perturbed slightly to investigate the sensitivity of the solutions, or in which we approximate a non-linear optimisation problem by a sequence of linear programs. In none of the aforementioned scenarios can barrier methods be used as efficiently as the simplex method (dual or primal, depending on the context).

7.6 Exercises

Exercise 7.1: Step size rule for the barrier method

Solve the following optimisation problem using the barrier method.

$$
\begin{aligned}
(P) \quad \min \quad & x_1 + x_2 && (7.9)\\
\text{s.t.} \quad & 2x_1 + x_2 \geq 8 && (7.10)\\
& x_1 + 2x_2 \geq 10 && (7.11)\\
& x_i \geq 0, \ \forall i \in \{1, 2\} && (7.12)
\end{aligned}
$$

(a) Solve the problem using a constant step size $\theta = 1$.

(b) Derive a step size rule for the barrier method based on retaining primal and dual feasibility.

(c) Amend the solution from part (a) to incorporate the step size calculation stage. Utilise the step size rule derived in part (b). Then, resolve the problem.

Chapter 8

Integer Programming Models

8.1 Types of integer programming problems

In this chapter, we will consider problems in which we have the presence of integer variables. As we will see in the next chapters, the inclusion of integrality requirements imposes further computational and theoretical challenges that we must overcome to be able to solve these problems. On the other hand, being able to consider integer variables allows for the modelling of far more complex and sophisticated systems. To an extent, this is precisely why integer programming problems, or more specifically, mixed-integer programming problems, are by far the most common in practice.

As we did in the first chapter, we will start our discussion by presenting general problems that have particular structures. These structures can often be identified in larger and more complex settings. This will also help exemplify how integer variables can be used to model particular features of optimisation problems.

Let us first specify what we mean by an integer programming problem. Our starting point is a linear programming problem:

$$\begin{aligned} (\mathrm{P}) : \min.\ & c^\top x \\ \text{s.t.:}\ & Ax \leq b \\ & x \geq 0, \end{aligned}$$

where A is an $m \times n$ matrix, c an n-dimensional vector, b an m-dimensional vector, and x a vector of n decision variables.

We say that P is an *integer programming* problem if the variables x must take integer values, i.e., $x \in \mathbb{Z}^n$. If the variables are further constrained to be binary, i.e., $x \in \{0,1\}^n$, we say that it is a *binary programming* problem. Perhaps the most common setting is when only a subset of the variables are constrained to be integer or binary (say p of them), i.e., $x \in \mathbb{R}^{n-p} \times \mathbb{Z}^p$. This is what is referred to as *mixed-integer programming*, or MIP. The most common setting for integer programming problems is to have binary variables only or a combination of binary and continuous variables.

One important distinction must be made. A closely related concept is that of *combinatorial optimisation problems*, which refer to problems of the form

$$\min_{S \subseteq N} \left\{ \sum_{j \in S} c_j : S \in \mathcal{F} \subseteq N \right\},$$

where c_j, $\forall j \in N$, is a weight, and $\mathcal{F}$ is a family of feasible subsets. As the name suggests, in these problems, we are trying to form combinations of elements such that a measure (i.e., an objective

function) is optimised. Integer programming happens to be an important framework for expressing combinatorial optimisation problems, though both integer programming and combinatorial optimisation expand further to other settings as well. To see this connection, let us define an *incidence vector* x^S of S such that

$$x_j^S = \begin{cases} 1, & \text{if } j \in S \\ 0, & \text{otherwise.} \end{cases}$$

Incidence vectors will permeate many of the (mixed-)integer programming formulations we will discuss. Notice that once x^S is defined, the objective function simply becomes $\sum_{j \in N} c_j x_j$. Integer programming formulations are particularly suited for combinatorial optimisation problems when $\mathcal{F}$ can be represented by a collection of linear constraints.

8.2 (Mixed-)integer programming applications

We will consider now a few examples of integer and mixed-integer programming models with somewhat general structures. As we will see, many of these examples have features that can be combined into more general models.

8.2.1 The assignment problem

Consider the following setting. Assume that we must execute n jobs, which must be assigned to the same number n of distinct workers. Each job can be assigned to a worker only, and analogously, each worker can only be assigned to one job. Assigning a worker i to a job j costs C_{ij}, which could measure, e.g., the time taken by worker i to execute job j. Our objective is to find a minimum-cost assignment between workers and jobs. Figure 8.1a illustrates all possible worker-job assignments as arcs between nodes representing workers on the left and jobs on the right. Figure 8.1b represents one possible assignment.

Figure 8.1: An illustration of all potential assignments as a graph and an example of one possible assignment, with total cost $C_{12} + C_{31} + C_{24} + C_{43}$

To represent the problem, let $x_{ij} = 1$ if worker i is assigned to job j and 0, otherwise. Let N be a set of indices of workers and jobs (we can use the same set since they are of the same number).

The integer programming model that represents the assignment problem is given by

$$\begin{aligned}(AP): \min. \quad & \sum_{i\in N}\sum_{j\in N} C_{ij}x_{ij} \\ \text{s.t.:} \quad & \sum_{j\in N} x_{ij} = 1,\ \forall i \in N \\ & \sum_{i\in N} x_{ij} = 1,\ \forall j \in N \\ & x_{ij} \in \{0,1\},\ \forall i, \forall j \in N.\end{aligned}$$

Before we proceed, let us make a parallel to combinatorial problems. The assignment problem is an example of a combinatorial problem, which can be posed by making (i) N the set of all job-worker pairs (i, j); (ii) $S \in \mathcal{F}$ the (i, j) pairs in which i and j appear in exactly one pair, and, (iii) x^S as x_{ij}, where $i, j \in N$. Thus, the assignment problem is an example of a combinatorial optimisation problem that can be represented as an integer programming formulation.

8.2.2 The knapsack problem

The knapsack problem is another combinatorial optimisation problem that arises in several applications. Consider that we have a collection of n items from which we must make a selection. Each item has associated a cost A_i (e.g., weight) and our selection must be such that the total cost associated with the selection does not exceed a budget B (e.g., weight limit). Each item has also a value C_i associated, and our objective is to find a maximum-valued selection of items such that it does not exceed the budget.

To model that, let us define $x_i = 1$ if item i is selected and 0 otherwise. Let $N = [n]$ be the set of items. Then, the knapsack problem can be represented as the following integer programming problem

$$\begin{aligned}(KP): \max_x \quad & \sum_{i=1}^{n} C_i x_i \\ \text{s.t.:} \quad & \sum_{i=1}^{n} A_i x_i \leq B \\ & x_i \in \{0,1\},\ \forall i \in N.\end{aligned}$$

Notice that the knapsack problem has variants in which an item can be selected a certain number of times, or that multiple knapsacks must be considered simultaneously, both being generalisations of KP.

Naturally, the knapsack problem is also a combinatorial optimisation problem, which can be stated by defining (i) N the set of all items $\{1, \ldots, n\}$, (ii) $S \in \mathcal{F}$ the subset of items with total cost not greater than B, and (iii) x^S as $x_i, \forall i \in N$.

8.2.3 The generalised assignment problem

The generalised assignment problem (or GAP) is a generalisation of the assignment problem including a structure that resembles that of a knapsack problem. In this case, we consider the notion of bins, to each the items have to be assigned. In this case, multiple items can be assigned to a

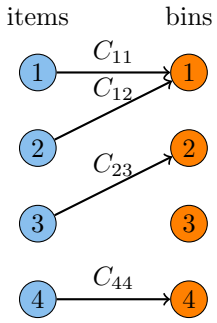


Figure 8.2: An example of a bin packing with total cost $C_{11} + C_{12} + C_{23} + C_{44}$

bin, or a bin might have no item assigned. In some contexts, this problem is also known as the *bin packing problem*.

In this case, we would like to assign all of the m items to n bins, observing that the capacity B of each bin cannot be exceeded by the weights A_i of the items assigned to that bin. We know that assigning the item $i \in [m]$ to the bin $j \in [n]$ costs C_{ij}. Our objective is to obtain a minimum-cost bin assignment (or packing) that does not exceed any of the bin capacities. Figure 8.2 illustrates a possible assignment of items to bins. Notice that the number of total bins does not necessarily need to be the same as the number of items.

To formulate the generalised assignment problem as an integer programming problem, let us define $x_{ij} = 1$ if item i is packed into bin j, and $x_{ij} = 0$ otherwise. Moreover, let M be the set of items and N be the set of bins. Then, the problem can be formulated as follows.

$$
\begin{aligned}
(GAP) : \min_{x} \quad & \sum_{i \in M} \sum_{j \in N} C_{ij} x_{ij} \\
\text{s.t.:} \quad & \sum_{j=1}^{n} x_{ij} = 1, \ \forall i \in M \\
& \sum_{i=1}^{m} A_i x_{ij} \leq B_j, \ \forall j \in N \\
& x_{ij} \in \{0, 1\}, \ \forall i \in M, \forall j \in N.
\end{aligned}
$$

Hopefully, the parallel to the combinatorial optimisation problem version is clear at this point and is left for the reader as a thought exercise.

8.2.4 The set covering problem

Set covering problems are related to the location of infrastructure (or facilities) with the objective of covering demand points and, thus, frequently recurring in settings where service centres such as fire brigades, hospitals, or police stations must be located to efficiently serve locations.

Let M be a set of regions that must be served by opening *service centres*. A centre can be opened at any of the N possible locations. If a centre is opened at location $j \in N$, then it serves (or covers) a subset of regions $S_j \subseteq M$ and has associated opening cost C_j. Our objective is to decide where to open the centres so that all regions are served and the total opening cost is minimised.

Figure 8.3 illustrates an example of a set covering problem based on a fictitious map broken into cells. Each of the cells represents a region that must be covered, i.e., $M = \{1, \ldots, 20\}$. The blue cells represent regions that can have a centre opened, that is, $N = \{3, 4, 7, 11, 12, 14, 19\}$. Notice that $N \subset M$. In this case, we assume that if a centre is opened at a blue cell, then it can serve the respective cell and all adjacent cells. For example, $S_3 = \{1, 2, 3, 8\}$, $S_4 = \{2, 4, 5, 6, 7\}$, and so forth.

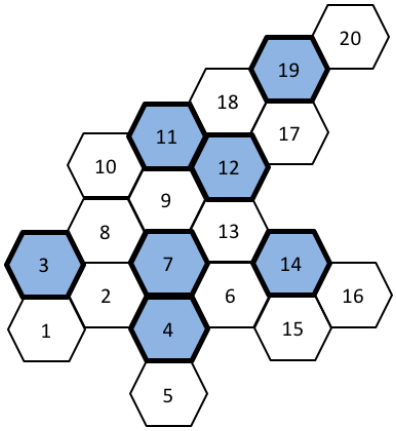


Figure 8.3: The hive map illustrating the set covering problem. Our objective is to cover all of the regions while minimising the total cost incurred by opening the centres at the blue cells

To model the set covering problem, and pretty much any other problem involving indexed subsets such as S_j, $\forall j \in N$, we need an auxiliary parameter matrix A, often referred to as *0-1 incidence matrix*. The matrix A is such that

$$A = \begin{cases} A_{ij} = 1, \text{ if } i \in S_j, \\ A_{ij} = 0, \text{ otherwise.} \end{cases}$$

For example, referring to Figure 8.3, the first column of A would refer to $j = 3$ and would have nonzero values at rows 1, 2, 3, and 8.

We are now ready to pose the set covering problem as an integer programming problem. For that, let $x_j = 1$ if a facility is opened (or a service centre is located) at location j and $x_j = 0$, otherwise. In addition, let M be the set of regions to be served and N be the set of candidate places to have a facility opened. Then, the set covering problem can be formulated as follows.

$$\begin{aligned} (SCP): \min_{x} \quad & \sum_{j \in N} C_j x_j \\ \text{s.t.:} \quad & \sum_{j \in N} A_{ij} x_j \geq 1, \ \forall i \in M \\ & x_j \in \{0, 1\}, \ \forall j \in N. \end{aligned}$$

As a final note, notice that this problem can also be posed as a combinatorial optimisation problem of the form

$$\min_{T \subseteq N} \left\{ \sum_{j \in T} C_j : \bigcup_{j \in T} S_j = M \right\},$$

in which the locations $j \in T$ can be represented as an incidence vector, as before.

8.2.5 Travelling salesperson problem

The travelling salesperson problem (TSP) is one of the most famous combinatorial optimisation problems, perhaps due to its interesting mix of simplicity while being computationally challenging. Assume that we must visit a collection of n cities at most once, and return to our initial point, forming a so-called *tour*. When travelling from a city i to a city j, we incur in the cost C_{ij}, representing, for example, distance or time. Our objective is to minimise the total cost of our tour. Notice that this is equivalent to finding the minimal cost permutation of $n-1$ cities, discarding the city which represents our starting and end point. Figure 8.4 illustrates a collection of cities and one possible tour.

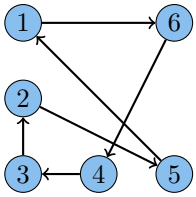


Figure 8.4: An example of a tour between the six cities

To pose the problem as an integer programming model, let us define $x_{ij} = 1$ if city j is visited directly after city i, and $x_{ij} = 0$ otherwise. Let $N = \{1, \ldots, n\}$ be the set of cities. We assume that x_{ii} is not defined for $i \in N$. A naive model for the travelling salesperson problem would be

$$
\begin{aligned}
(TSP): \min_{x} \quad & \sum_{i \in N} \sum_{j \in N} C_{ij} x_{ij} \\
\text{s.t.:} \quad & \sum_{j \in N \setminus \{i\}} x_{ij} = 1, \ \forall i \in N \\
& \sum_{i \in N \setminus \{j\}} x_{ij} = 1, \ \forall j \in N \\
& x_{ij} \in \{0, 1\}, \ \forall i, \forall j \in N : i \neq j
\end{aligned}
$$

First, notice that the proposed formulation of TSP is exactly the same as that of the assignment problem. However, this formulation has an issue. Although it can guarantee that all cities are only visited once, it cannot enforce an important feature of the problem, which is that the tour cannot present disconnections, i.e., contain *sub-tours*. In other words, the salesperson must physically visit from city to city when executing the tour and cannot "teleport" from one city to another. Figure 8.5 illustrates the concept of sub-tours.

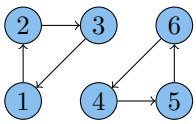


Figure 8.5: A feasible solution for the naive TSP model. Notice the two sub-tours formed

In order to prevent sub-tours, we must include constraints that can enforce the full connectivity of the tour. There are mainly two types of such constraints. The first is called *cut-set constraints* and is defined as

$$
\sum_{i \in S} \sum_{j \in N \setminus S} x_{ij} \geq 1, \ \forall S \subset N, 2 \leq |S| \leq n-1.
$$

The cut-set constraints act by guaranteeing that among any subset of nodes $S \subseteq N$ there is always at least one arc (i, j) connecting one of the nodes in S and a node not in S.

An alternative type of constraint is called *sub-tour elimination* constraint and is of the form

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - 1, \ \forall S \subset N, 2 \leq |S| \leq n - 1.$$

Differently from the cutset constraints, the sub-tour elimination constraints prevent the cardinality of the nodes in each subset from matching the cardinality of arcs within the same subset.

For example, consider the sub-tours illustrated in Figure 8.5 and assume that we would like to prevent the sub-tour formed by $S = \{1, 2, 3\}$. Then the cutset constraint would be of the form

$$x_{14} + x_{24} + x_{34} + x_{15} + x_{25} + x_{35} + x_{16} + x_{26} + x_{36} \geq 1$$

while the sub-tour elimination would be of the form

$$x_{12} + x_{13} + x_{21} + x_{23} + x_{31} + x_{32} \leq 2$$

There are some differences between these two constraints. Typically, cutset constraints are preferred for being stronger (we will discuss the notion of stronger constraints in the next chapters). In any case, either of them suffers from the same problem: the number of such constraints quickly becomes computationally prohibitive as the number of nodes increases. This is because one would have to generate a constraint to each possible node subset combination from sizes 2 to $n - 1$.

A possible remedy to this consists of relying on *delayed constraint generation*. In this case, one can start from the naive formulation TSP and, from the obtained solution, check whether there are any sub-tours formed. That being the case, only the constraints eliminating the observed sub-tours need to be generated, and the problem can be warm-started. This procedure typically terminates far earlier than having all of the possible cutset or sub-tour elimination constraints generated.

8.2.6 Uncapacitated facility location

This is the first mixed-integer programming model we consider. In this, we would like to design a network that can supply clients $i \in M$ by opening (or locating, as it is referred to in this context) facilities among candidate locations $j \in N$. Opening a facility incurs in the fixed cost F_j, and serving a client $i \in M$ from a facility that has been located at $j \in N$ costs C_{ij}. Our objective is to design the most cost-effective production and distribution network. That is, we must decide where to locate facilities and how to serve clients (from these facilities) with minimum total (locating plus service) cost. Figure 8.6a illustrates an example of the problem with $M = \{1, \ldots, 8\}$ and $N = \{1, \ldots, 6\}$ and Figure 8.6b presents one possible configuration with two facilities. The optimal number of facilities located and the client-facility association depends on the trade-offs between locating and service costs.

To formulate the problem as a mixed-integer programming problem, let us define x_{ij} as the fraction of the demand in $i \in M$ being served by a facility located at $j \in N$. In addition, we define the binary variable y_j such that $y_j = 1$, if a facility is located at $j \in N$ and 0, otherwise. With those,

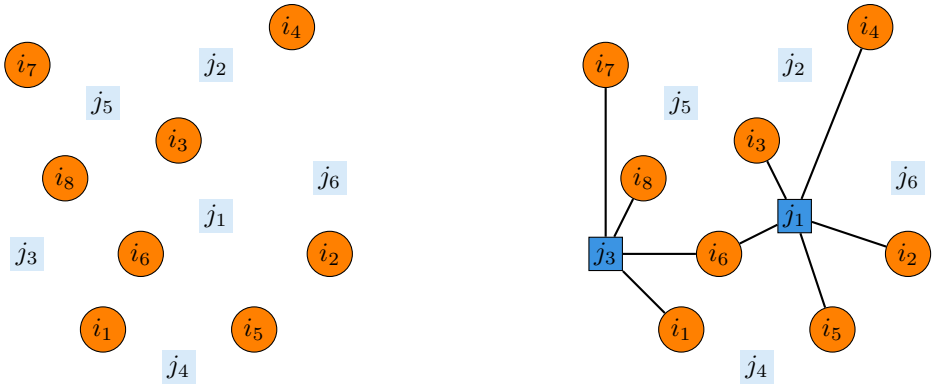


Figure 8.6: An illustration of the facility location problem and one possible solution with two facilities located (right)

the uncapacitated facility location (or UFL) problem can be formulated as

$$(UFL): \min_{x,y} \quad \sum_{j\in N} F_j y_j + \sum_{i\in M}\sum_{j\in N} C_{ij}x_{ij} \tag{8.1}$$

$$\text{s.t.:} \quad \sum_{j\in N} x_{ij} = 1, \ \forall i \in M \tag{8.2}$$

$$\sum_{i\in M} x_{ij} \leq m y_j, \ \forall j \in N \tag{8.3}$$

$$x_{ij} \geq 0, \forall i \in M, \ \forall j \in N \tag{8.4}$$

$$y_j \in \{0,1\}, \ \forall j \in N. \tag{8.5}$$

Some features of this model are worth highlighting. First, notice that the absolute values associated with the demand at nodes $i \in M$ are somewhat implicitly represented in the cost parameter C_{ij}. This is an important modelling feature that allows the formulation to be not only stronger but also more numerically favourable (avoiding large coefficients). Therefore, the demand is thought as being, at each node, 1 or 100%, and $0 \leq x_{ij} \leq 1$ represents the fraction of the demand at $i \in M$ being served by a facility eventually located at $j \in N$. Second, notice how the variable x_{ij} is only allowed to be greater than zero if the variable y_j is set to 1, due to (8.3). Notice that m, the number of clients, is acting as a maximum upper bound for the amount of demand being served from the facility j, which would be at most m when only one facility is located. That constraint is precisely the reason why the problem is called uncapacitated, since, in principle, there are no capacity limitations on how much demand is served from a facility.

Facility location problems are frequent in applications associated with supply chain planning problems and can be specialised to a multitude of settings, including capacitated versions (both nodes and arcs), versions where the arcs must also be located (or built), having multiple echelons, and so forth.

8.2.7 Uncapacitated lot-sizing

Another example of an important mixed-integer programming formulation is the uncapacitated lot-sizing. In this, we would like to plan the production of a single product over a collection of time periods T. We have encountered this problem before in Chapter 1, but now we consider the variation in which the production activity implies in a fixed cost F_t, representing, for example, a setup cost or the need for renting equipment. Once again, for each period $t \in T$, we incur a cost C_t to produce one unit in period t and H_t is paid to store one unit of product from period t to period $t+1$.

Let us define $p_t \geq 0$ be the amount produced in period $t \in T$, and $s_t \geq 0$ as the amount stored at the end of period $t \in T$. In addition, let $y_t \in \{0,1\}$ indicate whether production occurs in period $t \in T$. Also, assume that M is a sufficiently large constraint. Then, the uncapacitated lot-sizing (ULS) can be formulated as

$$\begin{aligned}(ULS): \min_{x,s,y} \ & \sum_{t \in N} (F_t y_t + P_t p_t + H_t s_t) \\ \text{s.t.:} \ & s_{t-1} + p_t = d_t + s_t, \ \forall t \in N \\ & p_t \leq M y_t, \ \forall t \in N \\ & s_t, p_t \geq 0, \ \forall t \in N \\ & y_t \in \{0,1\}, \ \forall t \in N.\end{aligned} \tag{8.6}$$

Notice that the formulation of ULS is very similar to that seen in Chapter 1, with exception of the variable y_t, its associated fixed cost term $\sum_{t \in T} F_j y_j$ and the constraint (8.6). This constraint is precisely what renders the "uncapacitated" nomenclature, and is commonly known in the context of mixed-integer programming as *big-M constraints*. Notice that the constant M is playing the role of $+\infty$: it only really makes the constraint relevant when $y_t = 0$, so that $p_t \leq 0$ is enforced, making thus $p_t = 0$. However, this interpretation has to be taken carefully. Big-M constraints are known for being the cause of numerical issues and worsening the performance of mixed-integer programming solver methods. Thus, the value of M must be set such that it is the smallest value possible such that it does not artificially create a constraint. Finding these values are often challenging and instance dependent. In the capacitated case, M can trivially take the value of the production capacity.

8.3 Good formulations

In this section, we discuss what makes a formulation a better formulation for a (mixed-)integer programming (MIP) problem. Just like it is the case with any mathematical programming application, there are potentially infinite possible ways to formulate the same problem. While in the case of linear programming (i.e., only continuous variables), alternative formulations typically do not lead to significant differences in terms of computational performance, the same is definitely not true in the context of MIP. In fact, whether a MIP problem can be solved in a reasonable computational time often depend on having a good, or *strong*, formulation.

Therefore, we must be able to recognise, from a set of equivalent formulations for the same problem, that which yields better computational performance. But first, we must be able to understand the source of these differences in computational performance. For that, the first thing to realise is that solution methods for MIP models rely on *successively solving linear programming models* called *linear programming (LP) relaxations*. How exactly this happens will be the subject of our

next chapters. But for now, one can infer that the best formulation will be the one that requires the solution of the least of such LP relaxations. And precisely, it so turns out that the number of LP relaxations that need to be solved (and, hence, performance) is strongly dependent on how closely the formulation resembles the *convex hull* of the feasible solutions.

An LP relaxation simply consists of a version of the original MIP problem in which the integrality requirements are dropped. Most of the methods used to solve MIP models are based on LP relaxation. There are several reasons why the employment of LP relaxations is a good strategy. First, we can solve (and resolve) LP problems efficiently. Secondly, the solution of the LP relaxation can be used to *reduce* the search space of the original MIP. However, simply rounding the solution of the LP relaxation will typically not lead to relevant solutions.

Let us illustrate the geometry of an integer programming model, such that the points we were discussing become more evident. Consider the problem

$$\begin{aligned}(P): \max_{x}.\ & x_1 + \frac{16}{25}x_2 \\ \text{s.t.:}\ & 50x_1 + 31x_2 \leq 250 \\ & 3x_1 + 31x_2 \geq -4 \\ & x_1, x_2 \in \mathbb{Z}_+.\end{aligned}$$

The feasible region of problem P is represented in Figure 8.7. First, notice how in this case the feasible region is not a polyhedral set anymore, but yet a collection of discrete points (represented in blue) that happen to be within the polyhedral set formed the linear constraints. This is one of the main complicating features of MIPs because the premise of convexity does not hold anymore.

Figure 8.7 also illustrates how rounding the solution obtained from the LP relaxation would, as it is often the case with MIPs, lead to infeasible solutions, except when x_1 is rounded up and x_2 rounded down, which leads to the suboptimal solution $(2, 5)$. However, one can still graphically find the optimal solution using the same procedure as that employed for linear programming problems, which would lead to the optimal integer solution $(5, 0)$.

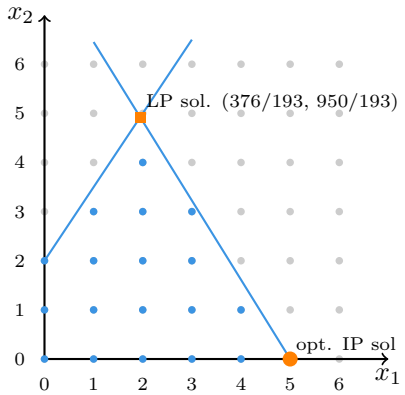


Figure 8.7: Graphical representation of the feasible region of the example

8.3.1 Comparing formulations

In order to be able to compare formulations, we require some specific definitions, including a precise definition of what is a *formulation*.

Definition 8.1. *A polyhedral set $P = \{x \in \mathbb{R}^{n+p} : Ax \leq b\}$ is a formulation for a set $X \subseteq \mathbb{Z}^n \times \mathbb{R}^p$ if and only if $X = P \cap (\mathbb{Z}^n \times \mathbb{R}^p)$.*

From Definition 8.1, we can conclude that two formulations P_1 and P_2 are *equivalent* if $P_1 \cap (\mathbb{Z}^n \times \mathbb{R}^p) = P_2 \cap (\mathbb{Z}^n \times \mathbb{R}^p)$. Moreover, from Definition 8.1, it becomes clear that the feasible region of an integer programming problem is a collection of points represented by X. This is illustrated in Figure 8.8, where one can see three alternative equivalent formulations, P_1, P_2, and P_3 for the same set X.

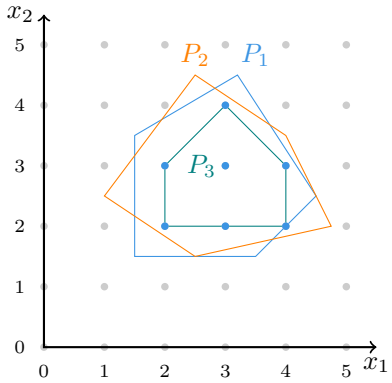


Figure 8.8: An illustration of three alternative formulations for X. Notice that P_3 is an ideal formulation, representing the convex hull of X.

Formulation P_3 has a special feature associated with it. Notice how all extreme points of P_3 belong to X. This has an important consequence. That implies that the solution of the original integer programming problem can be obtained by solving a single LP relaxation since the solution of both problems is the same. This is precisely what characterises an *ideal* formulation, which is that leading to a minimal (i.e., only one) number of required LP relaxation solutions as solving an LP relaxation over an ideal P yields a solution $x \in X$ for any cost vector c. This will only be the case if the formulation P is the *convex hull* of X.

This is the case because of two important properties relating to the set X and its convex hull $\mathbf{conv}(X)$. The first is that $\mathbf{conv}(X)$ is a polyhedral set, and the second is that the extreme points of $\mathbf{conv}(X)$ belong to X. We summarise those two facts in Proposition 8.2, to which we will refer shortly. Notice that the proof for the proposition can be derived from Definition 2.7 and Theorem 2.8, and is left as an exercise.

Proposition 8.2. $\mathbf{conv}(X)$ *is a polyhedral set and all its extreme points belong to X.*

If P is such that $P = \mathbf{conv}(X)$, the original problem

$$\text{min. } \left\{c^\top x : x \in X\right\},$$

can, in principle, be replaced with

$$\text{min. } \left\{c^\top x : x \in \mathbf{conv}(X)\right\},$$

where we have that

$$X = \{Ax \leq b, x \in \mathbb{Z}^n \times \mathbb{R}^p\} \text{ and } \mathbf{conv}(X) = \left\{Ax \leq b, x \in \mathbb{R}^{n+p}_+\right\}.$$

Unfortunately, this is often not the case. Typically, except for some specific cases, a description of $\mathbf{conv}(X)$ is not known and deriving it algorithmically is an impractical computational task. However, Proposition 8.2 allows us to define a structured way to compare formulations. This is summarised in Definition 8.3.

Definition 8.3. *Given a set $X \subseteq \mathbb{Z}^n \times \mathbb{R}^n$ and two formulations P_1 and P_2 for X, P_1 is a* better formulation *than P_2 if $P_1 \subset P_2$.*

Definition 8.3 gives us a framework to try to demonstrate that a given formulation is better than another. If we can show that $P_1 \subset P_2$, then, by definition (literally), P_1 is a better formulation than P_2. Clearly, this is not a perfect framework, since, for example, it would not be useful for comparing P_1 and P_2 in Figure 8.8, and, in fact, there is nothing that can be said a priori about the two in terms of which will render the better performance. Often, in the context of MIP, this sort of analysis can only rely on careful computational experimentation.

A final point to make is that sometimes one must compare formulations of distinct dimensions, that is, with a different number of variables. When that is the case, one can resort to projection, as a means to compare both formulations onto the same space of variables.

8.4 Exercises

Exercise 8.1: Uncapacitated lot sizing (ULS) formulations

Consider the following formulations $P^{(x,s,y)}_{\text{ULS-1}}$ and $P^{(w,y)}_{\text{ULS-2}}$ as linear (i.e., continuous) relaxations for the ULS problem defined over $N = \{1, \dots, n\}$ periods:

$$P^{(x,s,y)}_{\text{ULS-1}} = \left\{ \begin{array}{lll} (x,s,y): & s_{t-1} + x_t = d_t + s_t, & \forall t \in N \\ & x_t \le M y_t, & \forall t \in N \\ & s_0 = 0, & \\ & s_t \ge 0, & \forall t \in N \\ & x_t \ge 0, & \forall t \in N \\ & 0 \le y_t \le 1, & \forall t \in N \end{array} \right\} \quad \left| \begin{array}{ll} x_t & \text{production in period } t \\ s_t & \text{stock in period } t \\ y_t & \text{setup in period } t \\ d_t & \text{demand in period } t \\ M & \text{maximum production} \\ & M = \sum_{t \in N} d_t \end{array} \right.$$

$$P^{(w,y)}_{\text{ULS-2}} = \left\{ \begin{array}{lll} (w,y): & \sum_{i=1}^{t} w_{it} = d_t, & \forall t \in N \\ & w_{it} \le d_t y_i, & \forall i,t \in N : i \le t \\ & w_{it} \ge 0, & \forall i,t \in N : i \le t \\ & 0 \le y_t \le 1, & \forall t \in N \end{array} \right\} \quad \left| \begin{array}{ll} w_{it} & \text{production in period } i \\ & \text{to be used in period } t \\ y_t & \text{setup in period } t \end{array} \right.$$

(a) Use projection to show that $P^{(w,y)}_{\text{ULS-2}}$ is stronger than $P^{(x,s,y)}_{\text{ULS-1}}$, i.e, $P^{(w,y)}_{\text{ULS-2}} \subset P^{(x,s,y)}_{\text{ULS-1}}$.

Hint: First, construct an extended formulation $P^{(x,s,y,w)}_{\text{ULS-2}}$ by writing the variables x_t and s_t in terms of variables w_{it} and add them to $P^{(w,y)}_{\text{ULS-2}}$. Then, use a projection to show that

$$\underset{x,s,y}{\mathbf{proj}} \, (P^{(x,s,y,w)}_{\text{ULS-2}}) \subseteq P^{(x,s,y)}_{\text{ULS-1}},$$

which is equivalent to $P^{(w,y)}_{\text{ULS-2}} \subseteq P^{(x,s,y)}_{\text{ULS-1}}$. Do this by verifying that each constraint of $P^{(x,s,y)}_{\text{ULS-1}}$ is satisfied by all $(x,s,y) \in P^{(x,s,y,w)}_{\text{ULS-2}}$. Finally, show that a solution $(\bar{x}, \bar{s}, \bar{y})$ with $\bar{x}_t = d_t$ and $\bar{y}_t = d_t/M$, for all $t \in N$, satisfies $(\bar{x}, \bar{s}, \bar{y}) \in P^{(x,s,y)}_{\text{ULS-1}} \setminus P^{(x,s,y,w)}_{\text{ULS-2}}$.

(b) The optimisation problems associated with the two ULS formulations are

$$\text{(ULS-1)} \quad \begin{array}{lll} \min\limits_{x,s,y} & \sum\limits_{t \in N} (f_t y_t + p_t x_t + q_t s_t) & \\ \text{s.t.:} & s_{t-1} + x_t = d_t + s_t, & \forall t \in N \\ & x_t \le M y_t, & \forall t \in N \\ & s_0 = 0, & \\ & s_t \ge 0, & \forall t \in N \\ & x_t \ge 0, & \forall t \in N \\ & 0 \le y_t \le 1, & \forall t \in N \end{array} \quad \left| \begin{array}{ll} x_t & \text{production in period } t \\ s_t & \text{stock in period } t \\ y_t & \text{setup in period } t \\ d_t & \text{demand in period } t \\ M & \text{maximum production} \\ & M = \sum_{t \in N} d_t \end{array} \right.$$

$$
\begin{array}{lll|ll}
\text{(ULS-2)} & \min\limits_{w,y} & \sum\limits_{t\in N}\left[f_t y_t + p_t \sum\limits_{i=t}^{n} w_{ti} + q_t \sum\limits_{i=1}^{t}\left(\sum\limits_{j=i}^{n} w_{ij} - d_i\right)\right] & w_{it} & \text{production in period } i \\
& \text{s.t.:} & \sum\limits_{i=1}^{t} w_{it} = d_t, \qquad \forall t \in N & & \text{to be used in period } t \\
& & w_{it} \le d_t y_i, \qquad \forall i,t \in N : i \le t & y_t & \text{setup in period } t \\
& & w_{it} \ge 0, \qquad \forall i,t \in N : i \le t & & \\
& & 0 \le y_t \le 1, \qquad \forall t \in N & &
\end{array}
$$

Consider a ULS problem instance over $N = \{1, \ldots, 6\}$ periods with demands $d = (6, 7, 4, 6, 3, 8)$, set-up costs $f = (12, 15, 30, 23, 19, 45)$, unit production costs $p = (3, 4, 3, 4, 4, 5)$, unit storage costs $q = (1, 1, 1, 1, 1, 1)$, and maximum production capacity $M = \sum_{i=1}^{6} d_j = 34$. Solve the problems ULS-1 and ULS-2 with Julia using JuMP to verify the result of part (a) computationally.

Exercise 8.2: TSP formulation - MTZ

Show that the following formulation P_{MTZ} is valid for the TSP defined on a directed graph $G = (N, A)$ with $N = \{1, \ldots, n\}$ cities and arcs $A = \{(i, j) : i, j \in N, i \ne j\}$ between cities.

$$
P_{MTZ} = \begin{cases}
\sum\limits_{j\in N\setminus\{i\}} x_{ij} = 1, & \forall i \in N \\
\sum\limits_{j\in N\setminus\{i\}} x_{ji} = 1, & \forall i \in N \\
u_i - u_j + (n-1)x_{ij} \le n-2, & \forall i,j \in N \setminus \{1\} : i \ne j \quad (*) \\
x_{ij} \in \{0,1\}, & \forall i,j \in N : i \ne j
\end{cases}
$$

where $x_{ij} = 1$ if city $j \in N$ is visited immediately after city $i \in N$, and $x_{ij} = 0$ otherwise. Constraints $(*)$ with the variables $u_i \in \mathbb{R}$ for all $i \in N$ are called *Miller-Tucker-Zemlin* (MTZ) subtour elimination constraints.

Hint: Formulation P_{MTZ} is otherwise similar to the formulation presented, except for the constraints $(*)$ which replace either the cutset constraints

$$\sum_{i\in S}\sum_{j\in N\setminus S} x_{ij} \ge 1, \ \forall S \subset N, S \ne \emptyset,$$

or the subtour elimination constraints

$$\sum_{i\in S}\sum_{j\in S} x_{ij} \le |S| - 1, \ \forall S \subset N, 2 \le |S| \le n-1,$$

which are used to prevent subtours in TSP solutions. Thus, you have to show that:

(1) Constraints $(*)$ prevent subtours in any solution $x \in P_{MTZ}$.

(2) Every TSP solution x (on the same graph G) satisfies the constraints $(*)$.

You can prove (1) by contradiction. First, assume that a solution $x \in P_{MTZ}$ has a subtour with k arcs $(i_1, i_2), \ldots, (i_{k-1}, i_k), (i_k, i_1)$ and k nodes $\{i_1, \ldots, i_k\} \in N \setminus \{1\}$. Then, write the constraints $(*)$ for all arcs in this subtour and try to come up with a contradiction.

You can prove (2) by finding suitable values for each $u_2, \ldots, u_n$ that will satisfy the constraints $(*)$ for any TSP solution x. Recall that a TSP solution represents a *tour* that visits each of the $N = \{1, \ldots, n\}$ cities exactly once and returns to the starting city.

Exercise 8.3: TSP implementation

Solve a 15 node TSP instance using the formulation P_{MTZ} presented in Exercise 8.1. You can randomly generate city coordinates in xy-plane for all $N = \{1, \ldots, n\}$ cities. Letting c_{ij} denote the distance between cities i and j, the problem MTZ can be formulated as

$$
\begin{aligned}
(MTZ): \min_{x,u} \quad & \sum_{i \in N} \sum_{j \in N} c_{ij} x_{ij} & & \\
\text{s.t.:} \quad & \sum_{j \in N \setminus \{i\}} x_{ij} = 1, & & \forall i \in N, \\
& \sum_{j \in N \setminus \{i\}} x_{ji} = 1, & & \forall i \in N, \\
& u_i - u_j + (n-1) x_{ij} \leq n - 2, & & \forall i, j \in N \setminus \{1\} : i \neq j, \\
& x_{ij} \in \{0, 1\}, & & \forall i, j \in N : i \neq j.
\end{aligned}
$$

Implement the model and solve the problem instance with Julia using JuMP.

Exercise 8.4: TSP formulation - tightening the MTZ formulation

Recall the MTZ formulation for the Travelling Salesperson Problem (TSP) presented in Exercise 8.2.

(a) Show that the inequalities (8.7) - (8.10) are valid for the TSP problem (i.e., they hold for any feasible solution of the TSP problem), assuming that $n > 2$:

$$x_{ij} + x_{ji} \leq 1, \qquad \forall i, j \in N : i \neq j \tag{8.7}$$

$$u_i - u_j + (n-1) x_{ij} + (n-3) x_{ji} \leq n - 2, \qquad \forall i, j \in N \setminus \{1\} : i \neq j \tag{8.8}$$

$$u_j - 1 + (n-2) x_{1j} \leq n - 1 \qquad \forall j \in N \setminus \{1\} \tag{8.9}$$

$$1 - u_i + (n-1) x_{i1} \leq 0 \qquad \forall i \in N \setminus \{1\} \tag{8.10}$$

Hint: You can assume that the variables $u_2, \ldots, u_n$ take unique integer values from the set $\{2, \ldots, n\}$. That is, we have $u_i \in \{2, \ldots, n\}$ for all $i = 2, \ldots, n$ with $u_i \neq u_j$ for all $i, j \in 2, \ldots, n$. This holds for any TSP solution of problem MTZ as we showed in Exercise 8.2. If we fix city 1 as the starting city, then the value of each u_i represents the position of city i in the TSP tour, i.e., $u_i = t$ for $t = 2, \ldots, n$ if city $i \neq 1$ is the t:th city in the tour. You have to check that each of the inequalities (8.7) - (8.10) hold (individually) for any arc $(i, j) \in A$ and city $i \in N$ that are part of the inequality, by checking that the following two cases are satisfied: either $x_{ij} = 0$ or $x_{ij} = 1$.

(b) Add all four sets of inequalities (8.7) - (8.10) to the MTZ formulation and compare the computational performance against the model with no extra inequalities.

Exercise 8.5: Scheduling problem

A set of $N = \{1, \ldots, n\}$ jobs must be carried out on a single machine that can do only one job at a time. Each job $j \in N$ takes p_j hours to complete. Given job weights w_j for all $j \in N$, in what order should the jobs be carried out so as to minimise the weighted sum of their start times? Formulate this scheduling problem as a mixed integer-programming problem.

Exercise 8.6: Piecewise linear objective functions and logical constraints

Recall the paint factory problem

$$\begin{aligned} \text{max. } & z = 5x_1 + 4x_2 && (8.11)\\ \text{s.t.: } & 6x_1 + 4x_2 \leq 24 && (8.12)\\ & x_1 + 2x_2 \leq 6 && (8.13)\\ & x_2 - x_1 \leq 1 && (8.14)\\ & x_2 \leq 2 && (8.15)\\ & x_1, x_2 \geq 0, && (8.16) \end{aligned}$$

where x_1 representes the amount (in tons) of exterior paint produced, and x_2 the amount of interior paint produced. This is a simple linear problem that has been used as an illustrative example throughout the lecture notes.

To demonstrate some typical structures in integer programming models, we now add two modifications to the problem, namely

- Piecewise linear functions: the profit for exterior paint depends on the amount produced. The profit for the first ton produced is 7 ($1000/ton), while the next ton can only be sold at a profit of 5, the third for a profit of 3 and the fourth ton only makes a profit of 1.
- Logical constraints: the factory has to produce at least 1.5 tons of at least one of the two paint types, that is, either $x_1 \geq 1.5$ or $x_2 \geq 1.5$ must hold.

As seen in Fig. 8.9, these modifications result in a nonlinear objective function and a nonconvex feasible region. Using additional integer/binary variables, formulate and solve the problem as a mixed-integer linear problem.

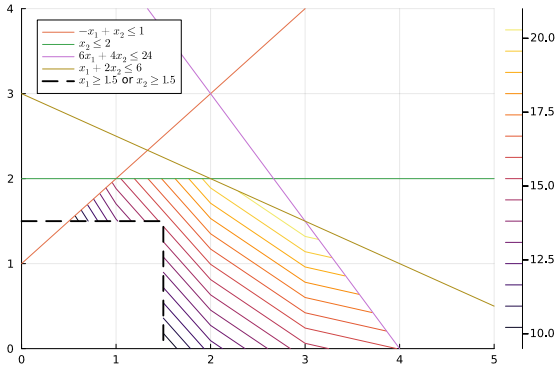


Figure 8.9: The feasible region of the problem, and the contours of the objective function

Chapter 9

Branch-and-bound Method

9.1 Optimality for integer programming problems

We will now discuss a method to solve mixed-integer programming problems that rely on the successive solution of linear programming relaxations. Although there are several methods that can be employed to solve combinatorial optimisation problems, often they are not capable of providing optimality guarantees for the solution obtained (e.g., are heuristics or metaheuristics) or do not exploit the availability of a (linear) mathematical programming formulation. To date, the most widespread method capable of both is generally known as a *branch-and-cut* method.

Branch-and-cut methods are composed of a combination of multiple parts, including, among other techniques, a branch-and-bound coordination scheme, cutting planes, and heuristics as well. In the next chapters, we will focus on each of these parts individually, starting with the *branch-and-bound* method.

9.2 Relaxations

Before we present the method itself, let us discuss the more general concept of *relaxation*. We have visited the concept somewhat informally before, but now we will concentrate on a more concrete definition.

Consider an integer programming problem of the form

$$z = \min_{x} \left\{ c^\top x : x \in X \subseteq \mathbb{Z}^n \right\}.$$

To prove that a given solution x^* is optimal, we must rely on the notion of *bounding*. That is, we must provide a pair of upper and lower bounds that are as close (or tight) as possible. If it happens that these bounds have the same value, and thus match the value of $z = c^\top x^*$, we have available a *certificate* of optimality for x^*. This concept must sound familiar to you. We already used similar arguments in Chapter 5, when we introduced the notion of dual bounds.

Most methods that can prove optimality work by bounding the optimal solution. In this context, bounding means to construct an increasing sequence of lower bounds

$$\underline{z}_1 < \underline{z}_2 < \cdots < \underline{z}_s \leq z$$

and a decreasing sequence of upper bounds

$$\overline{z}_1 > \overline{z}_2 > \cdots > \overline{z}_t \geq z$$

to obtain as tight as possible lower ($\underline{z} \leq z$) and upper ($\overline{z} \geq z$) bounds. Notice that the process can be arbitrarily stopped when $\overline{z}_t - \underline{z}_s \leq \epsilon$, where s and t are some positive integers and $\epsilon > 0$ is a

predefined (suitably small) tolerance. The term ϵ represents an *absolute optimality gap*, meaning that one can guarantee that the optimal value is at most greater than $\underline{z}$ by ϵ units and at most smaller than $\overline{z}$ by ϵ units. In other words, the optimal value must be either $\underline{z}$, $\overline{z}$, or a value in between.

This framework immediately poses the challenge of deriving such bounds efficiently. It turns out that this is a challenge that goes beyond the context of mixed-integer programming problems. In fact, we will see this idea of bounding in Chapter 12 when we discuss decomposition methods, which also generate lower and upper bounds during their execution.

Regardless of the context, bounds are typically of two types: *primal* bounds, which are bounds obtained by evaluating a *feasible* solution (i.e., that satisfy primal feasibility conditions); and *dual* bounds, which are typically attained when primal feasibility is allowed to be violated so that a dual feasible solution is obtained. In the context of minimisation, primal bounds are upper bounds (to be minimised), while dual bounds are lower bounds (to be maximised). Clearly, in the case of maximisation, the reverse holds.

Primal bounds can be obtained by means of a feasible solution. For example, one can heuristically assemble a solution that is feasible by construction. On the other hand, dual bounds are typically obtained by means of solving a *relaxation* of the original problem. We are ready now to provide Definition 9.1, which formally states the notion of relaxation.

Definition 9.1 (Relaxation)**.** *Problem*

$$(RP) : z_{RP} = min.\ \left\{\overline{c}^\top x : x \in \overline{X} \subseteq \mathbb{R}^n\right\}$$

is a relaxation of problem

$$(P) : z = min.\ \left\{c^\top x : x \in X \subseteq \mathbb{R}^n\right\}$$

if $X \subseteq \overline{X}$*, and* $\overline{c}^\top x \leq c^\top x$*,* $\forall x \in X$*.*

Definition 9.1 provides an interesting insight related to relaxations: they typically comprise an expansion of the feasible region, possibly combined with an objective function bounding. Thus, two main strategies to obtain relaxations are to enlarge the feasible set by dropping constraints and replacing the objective function with another of the same or smaller value. One might notice at this point that we have used a very similar argumentation to define linear programming duals in Chapter 5.

Clearly, for relaxations to be useful in the context of solving mixed-integer programming problems, they have to be easier to solve than the original problem. That being the case, we can then rely on two important properties that relaxations have, which are crucial for using them as a means to generate dual bounds. These are summarised in Proposition 9.2 and 9.3.

Proposition 9.2. *If RP is a relaxation of P, then* z_{RP} *is a dual bound for* z*.*

Proof. Let x^*_{RP} be the optimal solution for RP. For any optimal solution x^* of P, we have that $x^* \in X \subseteq \overline{X}$, which implies that $x^* \in \overline{X}$. Thus, we have that

$$z = c^\top x^* \geq \overline{c}^\top x^* \geq \overline{c}^\top x^*_{RP} = z_{RP}. \qquad \square$$

The first inequality is due to Definition 9.1 and the second is because x^* is simply a feasible solution, but not necessarily the optimal, for z_{RP}.

Proposition 9.3. *The following statements are true:*

1. *If a relaxation RP is infeasible, then P is infeasible.*

2. *Let x^* be an optimal solution for RP. If $x^* \in X$ and $\overline{c}^\top x^* = c^\top x^*$, then x^* is an optimal solution for P.*

Proof. To prove (1), simply notice that if $\overline{X} = \emptyset$, then $X = \emptyset$. Now, let x^* be the optimal solution for RP. To show (2), notice that as $x^* \in X$, $z \leq c^\top x^* = \overline{c}^\top x^*_{RP} = z_{RP}$. From Proposition 9.2, we have $z \geq z_{RP}$. Thus, $z = z_{RP}$. □

9.2.1 Linear programming relaxation

In the context of solving (mixed-)integer programming problems, we will rely on the notion of linear programming (LP) relaxations. We have briefly discussed the idea in the previous chapter, but, for the sake of precision, let us first define what we mean by the term.

Definition 9.4 (Linear programming (LP) relaxation)**.** *The LP relaxation of an integer programming problem min. $\left\{c^\top x : x \in P \cap \mathbb{Z}^n\right\}$ with $P = \left\{x \in \mathbb{R}^n_+ : Ax \leq b\right\}$ is the linear programming problem min. $\left\{c^\top x : x \in P\right\}$.*

Notice that an LP relaxation is indeed a relaxation, since we are enlarging the feasible region by dropping the integrality requirements while maintaining the same objective function (cf. Definition 9.1).

Let us consider a numerical example. Consider the integer programming problem

$$\begin{aligned} z = \max_x.\ & 4x_1 - x_2 \\ \text{s.t.:}\ & 7x_1 - 2x_2 \leq 14 \\ & x_2 \leq 3 \\ & 2x_1 - 2x_2 \leq 3 \\ & x \in \mathbb{Z}^2_+. \end{aligned}$$

A dual (upper; notice the maximisation) bound for z can be obtained by solving its LP relaxation, which yields the bound $z_{LP} = 8.42 \geq z$. A primal (lower) bound can be obtained by choosing any of the feasible solutions (e.g., $(2, 1)$, to which $z = 7 \leq z$). This is illustrated in Figure 9.1.

We can now briefly return to the discussion about better (or stronger) formulations for integer programming problems. Stronger formulations are characterised by those that wield stronger relaxations, or, specifically, that yield relaxations that are guaranteed to provide better (or tighter) dual bounds. This is formalised in Proposition 9.5.

Proposition 9.5. *Let P_1 and P_2 be formulations of the integer programming problem*

$$\min_x.\ \left\{c^\top x : x \in X\right\} \text{ with } X = P_1 \cap \mathbb{Z}^n = P_2 \cap \mathbb{Z}^n.$$

Assume P_1 is a better formulation than P_2 (i.e., $P_1 \subset P_2$). Let $z^i_{LP} = \min. \left\{c^\top x : x \in P_i\right\}$ for $i = 1, 2$. Then $z^1_{LP} \geq z^2_{LP}$ for any cost vector c.

Proof. Apply Proposition 9.2 by noticing that P_2 is a relaxation of P_1. □

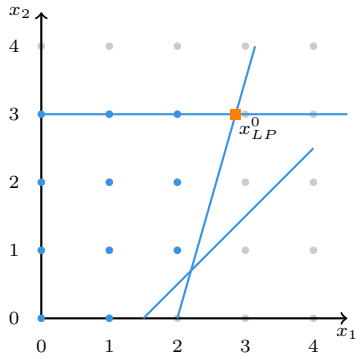


Figure 9.1: The feasible region of the example (represented by the blue dots) and the solution of the LP relaxation, with objective function value $z_{LP} = 8.42$

9.2.2 Relaxation for combinatorial optimisation

There is another important type of relaxation that is often exploited in the context of combinatorial optimisation. Specifically, a relaxation for a combinatorial optimisation problem that is also a combinatorial optimisation problem is called a *combinatorial relaxation*.

Efficient algorithms are known for some combinatorial optimisation problems, and this can be exploited in a solution method for a problem to which the combinatorial relaxation happens to be one of such problems.

Let us illustrate the concept with a couple of examples. Consider the travelling salesperson problem (TSP). Recall that, without considering the tour elimination constraints, we recover the assignment problem. It so turns out that the assignment problem can be solved efficiently (for example, using the Hungarian method) and thus can be used as a relaxation for the TSP, i.e.,

$$z_{TSP} = \min_{T \subseteq A} \left\{ \sum_{(i,j) \in T} c_{ij} : T \text{ forms a tour} \right\} \geq$$
$$z_{AP} = \min_{T \subseteq A} \left\{ \sum_{(i,j) \in T} c_{ij} : T \text{ forms an assignment} \right\}.$$

Still relating to the TSP, one can obtain even stronger combinatorial relaxations using *1-trees* for symmetric TSP. Let us first define some elements. Consider an undirected graph $G = (V, E)$ with edge (or arcs) weights c_e for $e \in E$. The objective is to find a minimum weight tour.

Now, notice the following: (i) a tour contains exactly two edges adjacent to the origin node (say, node 1) and a path through nodes $\{2, \ldots, |V|\}$; (ii) a tour is a special case of a *(spanning) tree*, which is any subset of edges that covers (or touch at least once) all nodes $v \in V$.

We can now define what is a 1-tree. A *1-tree* is a subgraph consisting of two edges adjacent to node 1 plus the edges of a tree on nodes $\{2, \ldots, |V|\}$. Clearly, every tour is a 1-tree with the additional requirement (or constraint) that every node has exactly two incident edges. Thus, the problem of finding minimal 1-trees is a relaxation for the problem of finding optimal tours. Figure 9.2 illustrates a 1-tree for an instance with eight nodes.

Once again, it so turns out that several efficient algorithms are known for forming minimal spanning

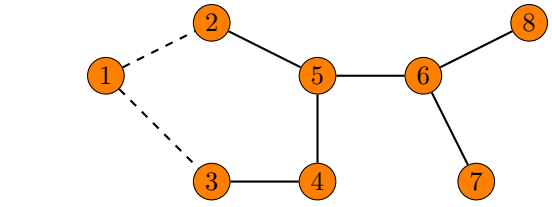


Figure 9.2: An example of a 1-tree considering eight nodes

trees, which can be efficiently utilised as a relaxation for the symmetric TSP, that is

$$z_{STSP} = \min_{T \subseteq E} \left\{ \sum_{e \in T} c_e : T \text{ forms a tour} \right\} \geq$$

$$z_{1-TREE} = \min_{T \subseteq E} \left\{ \sum_{e \in T} c_e : T \text{ forms a 1-tree} \right\}.$$

9.3 Branch-and-bound method

Relaxations play a major role in solving mixed-integer programming and/or combinatorial optimisation problems. However, they are only part of the framework (specifically, the bounding part of the branch-and-bound method). One still needs to be able to, from the solution of said relaxations, be able to construct a solution to the original problem.

Branch-and-bound is an algorithmic strategy that is far broader than mathematical programming and optimisation. In essence, it consists of a *divide-and-conquer* strategy, in which we first break an original problem into smaller and manageable (or solvable) parts and then recombine the solution of these parts into a solution for the original problem.

Specifically, let

$$(P) : z = \max_{x} \left\{ c^\top x : x \in S \right\}.$$

The working principle behind this strategy is based on the notion of decomposition, which is formalised in Proposition 9.6.

Proposition 9.6. *Let* $K = \{1, \ldots, |K|\}$ *and* $\bigcup_{k \in K} S_k = S$ *be a decomposition of* S*. Let* $z^k = \max_x \left\{ c^\top x : x \in S_k \right\}, \forall k \in K$*. Then*

$$z = \max_{k \in K} \left\{ z^k \right\}.$$

Now, one challenging aspect related to divide-and-conquer approaches is that, in order to be able to find a solution, one might need to repeat several times the strategy based on Proposition 9.6, which suggests breaking any given problem in multiple subproblems. This leads to a multi-layered collection of subproblems that must have their relationships managed. To address this issue, such methods typically rely on tree structures called *enumerative trees*, which are simply a representation that allows for keeping track of the relationship (represented by branches) between subproblems (represented by nodes).

Figure 9.3 represents an enumerative tree for a generic problem $S \subseteq \{0, 1\}^3$ in which one must define the value of a three-dimensional binary variable. The subproblems are formed by, at each

level, fixing one of the components to zero or one, forming then two subproblems. Any strategy to form subproblems that generate two subproblems (or children) is called a *binary branching*.

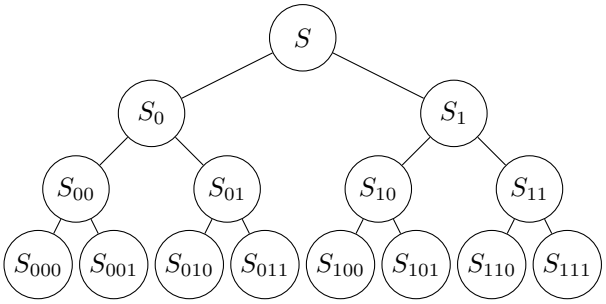


Figure 9.3: A enumeration tree using binary branching for a problem with three binary variables

Specifically, notice that, at the highest level, we have

$$S = S_0 \cup S_1 = \{x \in S : x_1 = 0\} \cup \{x \in S : x_1 = 1\},$$

which renders two subproblems. Then, each subproblem is again decomposed each into two children, such that

$$S_i = S_{i0} \cup S_{i1} = \{x \in S : x_1 = i, x_2 = 0\} \cup \{x \in S : x_1 = i, x_2 = 1\}.$$

Finally, once all of the variables are fixed, we arrive at what is called the *leaves* of the tree. These are such that they cannot be further divided, since they immediately yield a candidate solution for the original problem.

$$S_{ij} = S_{ij0} \cup S_{ij1} = \{x \in S : x_1 = i, x_2 = j, x_3 = 0\} \cup \{x \in S : x_1 = i, x_2 = j, x_3 = 1\}$$

Notice that applying Proposition 9.6, we can recover an optimal solution to the problem.

9.3.1 Bounding in enumerative trees

As you may suspect, the strategy of enumerating all possible solutions will quickly become computationally intractable and will most likely not be feasible for mixed-integer programming problems, or any relevant combinatorial optimisation for that matter. That is precisely when the notion of bounding comes to the spotlight: by possessing bound information on our original problem, we might be able to dismiss branches (or prune, in keeping with our tree analogy) from being searched, and hopefully find a solution without the need to exhaustively explore the enumeration tree.

The main principle behind the pruning of branches in enumerative search trees is summarised in Proposition 9.7.

Proposition 9.7. *Consider the problem P and let $S = \bigcup_{k \in K} S_k$ be a decomposition of S into smaller sets. Let $z^k = max._x \left\{c^\top x : x \in S_k\right\}$ for $k \in K$, and let $\overline{z}^k$ ($\underline{z}^k$) be an upper (lower) bound on z^k. Then $\overline{z} = \max_{k \in K} \left\{\overline{z}^k\right\}$ and $\underline{z} = \max_{k \in K} \left\{\underline{z}^k\right\}$.*

First, notice that P is a maximisation problem, for which an upper bound is a dual bound obtained from a relaxation, and a lower bound is a primal bound obtained from a feasible solution. Proposition 9.7 states that the best known primal (lower) bound can be applied *globally* to all of

the subproblems S_k, $\forall k \in K$. On the other hand, dual (upper) bounds can only be considered valid locally, since only the worst of the upper bounds can be guaranteed to hold globally.

Pruning branches is made possible by combining relaxations and global primal bounds. If, at any moment of the search, the solution of a relaxation of S_k is observed to be *worse* than a known global primal bound, then any further branching from that point onwards would be fruitless, since no solution found from that subproblem could be better than the relaxation for S_k. Specifically, we have that

$$\underline{z} \geq \overline{z}_k \geq \overline{z}_{k'}, \ \forall k' \text{ that is descendent of } k.$$

9.3.2 Linear-programming-based branch-and-bound

Branch-and-bound is the general nomenclature given to methods that operate based on solving relaxations of subproblems and using bounding information to preemptively prune branches in the enumerative search tree.

The characteristics that define a branch-and-bound method are thus the relaxation being solved (or how bounding is performed), and how subproblems are generated (how branching is performed). In the specific context of (mixed-)integer programming problems, bounding is performed utilising linear programming (LP) relaxations.

In regards to branching, we employ the following strategy utilising the information from the solution of the LP relaxation. At a given subproblem S_k, suppose we have an optimal solution with a fractional component $x_j^{*k} = \overline{x}_j \notin \mathbb{Z}^1$. We can then *branch* S_k into the following subproblems:

$$S_{k1} = S_k \cap \{x : x_j \leq \lfloor \overline{x}_j \rfloor\} \text{ and } S_{k2} = S_k \cap \{x : x_j \geq \lceil \overline{x}_j \rceil\}.$$

Notice that this implies that each of the subproblems will be disjunct (i.e., with no intersection in their feasible region) and have one additional constraint that eliminates the fractional part of the component around the solution of the LP relaxation.

Bounding can occur in three distinct ways. The first case is when the solution of the LP relaxation happens to be integer and, therefore, optimal for the subproblem itself. In this case, no further exploration along that subproblem is necessary and we say that the node has been *pruned by optimality*.

Figure 9.4 illustrates the process of pruning by optimality (in a maximisation problem). Each box denotes a subproblem, with the interval denoting known lower (primal) and upper (dual) bounds for the problem and x denoting the solution for the LP relaxation of the subproblem. In Figure 9.4, we see a pruning that is caused because a solution to the original (integer) subproblem has been identified by solving its (LP) relaxation, akin to the leaves in the enumerative tree represented in Figure 9.3. This can be concluded because the solution of the LP relaxation of subproblem S_1 is integer.

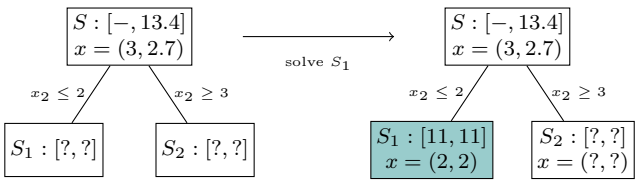


Figure 9.4: An example of pruning by optimality. Since the solution of LP relaxation of subproblem S_1 is integer, $x = (2, 2)$ must be optimal for S_1

Another type of pruning takes place when known global (primal) bounds can be used to prevent further exploration of a branch in the enumeration tree. Continuing the example in Figure 9.4, notice that the global lower (primal) bound $\underline{z} = 11$ becomes available and can be transmitted to all subproblems. Now suppose we solve the LP relaxation of S_2 and obtain the optimal value of $\overline{z}_2 = 9.7$. Notice that we are precisely in the situation described in Section 9.3.1. That is, the nodes descending from S_2 (i.e., their descendants) can only yield solutions that have objective function value worse than the dual bound of S_2, which, in turn, is worse than a known global primal (lower) bound. Thus, any further exploration among the descendants of S_2 would be fruitless in terms of yielding better solutions and can be pruned. This is known as *pruning by bound*, and is illustrated in Figure 9.5.

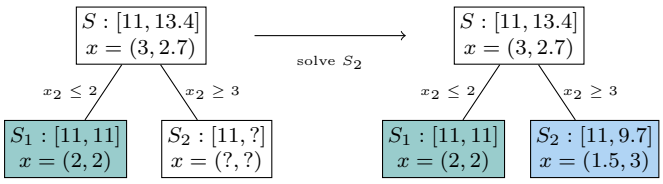


Figure 9.5: An example of pruning by bound. Notice that the newly found global bound holds for all subproblems. After solving the LP relaxation of S_2, we notice that $\overline{z}_2 \leq \underline{z}$, which renders the pruning.

The third type of pruning is called *pruning by infeasibility*, which takes place whenever the branching constraint added to the subproblem renders its relaxation infeasible, implying that the subproblem itself is infeasible (cf. Proposition 9.3)

Algorithm 5 presents a pseudocode for an LP-based branch-and-bound method. Notice that the algorithm keeps a list $\mathcal{L}$ of subproblems to be solved and requires that a certain rule to select which subproblem is solved next to be employed. This subproblem selection (often referred to as *search strategy*) can have considerable impacts on the performance of the method. Similarly, in case multiple components are found to be fractional, one must be chosen. Defining such *branching priorities* also has consequences to performance. We will discuss these in more depth in Chapter 11.

Also, recall that we have seen how to efficiently resolve a linear programming problem from an optimal basis once we include an additional constraint (in Chapter 6). It so turns out that an efficient dual simplex method is the kingpin of an efficient branch-and-bound method for (mixed)-integer programming problems.

Finally, although we developed the method in the context of integer programming problems, the method can be readily applied to mixed-integer programming problems, with the only difference being that the branch-and-bound steps are only applied to the integer variables while the continuous variables are naturally taken care of in the solution of the LP relaxations.

Let us finalise presenting a numerical example of the employment of branch-and-bound method to solve an integer programming problem. Consider the problem:

$$\begin{aligned} \max_{x} \;\; & z = 4x_1 - x_2 \\ \text{s.t.:} \;\; & 7x_1 - 2x_2 \leq 14 \\ & x_2 \leq 3 \\ & 2x_1 - 2x_2 \leq 3 \\ & x \in \mathbb{Z}^2_+. \end{aligned}$$

Algorithm 5 LP-relaxation-based branch-and-bound

1: **initialise.** $\mathcal{L} \leftarrow \{S\}$, $\underline{z} \leftarrow -\infty$, $\overline{x} \leftarrow -\infty$
2: **while** $\mathcal{L} \neq \emptyset$ **do**
3: select problem S_i from $\mathcal{L}$. $\mathcal{L} \leftarrow \mathcal{L} \setminus \{S_i\}$.
4: solve LP relaxation of S_i over P_i, obtaining z^i_{LP} and x^i_{LP}. $\overline{z}^i \leftarrow z^i_{LP}$.
5: **if** $S_i = \emptyset$ **then** return to step 2.
6: **else if** $\overline{z}^i \leq \underline{z}$ **then** return to step 2.
7: **else if** $x^i_{LP} \in \mathbb{Z}^n$ **then** $\underline{z} \leftarrow \overline{z}^i$, $\overline{x} \leftarrow x^i_{LP}$; and return to step 2
8: **end if**
9: select a fractional component x_j and create subproblems S_{i1} and S_{i2} with formulations P_{i1} and P_{i2}, respectively, such that

$$P_{i1} = P_i \cup \{x_j \leq \lfloor \overline{x}_j \rfloor\} \text{ and } P_{i2} = P_i \cup \{x_j \geq \lceil \overline{x}_j \rceil\}.$$

10: $\mathcal{L} \leftarrow \mathcal{L} \cup \{S_{i1}, S_{i2}\}$.
11: **end while**
12: **return** $(\overline{x}, \underline{z})$.

We start by solving its LP relaxation, as represented in Figure 9.6. We obtain the solution $x_{LP} = (20/7, 3)$ with objective value of $z = 59/7$. As the first component of x is fractional, we can generate subproblems by branching the node into subproblems S_1 and S_2, where

$$S_1 = S \cap \{x : x_1 \leq 2\}$$
$$S_2 = S \cap \{x : x_1 \geq 3\}.$$

The current enumerative (or branch-and-bound) tree representation is depicted in Figure 9.7.

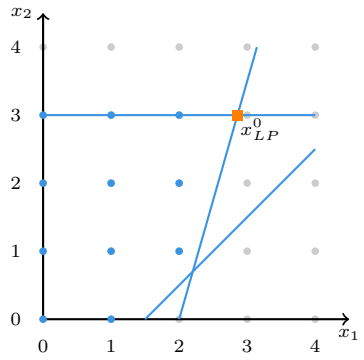


Figure 9.6: LP relaxation of the problem S

Suppose we arbitrarily choose to solve the relaxation of S_1 next. Notice that this subproblem consists of the problem S, with the added constraint $x_1 \leq 2$. The feasible region and solution of the LP relaxation of S_2 is depicted in Figure 9.8. Since we again obtain a new fractional solution $x^1_{LP} = (2, 1/2)$, we must branch on the second component, forming the subproblems

$$S_{11} = S_1 \cap \{x : x_2 = 0\}$$
$$S_{12} = S_1 \cap \{x : x_2 \geq 1\}.$$

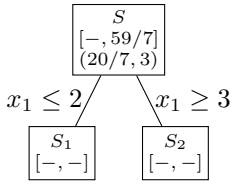


Figure 9.7: The branch-and-bound tree after branching S onto S_1 and S_2

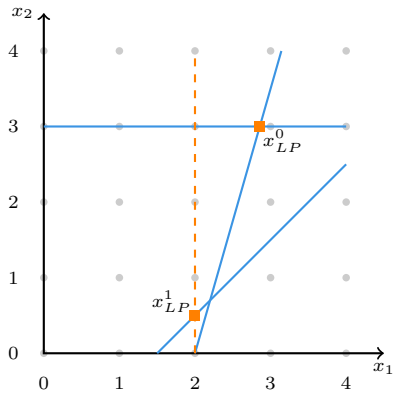


Figure 9.8: LP relaxation of subproblem S_1

Notice that, at this point, our list of active subproblems is formed by $\mathcal{L} = \{S_1, S_{11}, S_{12}\}$. Our current branch-and-bound tree is represented in Figure 9.9.

Suppose we arbitrarily choose to first solve S_2. One can see that this would render an infeasible subproblem, since the constraint $x_2 \geq 3$ does not intersect with the original feasible region and, thus, S_2 can be pruned by infeasibility.

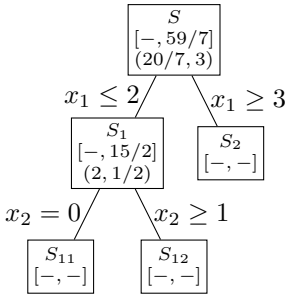


Figure 9.9: The branch-and-bound tree after branching S_1 onto S_{11} and S_{12}

Next, we choose to solve the LP relaxation of S_{12}, which yields an integer solution $x^{12}_{LP} = (2, 1)$. Therefore, an optimal solution for S_{12} was found, meaning that a global primal (lower) bound has been found and can be propagated to the whole branch-and-bound tree. Solving S_{11} next, we obtain the solution $x^{11}_{LP} = (3/2, 0)$ with optimal value $z = 6$. Since a better global primal (lower) bound is known, we can prune S_{11} by bound. As there are no further nodes to be explored, the solution for the original problem is the best (and, in this case, the single) integer solution found in the process (cf. Proposition 9.6), $x^* = (2, 1)$, $z^* = 7$. Figure 9.10 illustrates the feasible region

of the subproblems and their respective optimal solutions, while Figure 9.11 presents the final branch-and-bound tree with all branches pruned.

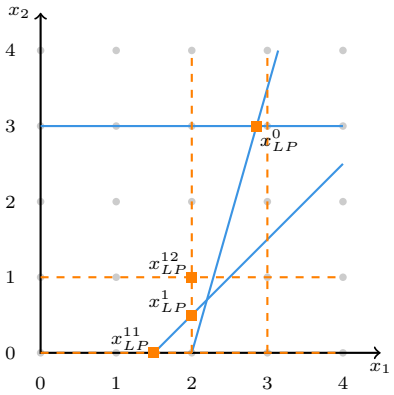


Figure 9.10: LP relaxations of all subproblems. Notice that S_{11} and S_{12} includes the constraints $x_1 \leq 2$ from the parent node S_1

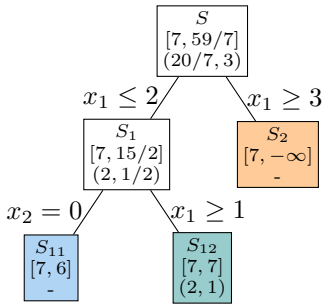


Figure 9.11: The final branch-and-bound tree

Notice that in this example, the order in which we solved the subproblems was crucial for pruning by bound the subproblem S_{11}, only possible because we happened to solve the LP relaxation of S_{12} first and that happened to yield a feasible solution and associated primal bound. This illustrates an important aspect associated with the branch and bound method: having good feasible solutions available early on in the process increases the likelihood of performing more pruning by bound, which is highly desirable in terms of computational savings (and thus, performance). We will discuss in more detail the impacts of different search strategies later on when we consider this and other aspects involved in the implementation of mixed-integer programming solvers.

9.4 Exercises

Problem 9.1: Uncapacitated Facility Location (UFL)

(a) Let $N = \{1, \ldots, n\}$ be a set of potential facilities and $M = \{1, \ldots, m\}$ a set of clients. Let $y_j = 1$ if facility j is opened, and $y_j = 0$ otherwise. Moreover, let x_{ij} be the fraction of client i's demand satisfied from facility j. The UFL can be formulated as the mixed-integer problem:

$$\begin{aligned}
(\text{UFL-W}) : \min_{x,y} \quad & \sum_{j \in N} f_j y_j + \sum_{i \in M} \sum_{j \in N} c_{ij} x_{ij} && && (9.1)\\
\text{s.t.:} \quad & \sum_{j \in N} x_{ij} = 1, && \forall i \in M, && (9.2)\\
& \sum_{i \in M} x_{ij} \leq m y_j, && \forall j \in N, && (9.3)\\
& x_{ij} \geq 0, && \forall i \in M, \forall j \in N, && (9.4)\\
& y_j \in \{0, 1\}, && \forall j \in N, && (9.5)
\end{aligned}$$

where f_j is the cost of opening facility j, and c_{ij} is the cost of satisfying client i's demand from facility j. Consider an instance of the UFL with opening costs $f = (4, 3, 4, 4, 7)$ and client costs

$$(c_{ij}) = \begin{pmatrix} 12 & 13 & 6 & 0 & 1 \\ 8 & 4 & 9 & 1 & 2 \\ 2 & 6 & 6 & 0 & 1 \\ 3 & 5 & 2 & 1 & 8 \\ 8 & 0 & 5 & 10 & 8 \\ 2 & 0 & 3 & 4 & 1 \end{pmatrix}$$

Implement (the model) and solve the problem with Julia using JuMP.

(b) An alternative formulation of the UFL is of the form

$$\begin{aligned}
(\text{UFL-S}) : \min_{x,y} \quad & \sum_{j \in N} f_j y_j + \sum_{i \in M} \sum_{j \in N} c_{ij} x_{ij} && && (9.6)\\
\text{s.t.:} \quad & \sum_{j \in N} x_{ij} = 1, && \forall i \in M, && (9.7)\\
& x_{ij} \leq y_j, && \forall i \in M, \forall j \in N, && (9.8)\\
& x_{ij} \geq 0, && \forall i \in M, \forall j \in N, && (9.9)\\
& y_j \in \{0, 1\}, && \forall j \in N. && (9.10)
\end{aligned}$$

Linear programming (LP) relaxations of these problems can be obtained by relaxing the binary constraints $y_j \in \{0, 1\}$ to $0 \leq y_j \leq 1$ for all $j \in N$. For the same instance as in part (a), solve the LP relaxations of UFL-W and UFL-S and compare the optimal costs of the LP relaxations against the optimal integer cost obtained in part (a).

Problem 9.2: LP-based branch-and-bound method

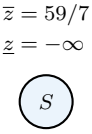
Consider the following IP problem and its standard form

$$\begin{array}{lll} \text{(IP)} & z = \max & 4x_1 - x_2 \\ & \text{s.t.:} & 7x_1 - 2x_2 \le 14 \\ & & x_2 \le 3 \\ & & 2x_1 - 2x_2 \le 3 \\ & & x_1, x_2 \in \mathbb{Z}_+ \end{array} \qquad \begin{array}{lllllll} \text{(IP)} & \text{z} = & \max & 4x_1 - x_2 & & & \\ & & \text{s.t.:} & 7x_1 + 2x_2 + x_3 & & = & 14 \\ & & & x_2 + x_4 & & = & 3 \\ & & & 2x_1 - 2x_2 + x_5 & & = & 3 \\ & & & x_1, \ldots, x_5 \in \mathbb{Z}_+ & & & \end{array}$$

Solve the IP problem by LP-based branch and bound, i.e., use LP relaxations to compute dual (upper) bounds. Use dual simplex to efficiently solve the subproblem of each node starting from the optimal basis of the previous node. Recall that the LP relaxation of IP is obtained by relaxing the variables $x_1, \ldots, x_5 \in \mathbb{Z}_+$ to $x_1, \ldots, x_5 \ge 0$.

Hint: The initial dual bound $\overline{z}$ is obtained by solving the LP relaxation of IP at the root node S. Let $[\underline{z}, \overline{z}]$ be the lower and upper bounds of each node. The optimal tableau and the initial branch-and-bound tree with only the root node S are shown below.

	-z	x_1	x_2	x_3	x_4	x_5
	-59/7	0	0	-4/7	-1/7	0
$x_1 =$	20/7	1	0	1/7	2/7	0
$x_2 =$	3	0	1	0	1	0
$x_5 =$	23/7	0	0	-2/7	10/7	1



You can proceed by branching on the fractional variable x_1 and imposing either $x_1 \le 2$ or $x \ge 3$. This creates two new subproblems $S_1 = S \cap \{x_1 \le 2\}$ and $S_2 = S \cap \{x_1 \ge 3\}$ in the branch-and-bound tree that can be solved efficiently using the dual simplex method, starting from the optimal tableau of S shown above, by first adding the new constraint $x_1 \le 2$ for S_1 or $x_1 \ge 3$ for S_2 to the optimal tableau. The dual simplex method can be applied immediately if the new constraint is always written in terms of non-basic variables before adding it to the tableau as a new row, possibly multiplying the constraint by -1 if needed.

Problem 9.3: Employing the branch-and-bound method graphically

Consider the following integer programming problem IP:

$$\begin{array}{lll} (IP): \max. & z = x_1 + 2x_2 \\ & \text{s.t.:} & -3x_1 + 4x_2 \le 4 \\ & & 3x_1 + 2x_2 \le 11 \\ & & 2x_1 - x_2 \le 5 \\ & & x_1, x_2 \in \mathbb{Z}_+ \end{array}$$

Plot (or draw) the feasible region of the linear programming (LP) relaxation of the problem IP, then solve the problems using the figure. Recall that the LP relaxation of IP is obtained by

replacing the integrality constraints $x_1, x_2 \in \mathbb{Z}_+$ by linear nonnegativity $x_1, x_2 \geq 0$ and upper bounds corresponding to the upper bounds of the integer variables ($x_1, x_2 \leq 1$ for binary variables).

(a) What is the optimal cost z_{LP} of the LP relaxation of the problem IP? What is the optimal cost z of the problem IP?

(b) Draw the border of the convex hull of the feasible solutions of the problem IP. Recall that the convex hull represents the *ideal* formulation for the problem IP.

(c) Solve the problem IP by LP-relaxation based branch-and-bound. You can solve the LP relaxations at each node of the branch-and-bound tree graphically. Start the branch-and-bound procedure without any primal bound.

Chapter 10

Cutting-planes Method

10.1 Valid inequalities

In this chapter, we will discuss the idea of generating and adding constraints to improve a formulation of a (possibly mixed-)integer programming problem. This idea can either be implemented in a priori setting, for example before employing the branch-and-bound method or as the solution method itself. These constraints are often called *valid inequalities* or *cuts*, though the latter is typically used in the context of cutting-plane methods.

Let us start by defining the integer programming problem

$$(IP) : \max_{x} \ \left\{ c^\top x : x \in X \right\}$$

where $X = P \cap \mathbb{Z}^n$ and $P = \{x \in \mathbb{R}^n : Ax \leq b, x \geq 0\}$, with $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$.

The idea of using constraints to solve IP is founded in the following observations. We know that $\mathbf{conv}(X)$ is a (convex) polyhedral set (cf. Definition 2.7) and, being so, there exists a finite set of inequalities $\tilde{A}x \leq \tilde{b}$ such that

$$\mathbf{conv}(X) = \left\{ x \in \mathbb{R}^n : \tilde{A}x \leq \tilde{b}, x \geq 0 \right\}.$$

Furthermore, if we had available $\tilde{A}x \leq \tilde{b}$, then we could solve IP by solving its linear programming relaxation.

Cutting-plane methods are based on the idea of iteratively approximating the set of inequalities $\tilde{A}x \leq \tilde{b}$ by adding constraints to the formulation P of IP. These constraints are called *valid inequalities*, a term we define more precisely in Definition 10.1.

Definition 10.1 (Valid inequality)**.** *An inequality $\pi^\top x \leq \pi_0$ is valid for $X \subset \mathbb{R}^n$ if $\pi^\top x \leq \pi_0$ for all $x \in X$.*

Notice that the condition for an inequality to be valid is that it does not remove any of the point in the original integer set X. In light of the idea of gradually approximating $\mathbf{conv}(X)$, one can infer that good valid inequalities are those that can "cut off" some of the area defined by the polyhedral set P, but without removing any of the points in X. This is precisely where the name *cut* comes from. Figure 10.1 illustrates the process of adding a valid inequality to a formulation P. Notice how the inequality exposes one of the facets of the convex hull of X. Cuts like such are called "facet-defining" and are the strongest types of cuts one can generate. We will postpone the discussion of stronger cuts to later in this chapter.

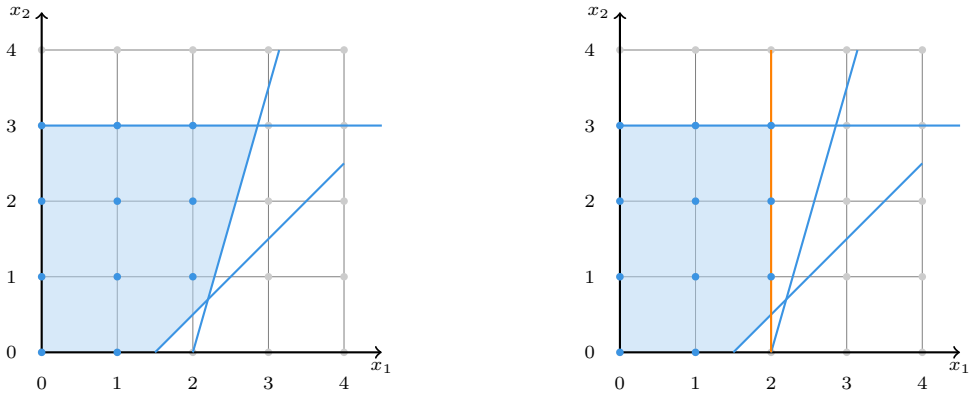


Figure 10.1: Illustration of a valid inequality being added to a formulation P. Notice how the inequality cuts off a portion of the polyhedral set P while not removing any of the feasible points X (represented by the dots)

10.2 The Chvátal-Gomory procedure

To develop a systematic procedure for generating valid inequalities in the context of a solution method for integer programming problems, we will rely on a two-step procedure. First, a cut that is valid for the polyhedral set P is (somewhat automatically) generated, and then it is made valid for the integer set X by a simple rounding procedure. Before we proceed, let us define the notion of valid inequalities in the context of linear programming problems.

Proposition 10.2 (Valid inequalities for polyhedral sets)**.** *An inequality $\pi^\top x \leq \pi_0$ is valid for $P = \{x \in \mathbb{R}^n : Ax \leq b, x \geq 0\}$, if and only if $P \neq \emptyset$ and there exists $u \geq 0$ such that $u^\top A \geq \pi$ and $u^\top b \leq \pi_0$.*

Proof. First, notice that, for $u \geq 0$ and $x \in P$, we have

$$\begin{aligned} Ax &\leq b \\ u^\top Ax &\leq u^\top b \\ \pi^\top x \leq u^\top Ax &\leq u^\top b \leq \pi_0, \end{aligned}$$

and, thus, it implies the validity of the cut, i.e., that $\pi^\top x \leq \pi_0, \forall x \in P$. Now, let us consider the other direction, which we can use linear programming duality to show. First, consider the primal problem

$$\begin{aligned} \max.\ & \pi^\top x \\ \text{s.t.:}\ & Ax \leq b \\ & x \geq 0 \end{aligned}$$

and its dual

$$\begin{aligned} \min.\ & u^\top b \\ \text{s.t.:}\ & u^\top A \geq \pi \\ & u \geq 0. \end{aligned}$$

Notice that $u^\top A \geq \pi$ can be seen as a consequence of dual feasibility, which is guaranteed to hold for some u since $\pi^\top x$ is bounded. Then, strong duality gives $u^\top b = \pi^\top x \leq \pi_0$, which completes the proof. □

One thing to notice is that valid cuts in the context of polyhedral sets are somewhat redundant, since, by definition, they do not alter the polyhedral set by any means. However, the concept can be combined with a simple yet powerful way of generating valid inequalities for integer sets by using rounding. This is stated in Proposition 10.3.

Proposition 10.3 (Valid inequalities for integer sets)**.** *Let $X = \left\{y \in \mathbb{Z}^1 : y \leq b\right\}$. The inequality $y \leq \lfloor b \rfloor$ is valid for X.*

The proof of Proposition 10.3 is somewhat straightforward and left as a thought exercise.

We can combine Propositions 10.2 and 10.3 into a single procedure to automatically generate valid inequalities. Let us start with a numerical example. Consider the set $X = P \cap \mathbb{Z}^n$ where P is defined by

$$P = \left\{x \in \mathbb{R}^2_+ : 7x_1 - 2x_2 \leq 14, x_2 \leq 3, 2x_1 - 2x_2 \leq 3\right\}.$$

First, let $u = \left[\frac{2}{7}, \frac{37}{63}, 0\right]$, which, for now, we can assume that they were arbitrarily chosen. We can then combine the constraints in P (the $Ax \leq b$ in Proposition 10.2) forming the constraint (equivalent to $u^\top Ax \leq u^\top b$)

$$2x_1 + \frac{1}{63}x_2 \leq \frac{121}{21}.$$

Now, notice that the constraint would remain valid for P if we simply *round down* the coefficients on the left-hand side (as $x \geq 0$ and all coefficients are positive). This would lead to the new constraint (notice that this yields a vector π in Proposition 10.2)

$$2x_1 + 0x_2 \leq \frac{121}{21}.$$

Finally, we can invoke Proposition 10.3 to generate a cut valid for X. This can be achieved by simply rounding down the righthand side (yielding π_0), obtaining

$$2x_1 + 0x_2 \leq 5,$$

which is valid for X, but not for P. Notice that, apart from the vector of weights u used to combine the constraints, everything else in the procedure of generating the valid inequality for X is automated. This procedure is known as the *Chvátal-gomory procedure* and can be formalised as follows.

Definition 10.4 (Chvátal-Gomory procedure)**.** *Consider the integer set $X = P \cap \mathbb{Z}^n$ where $P = \left\{x \in \mathbb{R}^n_+ : Ax \leq b\right\}$, A is an $m \times n$ matrix with columns $\{A_1, \ldots, A_n\}$ and $u \in \mathbb{R}^m_+$.*

The Chvátal-Gomory procedure consists of the following set of steps to generate valid inequalities for X:

1. *$\sum_{j=1}^n u^\top A_j x_j \leq u^\top b$ is valid for P as $u \geq 0$;*
2. *$\sum_{j=1}^n \lfloor u^\top A_j \rfloor x_j \leq u^\top b$ is valid for P as $x \geq 0$;*
3. *$\sum_{j=1}^n \lfloor u^\top A_j \rfloor x_j \leq \lfloor u^\top b \rfloor$ is valid for X as $\lfloor u^\top b \rfloor$ is integer.*

Perhaps the most striking result in the theory of integer programming is that *every* valid inequality for an integer set X can be obtained by employing the Chvátal-gomory procedure a number of times. This is formalised in Theorem 10.5, which has its proof provided as an exercise (see Exercise 10.1).

Theorem 10.5. *Every valid inequality for X can be obtained by applying the Chvátal-Gomory procedure a finite number of times.*

10.3 The cutting-plane method

Let us now consider how one could use valid inequalities to devise a solution method. The working paradigm behind a cutting-plane method is the *separation principle*.

The separation principle states that, given an integer set $X = P \cap \mathbb{Z}^n$, if a solution $x \notin X$, then there exists a hyperplane $\pi^\top x \leq \pi_0$ separating x and X. As one might infer, the challenge is how one can generate such pairs (π, π_0). This is precisely what is called the *separation problem* in the context of integer programming.

In general, these valid inequalities are generated from a family of inequalities $\mathcal{F}$, which are related to each other by properties related to, e.g., problem structure or the nature of the inequality itself. One way of thinking about it is to see the family of inequalities $\mathcal{F}$ as a means to dictate, to some extent, how the selection of weights u in the Chvátal-Gomory procedure is defined.

In any case, in possession of a family of inequalities and a method to solve the separation problem, we can pose a cutting-plane method in general terms. This is stated in Algorithm 6.

Algorithm 6 Cutting-plane algorithm

initialise. let $\mathcal{F} \subseteq \left\{(\pi, \pi_0) : \pi^\top x \leq \pi_0 \text{ is valid for } X\right\}$. $k = 0$.
while $x^k_{LP} \notin \mathbb{Z}^n$ **do**
    solve the LP relaxation over P, obtaining the optimal objective value z^k_{LP} and optimal solution x^k_{LP}.
    if $x^k_{LP} \notin \mathbb{Z}^n$ **then** find $(\pi^k, \pi^k_0) \in \mathcal{F}$ such that ${\pi^k}^\top x^k_{LP} > \pi^k_0$.
    else
        return (x^k_{LP}, z^k_{LP}).
    end if
    $P \leftarrow P \cup \left\{{\pi^k}^\top x \leq \pi^k_0\right\}$. $k = k + 1$.
end while
return (x^k_{LP}, z^k_{LP}).

The motivation for employing cutting-plane algorithms lies in the belief that only a few of all $|\mathcal{F}|$ inequalities (assuming $\mathcal{F}$ is finite, which might not be necessarily the case) is necessary, circumventing the computationally prohibitive need of generating all possible inequalities from $\mathcal{F}$.

Some other complicating aspects must be observed when dealing with cutting-plane algorithms. First, it might be so that a given family of valid inequalities $\mathcal{F}$ is not sufficient to expose the optimal solution $x \in X$, which might be the case, for example, if $\mathcal{F}$ cannot fully describe $\mathbf{conv}(X)$ or if the separation problem is unsolvable. In that case, the algorithm will terminate with a solution for the LP relaxation that is not integer, i.e., $x^k_{LP} \notin \mathbb{Z}^n$.

However, failing to converge to an integer solution is not a complete failure since, in the process, we have improved the formulation P (cf. Definition 8.3). In fact, this idea plays a major role in

professional-grade implementations of mixed-integer programming solvers, as we will see later.

10.4 Gomory's fractional cutting-plane method

One important cutting-plane method that is guaranteed to converge (in theory) to an integer solution is Gomory's fractional cutting-plane method. The method consists of exploiting the Chvátal-Gomory procedure (cf. Definition 10.4) to be the family of cuts generated while solving separation problem becomes simply the process of rounding to be applied to solutions of LP relaxations.

Specifically, consider the integer programming problem

$$(IP) : \max_x \ \left\{c^\top x : x \in X\right\},$$

where $X = \left\{x \in \mathbb{Z}^n_+ : Ax = b\right\}$. Recall that the optimal solution of the LP relaxation is characterised by a basis B formed by columns of the matrix A, i.e.,

$$A = [B \mid N] \text{ and } x = (x_B, x_N),$$

where x_B are the basic components of the solution and $x_N = 0$ the nonbasic components. The matrix N if formed by columns of A associated with the nonbasic variables x_N.

As we have discussed in Chapter 3, the system of equation $Ax = b$ can be written as

$$Bx_B + Nx_N = b \text{ or } x_B + B^{-1}Nx_N = B^{-1}b,$$

which is equivalent to $B^{-1}Ax = B^{-1}b$. Now, let $\overline{a}_{ij}$ be the element in row i and column j in $B^{-1}A$, and let $\overline{a}_{i0} = (B^{-1}b)_i$ be the i-th component of $B^{-1}b$. With that, we can represent the set of feasible solutions X as

$$\begin{aligned} &x_{B(i)} + \sum_{j \in I_N} \overline{a}_{ij} x_j = \overline{a}_{i0}, \ \forall i \in I \\ &x_j \in \mathbb{Z}_+, \ \forall j \in J, \end{aligned}$$

where $I = \{1, \ldots, m\}$, $J = \{1, \ldots, n\}$, $I_B \subset J$ are the indices of basic variables and $I_N = J \setminus I_B$ the indices of nonbasic variables. Notice that, at this point, we are simply recasting the problem IP by performing permutations of columns, since basic feasible solutions for the LP relaxation do not necessarily translate into a feasible solution for X.

However, assuming that we solve the LP relaxation of the integer programming problem IP and obtain an optimal solution $x = (x_B, x_N)$ with associated optimal basis B. If x is fractional, then it means that $\overline{a}_{i0}$ is fractional for some i.

From any of the rows i with fractional $\overline{a}_{i0}$, we can derive a valid inequality using the Chavátal-Gomory procedure. These inequalities, commonly referred to as *CG cuts*, take the form

$$x_{B(i)} + \sum_{j \in I_N} \lfloor \overline{a}_{ij} \rfloor x_j \leq \lfloor \overline{a}_{i0} \rfloor. \tag{10.1}$$

As this is thought to be used in conjunction with the simplex method, we must be able to state (10.1) in terms of the nonbasic variables x_j, $\forall j \in I_N$. To do so, we can replace $x_{B(i)} = \overline{a}_{i0} - \sum_{j \in I_N} \overline{a}_{ij} x_j$, obtaining

$$\sum_{j \in I_N} (\overline{a}_{ij} - \lfloor \overline{a}_{ij} \rfloor) x_j \geq (\overline{a}_{i0} - \lfloor \overline{a}_{i0} \rfloor),$$

which, by defining $f_{ij} = \overline{a}_{ij} - \lfloor \overline{a}_{ij} \rfloor$, can be written in the more conventional form

$$\sum_{j \in I_N} f_{ij} x_j \geq f_{i0}. \tag{10.2}$$

In the form of (10.2), this inequality is referred to as the Gomory (fractional) cut.

Notice that the inequality (10.2) is not satisfied by the optimal solution of the LP relaxation, since $x_j = 0, \forall j \in I_N$ and $f_{i0} > 0$. Therefore, this indicates that a cutting-plane method using this idea benefits from the employment of dual simplex, in line with the discussion in Section 6.1.2.

Let us present a numerical example illustrating the employment of Gomory's fractional cutting plane algorithm for solving the following integer programming problem

$$\begin{aligned} z = \max_x. \; & 4x_1 - x_2 \\ \text{s.t.:} \; & 7x_1 - 2x_2 \leq 14 \\ & x_2 \leq 3 \\ & 2x_1 - 2x_2 \leq 3 \\ & x_1, x_2 \in \mathbb{Z}_+. \end{aligned}$$

Figure 10.2 illustrates the feasible region of the problem and indicates the solution of its LP relaxation. Considering the tableau representation of the optimal basis for the LP relaxation, we have

x_1	x_2	x_3	x_4	x_5	RHS
0	0	-4/7	-1/7	0	59/7
1	0	1/7	2/7	0	20/7
0	1	0	1	0	3
0	0	-2/7	10/7	1	23/7

Notice that the tableau indicates that the component x_1 in the optimal solution is fractional. Thus, we can choose that row to generate a Gomory cut. This will lead to the new constraint (with added respective slack variable $s \geq 0$.

$$\frac{1}{7} x_3 + \frac{2}{7} x_4 - s = \frac{6}{7}.$$

We can proceed to add this new constraint to the problem, effectively adding an additional row to the tableau. After multiplying it by -1 (so we have s as a basic variable complementing the augmented basis), we obtain the new tableau

x_1	x_2	x_3	x_4	x_5	s	RHS
0	0	-4/7	-1/7	0	0	59/7
1	0	1/7	2/7	0	0	20/7
0	1	0	1	0	0	3
0	0	-2/7	10/7	1	0	23/7
0	0	-1/7	-2/7	0	1	-6/7

Notice that the solution remains dual feasible, which indicates the suitability of the dual simplex method. Applying the dual simplex method leads to the optimal tableau

x_1	x_2	x_3	x_4	x_5	s	RHS
0	0	0	0	-1/2	-3	15/2
1	0	0	0	0	1	2
0	1	0	0	-1/2	1	1/2
0	0	1	0	-1	-5	1
0	0	0	1	1/2	-1	5/2

Notice that we still have a fractional component, this time associated with x_2. We proceed in an analogous fashion, first generating the Gomory cut and adding the slack variable $t \geq 0$, thus obtaining

$$\frac{1}{2}x_5 - t = \frac{1}{2}.$$

Then, adding it to the previous tableau and employing the dual simplex again, leads to the optimal tableau

x_1	x_2	x_3	x_4	x_5	s	t	RHS
0	0	0	0	0	-4	-1	7
1	0	0	0	0	1	0	2
0	1	0	0	0	0	-1	1
0	0	1	0	0	-7	-2	2
0	0	0	1	0	0	1	2
0	0	0	0	1	-2	-2	1

Notice that now all variables are integer, and thus, an optimal solution for the original integer programming problem was found.

Some points are worth noticing. First, notice that, at the optimum, all variables, including the slacks, are integer. This is a consequence of having the Gomory cuts active at the optimal solution since (10.1), and consequently, (10.2), have both the left- and right-hand sides integer. Also, notice that at each iteration the problem increases in size, due to the new constraint being added, which implies that the basis also increases in size. Though this is an issue also in the branch-and-bound method, it can be a more prominent computational issue in the context of cutting-plane methods.

We can also interpret the progress of the algorithm in graphical terms. First of all, notice that we can express the cuts in terms of the original variables (x_1, x_2) by noticing that the original formulation gives $x_3 = 14 - 7x_1 + x_2$ and $x_4 = 3 - x_2$. Substituting x_3 and x_4 in the cut $\frac{1}{7}x_3 + \frac{2}{7}x_4 - s = \frac{6}{7}$ gives $x_1 \leq 2$. More generally, we have that cuts can be expressed using the original problem variables, as stated in Proposition 10.6.

Proposition 10.6. *Let β be the row l of B^{-1} selected to generate the cut, and let $q_i = \beta_i - \lfloor \beta_i \rfloor$, $i \in \{1, \ldots, m\}$. Then the cut $\sum_{j \in I_N} f_{lj} x_j \geq f_{l0}$, written in terms of the original variables, is the Chvátal-Gomory inequality*

$$\sum_{j=1}^{n} \lfloor qA_j \rfloor x_j \leq \lfloor qb \rfloor.$$

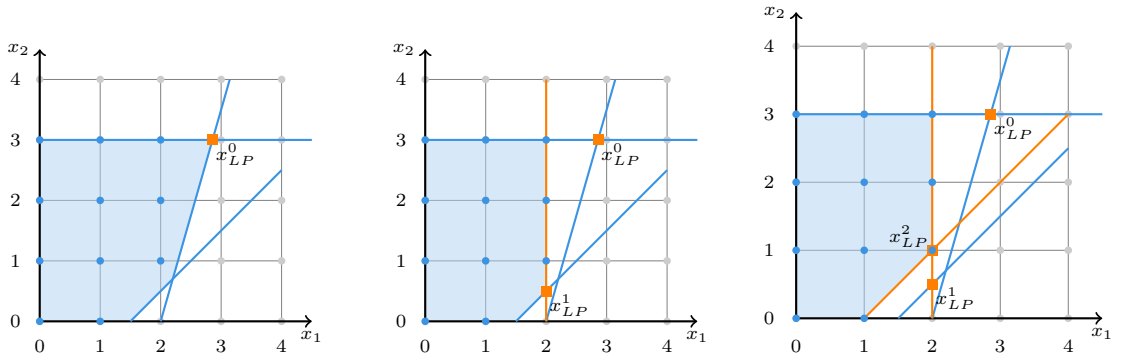


Figure 10.2: Feasible region of the LP relaxation (polyhedral set) and of the integer programming problem (blue dots) at each of three iterations taken to solve the integer programming problem. The inequalities in orange represent the Gomory cut added at each iteration expressed as a function of x_1 and x_2

10.5 Obtaining stronger inequalities

We finalise this chapter by discussing the notion of strong inequalities and providing an example of how they can be made stronger in some cases by means of an external process.

10.5.1 Strong inequalities

The notion of strong inequalities arises from the notion of stronger formulations, cf. Definition 8.3. That is, we say that an inequality is strong in terms of its relative quality in describing the convex hull of the integer set.

However, in this context, we are interested in comparing two alternative inequalities to decide which is stronger. For that, we can rely on the notions of *dominance* and related *redundancy* of inequalities.

Consider two valid inequalities $\pi x \leq \pi_0$ and $\mu x \leq \mu_0$ that are valid for a polyhedral set $P = \left\{x \in \mathbb{R}^n_+ : Ax \leq b\right\}$. Definition 10.7 formalises the notion of dominance.

Definition 10.7 (Dominance)**.** *The inequality* $\pi x \leq \pi_0$ dominates $\mu x \leq \mu_0$ *if these exists* $u > 0$ *such that* $\pi \geq u\mu$, $\pi_0 \leq u\mu_0$, *and* $(\pi, \pi_0) \neq (u\mu, u\mu_0)$.

Let us illustrate the concept of dominance with a numerical example. Consider the inequalities $2x_1 + 4x_2 \leq 9$ and $x_1 + 3x_2 \leq 4$, which are valid for $P = \mathbf{conv}(X)$, where

$$X = \{(0,0), (0,1), (0,2), (0,3), (0,4), (1,0), (1,1)\}.$$

Notice that, if we consider $u = 1/2$, we have that, for any $x = (x_1, x_2)$, that $1x_1 + 3x_2 \geq u(2x_1 + 4x_2) = x_1 + 2x_2$ and that $4 \leq 9u = 9/2$. Thus, we say that $x_1 + 3x_2 \leq 4$ dominates $2x_1 + 4x_2 \leq 9$. Figure 10.3 illustrates the two inequalities. Notice that $x_1 + 3x_2 \leq 4$ is a stronger inequality since it is more efficient in representing the convex hull of X than $2x_1 + 4x_2 \leq 9$.

Another related concept is the notion of redundancy. Clearly, in the presence of two constraints in which one dominates the other, the dominated constraint is also redundant and can be safely

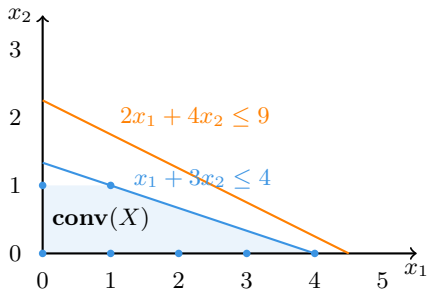


Figure 10.3: Illustration of dominance between constraints. Notice that $x_1 + 3x_2 \leq 4$ dominates $2x_1 + 4x_2 \leq 9$ and is thus stronger

removed from the formulation of a polyhedral set P. However, in some cases one might not be able to identify redundant constraints simply because no constraint is clearly dominated by another.

Even then, there might be a way to identify weak (or redundant) constraints by combining two or more constraints that then form a dominating constraint. This is formalised in Definition 10.8.

Definition 10.8 (Redundancy)**.** *The inequality* $\pi x \leq \pi_0$ *is* redundant *for* P *if there exists* $k \geq 1$ *valid inequalities* $\pi^i x \leq \pi_0^i$ *and* $k \geq 1$ *vectors* $u_i > 0$*, for* $i \in \{1, \ldots, k\}$*, such that* $\left(\sum_{i=1}^k u_i \pi^i\right) x \leq \sum_{i=1}^k u_i \pi_0^i$ *dominates* $\pi x \leq \pi_0$*.*

Once again, let us illustrate the concept with a numerical example. Consider we generate the inequality $5x_1 - 2x_2 \leq 6$, which is valid for the polyhedral set

$$P = \left\{x \in \mathbb{R}^2_+ : 6x_1 - x_2 \leq 9, 9x_1 - 5x_2 \leq 6\right\}.$$

The inequality $5x_1 - 2x_2 \leq 6$ is not dominated by any of the inequalities forming P. However, if we set $u_i = (\frac{1}{3}, \frac{1}{3})$, we obtain $5x_1 - 2x_2 \leq 5$, which in turn dominates $5x_1 - 2x_2 \leq 6$. Thus, we can conclude that the generated inequality is redundant and does not improve the formulation of P. This is illustrated in Figure 10.4.

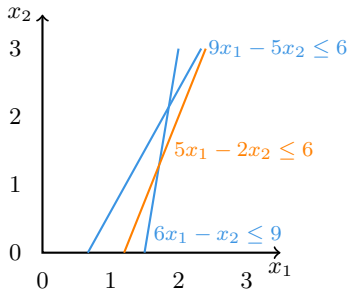


Figure 10.4: Illustration of a redundant inequality. Notice how the inequality $5x_1 - 2x_2 \leq 6$ (in orange) does not dominate any of the other inequalities

As one might realise, checking whether a newly generated inequality improves the current formulation is a demanding task, as it requires finding the correct set of coefficients u for all constraints currently forming the polyhedral set P. Nonetheless, the notions of redundancy and dominance can be used to guide procedures that generate or improve existing inequalities. Let us discuss one of such procedures, in the context of 0-1 knapsack inequalities.

10.5.2 Strengthening 0-1 knapsack inequalities

Let us consider the family of constraints known as knapsack constraints and see how they can be strengthened. For that, let us first define the knapsack set

$$X = \left\{ x \in \{0,1\}^n : \sum_{j=1}^{n} a_j x_j \leq b \right\}.$$

We assume that $a_j \geq 0$, $j \in N = [n]$, and $b > 0$. Let us start by defining the notion of a *minimal cover*.

Definition 10.9 (minimal cover)**.** *A set $C \subseteq N$ is a cover if $\sum_{j \in C} a_j > b$. A cover C is minimal if $C \setminus \{j\}$ for all $j \in C$ is not a cover.*

Notice that a cover C refers to any selection of items that exceed the budget b of the constraint and this selection is said to be a minimal cover if, upon removing of any item of the selection, the constraint becomes satisfied. This logic allows us to design a way to generate valid inequalities using covers. This is the main result in Proposition 10.10.

Proposition 10.10. *If $C \subseteq N$ is a cover for X, then a valid cover inequality for X is*

$$\sum_{j \in C} x_j \leq |C| - 1.$$

Proof. Let $R = \left\{ j \in N : x_j^R = 1 \right\}$, for $x^R \in X$. If $\sum_{j \in C} x_j^R > |C| - 1$, then $|R \cap C| = |C|$ and $C \subseteq R$. Thus, $\sum_{j \in N} a_j x_j^R = \sum_{j \in R} a_j > b$, which violates the inequality and implies that $x^R \notin X$. □

The usefulness of Proposition 10.10 becomes evident if C is a minimal cover. Let us consider a numerical example to illustrate this. Consider the knapsack set

$$X = \left\{ x \in \{0,1\}^7 : 11x_1 + 6x_2 + 6x_3 + 5x_4 + 5x_5 + 4x_6 + x_7 \leq 19 \right\}.$$

The following are minimal cover inequalities for X.

$$\begin{aligned} x_1 + x_2 + x_3 &\leq 2 \\ x_1 + x_2 + x_6 &\leq 2 \\ x_1 + x_5 + x_6 &\leq 2 \\ x_3 + x_4 + x_5 + x_6 &\leq 3 \end{aligned}$$

Notice that we would obtain a non-minimal cover C' by adding an inequality $x_i \leq 1$, $i \in C' \setminus C$, to a minimal cover inequality of C, which makes the cover inequality somewhat redundant. For example, adding $x_7 \leq 1$ to a minimal cover inequality $x_1 + x_2 + x_3 \leq 2$ of $C = \{1,2,3\}$ yields a non-minimal cover $C' = \{1,2,3,7\}$ with inequality $x_1 + x_2 + x_3 + x_7 \leq 3$.

There is a simple way to strengthen cover inequalities, using the notion of *extended* cover inequalities. One can extend a cover inequality by expanding the set C with elements that have a coefficient a_j, $j \in N \setminus C$ greater or equal to all coefficients a_i, $i \in C$. This guarantees that a swap between elements must happen for the inequality to be feasible, meaning that the right-hand side of the constraint remains $|C| - 1$. This is summarised in Proposition 10.11.

Proposition 10.11. *If C is a cover for X, the extended cover inequality*

$$\sum_{j \in E(C)} x_j \leq |C| - 1$$

with $E(C) = C \cup \{j \in N : a_j \geq a_i, \forall i \in C\}$ is valid for X.

We leave the proof as a thought exercise. Let us however illustrate this using the previous numerical example. For $C = \{3, 4, 5, 6\}$, $E(C) = \{1, 2, 3, 4, 5, 6\}$, yielding the inequality

$$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 \leq 3$$

which is stronger than $x_3 + x_4 + x_5 + x_6 \leq 3$ as the former dominates the latter, cf. Definition 10.7.

The above serves as an example of how strong inequalities can be generated for problems with a known structure. However, this is just one example of many other well-known cutting-generation methods. In the next chapter, we will mention a few other alternative techniques to generate valid inequalities that enrich professional-grade implementations of MIP solvers.

10.6 Exercises

Exercise 10.1: Chvátal-Gomory (C-G) procedure

Consider the set $X = P \cap \mathbb{Z}^n$ where $P = \{x \in \mathbb{R}^n : Ax \leq b, x \geq 0\}$ and in which A is an $m \times n$ matrix with columns $\{A_1, \ldots, A_n\}$. Let $u \in \mathbb{R}^m$ with $u \geq 0$. The Chvátal-Gomory (C-G) procedure to construct valid inequalities for X uses the following 3 steps:

1. $\sum_{j=1}^{n} uA_j x_j \leq ub$ is valid for P, as $u \geq 0$ and $\sum_{j=1}^{n} A_j x_j \leq b$.
2. $\sum_{j=1}^{n} \lfloor uA_j \rfloor x_j \leq ub$ is valid for P, as $x \geq 0$.
3. $\sum_{j=1}^{n} \lfloor uA_j \rfloor x_j \leq \lfloor ub \rfloor$ is valid for X, as any $x \in X$ is integer and thus $\sum_{j=1}^{n} \lfloor uA_j \rfloor x_j$ is integer.

Show that every valid inequality for X can be obtained by applying the Chvátal-Gomory procedure a finite number of times.

Hint: We show this for the 0-1 case. Thus, let $P = \{x \in \mathbb{R}^n : Ax \leq b, 0 \leq x \leq 1\}$, $X = P \cap \mathbb{Z}^n$, and suppose that $\pi x \leq \pi_0$ with $\pi, \pi_0 \in \mathbb{Z}$ is a valid inequality for X. We show that $\pi x \leq \pi_0$ can be obtained by applying Chvátal-Gomory procedure a finite number of times. We do this in parts by proving the following claims **C1**, **C2**, **C3**, **C4**, and **C5**.

C1. An inequality $\pi x \leq \pi_0 + t$ with $t \in \mathbb{Z}_+$ is valid for P.

C2. For a large enough $M \in \mathbb{Z}_+$, the inequality

$$\pi x \leq \pi_0 + M\Big(\sum_{j \in N^0} x_j + \sum_{j \in N^1} (1 - x_j)\Big) \tag{10.3}$$

is valid for P for every partition (N^0, N^1) of N.

C3. If $\pi x \leq \pi_0 + \tau + 1$ is a valid inequality for X with $\tau \in \mathbb{Z}_+$, then

$$\pi x \leq \pi_0 + \tau + \sum_{j \in N^0} x_j + \sum_{j \in N^1} (1 - x_j) \tag{10.4}$$

is also a valid inequality for X for every partition (N^0, N^1) of N.

C4. If

$$\pi x \leq \pi_0 + \tau + \sum_{j \in T^0 \cup \{p\}} x_j + \sum_{j \in T^1} (1 - x_j) \tag{10.5}$$

and

$$\pi x \leq \pi_0 + \tau + \sum_{j \in T^0} x_j + \sum_{j \in T^1 \cup \{p\}} (1 - x_j) \tag{10.6}$$

are valid inequalities for X, where $\tau \in \mathbb{Z}_+$ and (T^0, T^1) is any partition of $\{1, \ldots, p-1\}$, then

$$\pi x \leq \pi_0 + \tau + \sum_{j \in T^0} x_j + \sum_{j \in T^1} (1 - x_j) \tag{10.7}$$

is also a valid inequality for X.

C5. If

$$\pi x \leq \pi_0 + \tau + 1 \tag{10.8}$$

is a valid inequality for X with $\tau \in \mathbb{Z}_+$, then

$$\pi x \leq \pi_0 + \tau \tag{10.9}$$

is also a valid inequality for X.

Finally, after proving the claims **C1** - **C5**, if we start with $\tau = t - 1 \in \mathbb{Z}_+$ and successively apply **C5** for $\tau = t - 1, \ldots, 0$, turning each valid inequality (10.8) of X into a new one (10.9), it leads to the inequality $\pi x \leq \pi_0$ which is valid for X. This shows that every valid inequality $\pi x \leq \pi_0$ of X with $\pi, \pi_0 \in \mathbb{Z}_+$ can be obtained by applying the C-G procedure a finite number of times.

Exercise 10.2: Chvátal-Gomory (C-G) procedure example

(a) Consider the set $X = \{\mathbf{x} \in 0, 1^5 : 3x_1 - 4x_2 + 2x_3 - 3x_4 + x_5 \leq -2\}$. Derive the following inequalities as C-G inequalities:

  (*i*) $x_2 + x_4 \geq 1$

  (*ii*) $x_1 \leq x_2$

(b) Consider the set $X = \{x \in 0, 1^4 : x_i + x_j \leq 1 \text{ for all } i, j \in \{1, \ldots, 4\} : i \neq j\}$. Derive the clique inequalities $x_1 + x_2 + x_3 \leq 1$ and $x_1 + x_2 + x_3 + x_4 \leq 1$ as C-G inequalities.

Exercise 10.3: Cuts from the simplex tableau

Consider the integer programming problem IP:

$$\begin{aligned} (IP) \quad \max_{x_1, x_2} \quad & 2x_1 + 5x_2 \\ \text{s.t.:} \quad & 4x_1 + x_2 \leq 28 \\ & x_1 + 4x_2 \leq 27 \\ & x_1 - x_2 \leq 1 \\ & x_1, x_2 \in \mathbb{Z}_+. \end{aligned}$$

The LP relaxation of the problem IP is obtained by relaxing the integrality constraints $x_1, x_2 \in \mathbb{Z}_+$ to $x_1 \geq 0$ and $x_2 \geq 0$. The LP relaxation of IP in standard form is the problem LP:

$$\begin{aligned} (LP) \quad \max_{x_1, x_2} \quad & 2x_1 + 5x_2 \\ \text{s.t.:} \quad & 4x_1 + x_2 + x_3 = 28 \\ & x_1 + 4x_2 + x_4 = 27 \\ & x_1 - x_2 + x_5 = 1 \\ & x_1, x_2, x_3, x_4, x_5 \geq 0 \end{aligned}$$

The optimal Simplex tableau after solving the problem LP with primal Simplex is

x_1	x_2	x_3	x_4	x_5	RHS
0	0	-1/5	-6/5	0	-38
1	0	4/15	-1/15	0	17/3
0	1	-1/15	4/15	0	16/3
0	0	-1/3	1/3	1	2/3

(a) Derive two fractional Gomory cuts from the rows of x_1 and x_5, and express them in terms of the original variables x_1 and x_2.

(b) Derive the same cuts as in part (a) as Chvátal-Gomory cuts. *Hint:* Use Proposition 5 from Lecture 9. Recall that the bottom-right part of the tableau corresponds to $B^{-1}A$, where B^{-1} is the inverse of the optimal basis matrix and A is the original constraint matrix. You can thus obtain the matrix B^{-1} from the optimal Simplex tableau, since the last three columns of A form an identity matrix.

Exercise 10.4: More Gomory cuts

Consider the following integer programming problem IP:

$$\begin{aligned} (IP): \max.\ & z = x_1 + 2x_2 \\ \text{s.t.:}\ & -3x_1 + 4x_2 \leq 4 \\ & 3x_1 + 2x_2 \leq 11 \\ & 2x_1 - x_2 \leq 5 \\ & x_1, x_2 \in \mathbb{Z}_+ \end{aligned}$$

Solve the problem by adding Gomory cuts to the LP relaxation until you find an integer solution.

Exercise 10.5. The cover separation problem

Consider the 0-1 knapsack set :

$$X = \left\{ x \in \{0,1\}^7 : 11x_1 + 6x_2 + 6x_3 + 5x_4 + 5x_5 + 4x_6 + x_7 \leq 19 \right\}$$

and a solution $\bar{x} = (0, 2/3, 0, 1, 1, 1, 1)$ to its LP relaxation. Find a cover inequality cutting out (violated by) the fractional solution $\bar{x}$.

Chapter 11

Mixed-integer Programming Solvers

11.1 Modern mixed-integer linear programming solvers

In this chapter, we will discuss some of the numerous features that are shared by most professional-grade implementations of mixed-integer programming (MIP) solvers. As it will become clear, MIP solvers are formed by an intricate collection of techniques that have been developed over the last few decades. The continuous improvement and development of new such techniques have enabled performance improvements beyond purely hardware performance progress. In fact, this is a very lively and exciting research area, with new features being proposed and incorporated in these solvers with the frequent new releases of these tools.

The main difference between MIP solver implementations is which "tricks" and techniques they have implemented. In some cases, these are not disclosed in full detail, since the high-performing solvers are commercial products subject to trade secrets. Luckily, some open-source (such as CBC, SCIP, and HiGHS) have been made available, but they are not up to par with commercial implementations in terms of performance as yet.

We will focus on describing the most important techniques forming a professional-grade MIP solver implementation. Most MIP solvers allow for significant tuning and on-off toggling of these techniques. Therefore, knowing the most important techniques and how they work can be beneficial in configuring MIP solvers to your own needs.

MIP solvers implement a method that is called *branch-and-cut* which consists of a combination of the linear-programming (LP)-based branch-and-bound method (as described in Chapter 9) and a cutting-plane method (as described in Chapter 10) that is employed at the root note (or the first subproblem LP relaxation) and possibly at later nodes as well. Figure 11.1 illustrates the typical flowchart of a MIP solver algorithm.

The first phase consists of a preprocessing phase called *presolve*. In that, the problem formulation is analysed to check whether redundant constraints or "loose" variables can be trivially removed. In addition, more sophisticated techniques can be employed to try to infer the optimal value of some variables via logic or to tighten their bounds. For simple enough problems, the presolve might be capable of returning an optimal solution or a certificate that the problem is infeasible or unbounded.

Then, the main solution loop starts, similarly to what we have described in Chapter 9 when discussing the branch-and-bound method. A node selection method is employed and the LP relaxation is solved. Then, branching is applied and the process continues until an optimal solution has been found.

The main difference however relates to the extra *cuts* and *heuristics* phases. Together with the

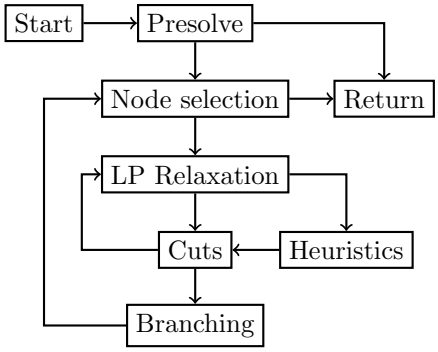


Figure 11.1: The flowchart of a typical MIP solver. The nodes represent phases of the algorithm

presolve, these are likely the phases that most differ between implementations of MIP solvers. The cut phase consists of the employment of a cutting-plane method onto the current LP relaxation with the aim of either obtaining an integer solution (and thus pruning the branch by optimality) or strengthening the formulation of the LP relaxation, as discussed in Chapter 10. Each solver will have their own family of cuts that are used in this phase, and typically a collection of cut families is used simultaneously. The heuristics phase is used in combination to try to obtain primal feasible solutions from the LP relaxations (possibly augmented by cuts) so primal bounds (integer and feasible solution values) can be obtained and broadcasted to the whole search tree, hopefully fostering pruning by bound.

In what follows, we will discuss some of the main techniques in each of these phases.

11.2 Presolving methods

Presolving (or preprocessing) methods are typically employed before the start of the branch-and-cut method. These methods have three main goals: (i) reducing the problem size by fixing variables and eliminating constraints; (ii) strengthening the LP relaxation by identifying bounds and coefficients that can be tightened; and (iii) exploiting integrality to improve formulation and identify problem structures (e.g., knapsack or assignment structures)

Detecting infeasibility and redundancy

Many techniques used in the preprocessing phase rely on the notion of constraint activity. Consider the constraint $a^\top x \leq b$, with $x \in \mathbb{R}^n$ as a decision variable vector, $l \leq x \leq u$, $b \in \mathbb{R}$, where (a, b, l, u) are given. We say that the *minimum activity* of the constraint is given by

$$\alpha_{\min} = \min\left\{a^\top x : l \leq x \leq u\right\} = \sum_{j:a_j>0} a_j l_j + \sum_{j:a_j<0} a_j u_j.$$

Analogously, the *maximum activity* of a constraint is given by

$$\alpha_{\max} = \max\left\{a^\top x : l \leq x \leq u\right\} = \sum_{j:a_j>0} a_j u_j + \sum_{j:a_j<0} a_j l_j$$

Notice that the constraint activity is simply capturing what is the minimum and maximum (respectively) values the left-hand side of $a^\top x \leq b$ could assume. This constraint activity can be used

in a number of ways. For example, if there is a constraint for which $\alpha_{\min} > b$, then the problem is trivially *infeasible*. On the other hand, if one observes that $\alpha_{\max} \leq b$ for a given constraint, then the constraint can be safely removed since it is guaranteed to be redundant.

Bound tightening

Another important presolving method is bound tightening, which, as the name suggests, tries to tighten lower and upper bounds of variables, thus strengthening the LP relaxation formulation. There are alternative ways that this can be done, and they typically trade off how much tightening can be observed and how much computational effort they require.

One simple way of employing bound tightening is by noticing the following. Assume, for simplicity, that $a_j > 0$, $\forall j \in J$. Then, we have that

$$\alpha_{\min} = \sum_{j \in J} a_j l_j = a^\top l \leq a^\top x \leq b,$$

where $l \in \mathbb{R}^{|J|}$. From the second inequality, we obtain

$$a_j x_j \leq b - a^\top x + a_j x_j. \tag{11.1}$$

Let $a^\top = (\overline{a}, a_j)^\top$, $x = (\overline{x}, x_j)$ and $l = (\bar{l}, l_j)$, then we have $\alpha_{\min} = a^\top l = \overline{a}^\top \bar{l} + a_j l_j$ and $a^\top x = \overline{a}^\top \overline{x} + a_j x_j$. By the definition of *minimum activity*, we know that $\overline{\alpha}_{\min} = \overline{a}^\top \bar{l} \leq \overline{a}^\top \overline{x}$, which is equivalent to

$$\alpha_{\min} - a_j l_j \leq a^\top x - a_j x_j.$$

This can be reformulated as

$$b - a^\top x + a_j x_j \leq b - \alpha_{\min} + a_j l_j.$$

Combining this result with what was obtained from (11.1), we have that

$$a_j x_j \leq b - a^\top x + a_j x_j \leq b - \alpha_{\min} + a_j l_j,$$

from which we can extract the bound

$$x_j \leq \frac{b - \alpha_{\min} + a_j l_j}{a_j},$$

that is, an upper bound for x_j. The procedure can be analogously adapted to obtain a lower bound as well. Furthermore, rounding can be employed in the presence of integer variables.

Another common bound tightening technique consists of solving a linear programming subproblem for each variable x_j, $j = 1, \ldots, n$. Let

$$IP : \text{min. } \left\{ c^\top x : x \in X = P \cap \mathbb{Z}^n \right\}$$

where P is a polyhedral set. Then, the optimal solution value of the subproblem

$$LP_{x_j} : \text{min. } \{ x_j : x \in P \}$$

provides a lower bound for x_j that considers all possible constraints at once. Analogously, solving LP_{x_j} as a maximisation problem yields an upper bound. Though this can be done somewhat efficiently, this clearly has steeper computational requirements.

Coefficient tightening

Different from bound tightening, coefficient tightening techniques aim at improving the strength of existing constraints. The simplest form consists of the following. Let $a_j > 0$, $x_j \in \{0, 1\}$, such that $\alpha_{\max} - a_j < b$. If such coefficients are identified, then the constraint

$$a_j x_j + \sum_{j':j' \neq j} a_{j'} x_{j'} \leq b$$

can be modified to become

$$(\alpha_{\max} - b)x_j + \sum_{j':j' \neq j} a_{j'} x_{j'} \leq (\alpha_{\max} - a_j).$$

Notice that the modified constraint is valid for the original integer set while dominating the original constraint (cf. Definition 10.7 since, on the left-hand side, we have that $\alpha_{\max} - b < a_j$ and on the right-hand side we have $\alpha_{\max} - a_j < b$).

Other methods

There is a wide range of methods employed as preprocessing, and they vary greatly among different solvers and even different modelling languages. Thus, compiling an exhaustive list is no trivial feat. Some other common methods that are employed include:

- **Merge of parallel rows and columns**: methods implemented to identify pairs of rows (constraints) and columns (variables) with constant proportionality coefficient (i.e., are linearly dependent) and merge them into a single entity, thus reducing the size of the model.
- **Domination tests between constraints**: heuristics that test whether domination between selected constraints can be asserted so that some constraints can be deemed redundant and removed.
- **Clique merging**: a clique is a subset of vertices of a graph that are fully connected. Assume that $x_j \in \{0, 1\}$ for $j \in \{1, 2, 3\}$ and that the three constraints hold:

  $$\begin{aligned} x_1 + x_2 &\leq 1 \\ x_1 + x_3 &\leq 1 \\ x_2 + x_3 &\leq 1. \end{aligned}$$

  Then, one can think of these constraints forming a clique between the imaginary nodes 1, 2, and 3, which renders the clique cut $x_1 + x_2 + x_3 \leq 1$. Many other similar ideas using this graph representation of implication constraints, known as *conflict graphs* are implemented in presolvers to generate valid inequalities for MIPs.
- **Greatest common denominator (GCD) reduction**: Assume that $x_j \in \{0, 1\}$ for $j \in J$. We can use the GCD of the coefficients $a = [a_1, \ldots, a_n]$ to generate or tighten inequalities. Let $\mathbf{gcd}(a)$ be the GCD of all coefficients a_j in a. Then, we can generate the valid inequality

  $$\sum_{j \in J} \frac{a_j}{\mathbf{gcd}(a)} x_j \leq \left\lfloor \frac{b}{\mathbf{gcd}(a)} \right\rfloor.$$

Some final remarks are worth making. Most solvers might, at some point in the process of solving the MIP refer to something called *restart*, which consists of reapplying some or all of the techniques associated with the preprocessing phase after a few iterations of the branch-and-cut process. This can be beneficial since in the solution process new constraints are generated (cuts) which might lead to new opportunities for reducing the problem or further tightening bounds.

In addition, conflict graphs can contain information that can be exploited in a new round of preprocessing and transmitted across all search tree, a process known as propagation. Conflict graphs and propagation are techniques originally devised for constraint programming and satisfiability (SAT) problems, but have made considerable inroads in MIP solvers as well.

11.3 Cut generation

One major component of MIP solvers is its cut generation procedures. In practice, MIP solvers implement what is called *branch-and-cut*, which is a method formed by the amalgamation of branch-and-bound (as we have seen in Chapter 9) and cutting-plane methods (Chapter 10). In fact, the combination of both methods is arguably the most important methodological development that ultimately rendered MIP solvers reliable to be used in practice in many applications.

There is an interesting symbiotic relationship emerging from the combination of the two methods. In general, cutting planes are useful in tightening the existing formulation but often fail in being able to generate all the cuts required to expose an integer optimal solution. On the other hand, branch-and-bound might require considerable computational efforts to adequately expose the integer optimal solutions of all subproblems, but is guaranteed to converge in finite time (even if this time is not feasible in a practical sense). Thus, the combination of both allows for a method that is far more likely to terminate within reasonable computation time.

However, some care must be considered when generating cuts in branch-and-cut settings, since it can quickly increase the dimension of the subproblems, which would lead to amplified consequences to the performance of the branch-and-bound part. This is mitigated with a feature called *cut pool*, which consists of a way to make sure that only selected cuts, e.g., cuts that are violated by the solution of the relaxation, are considered in the subproblem. Such cuts are chosen using what is referred to as cut selection routines.

Furthermore, most professional-grade solvers allow for the definition of user cuts, which are user-defined cuts that are problem-specific, but give strong customisation powers to advanced users. As it turns out, the types of cuts available is one of the main differentiators between solvers and the search for cuts that are both efficient and generic is an active research and development direction.

The most common types of cuts utilised include fractional (or Gomory) cuts, of the form

$$\sum_{j \in I_N} f_{ij} x_j \geq f_{i0}, \text{ where } f_{ij} = \overline{a}_{ij} - \lfloor \overline{a}_{ij} \rfloor$$

and Chvátal-Gomory cuts for pure integer constraints, which are dependent on the heuristic or process used to define the values of the multipliers u in

$$\sum_{j=1}^{n} \lfloor uA_j \rfloor x_j \leq \lfloor ub \rfloor.$$

For example, zero-half cuts (with $u \in [0, 1/2]$) and mod-k cuts ($u \in \{0, 1/k, \ldots, (k-1)/k\}$) are available in most solvers. Other cuts such as knapsack (or cover) inequalities, mixed-integer round-

ing, and clique cuts are also common and, although they can be shown to be related to the Chvátal-Gomory procedure, are generated by means of heuristics (normally referring to a cut generation procedure).

11.3.1 Cut management: generation, selection and discarding

With the numerous possibilities of cuts to be generated, it becomes clear that one must be mindful of the total of cuts generated and its associated trade-offs. These trade-offs are managed taking into account which cuts to generate and which cuts to select from the cut pool to be added to the problem. In particular, in the case of the generation and selection of cuts, this can often be influenced by users, by selecting how "aggressively" cuts are generated.

For example, although dual simplex is perfectly suited for resolving the linear programming (LP) relaxation after the addition of cuts, if multiple cuts are added at once, it might mean that resolving the LP relaxation from the previous dual optimal basis is not much more beneficial than resolving the LP relaxation from scratch. Moreover, adding cuts increases the size of the basis in the problem, which in turn increases the size of the basic matrix B and its inverse B^{-1}, thus gradually increasing the time per iteration of the method. Lastly, some cuts might require the solution of a demanding separation problem, e.g., lifted cover inequalities. Although these can provide considerable improvements to the formulation, the computational effort required for generating them might dampen the performance improvement they incur.

Selecting which cuts would be the most efficient is not an easy task. Ideally, we would like to choose the minimal number of cuts that has the most impact in terms of improving the LP relaxation. Clearly, we can only try to achieve this via proxies. Solvers normally associate a scoring to each cut taking into account a collection of criteria and these scores are then used in the process of selecting the cuts (their cut selection routines), e.g., selecting a fraction of the top scorer cuts or discarding those that the score is below a certain threshold.

One criterion for trivially discarding cuts is numerical stability. Cuts with too large or too small coefficients are prone to cause numerical issues due to the matrix decomposition methods employed within the simplex method. Therefore, these can be easily disregarded in the selection process.

An important proxy for efficiency is the notion of *depth* of the cut. That is, a cut can have its depth measured by the distance between the hyperplane that forms the cut and the solution of the LP relaxation. The larger this distance, the "deeper" the cut is cutting through the LP relaxation, which could potentially mean that the cut is more efficient.

Another important proxy is *orthogonality*. Cuts that are pairwise orthogonal with other cuts are likely to be more effective. This is easy to see if you think of the extreme case of zero orthogonality, or the cuts being parallel, clearly meaning that one of the cuts is dominated by the other. Orthogonality can also be measured against the objective function, in which case we are interested in cuts that are almost parallel (but not exactly, as this would lead to numerical issues) to the objective function since those are more likely to cause improvement in the dual bound (LP relaxation optimal) value.

11.4 Variable selection: branching strategy

As we discussed in Chapter 9, some decisions in terms of selecting variables to branch and subproblems to solve can have a great impact in the total number of subproblems solved in a branch-and-bound method. Variable selection is still a topic under intensive research, with newer ideas

only recently being made available in the main solvers.

Variable selection, commonly referred to as *branching strategies* in most MIP solver implementations, refers to the decision of which of the currently fractional variables should be chosen to generate subproblems. There are basically three main methods most commonly used, which we discuss next. Furthermore, most MIP solvers allow the user to set priority weights for the variables, which define priority orders for variable selection. These can be useful, for example, when the user knows that the problem possesses a dependence structure between variables (e.g., location and allocation variables, where allocation can only happen if the location decision is made) that the solver cannot infer automatically.

Maximum infeasibility

The branching strategy called *maximum infeasibility* consists of choosing the variable with the fractional part as close to 0.5 as possible, or, more precisely, selecting the variables $j \in \{1, \ldots, n\}$ as

$$\arg \max_{j \in \{1,\ldots,n\}} \min \{f_j, 1 - f_j\}$$

where $f_j = x_j - \lfloor x_j \rfloor$. This, in effect, is trying to reduce as most as possible the infeasibility of the LP relaxation solution, which in turn would more quickly lead to a feasible (i.e., integer) solution. An analogous form called *minimum infeasibility* is often available, and as the name suggests, focuses on selecting the variables that are closer to being integer-valued.

Strong branching

Strong branching can be understood as an explicit look-ahead strategy. That is, to decide which variable to branch on, the method performs branching on all possible variables and chooses the one that provides the best improvement on the dual (LP relaxation) bound. Specifically, for each fractional variable x_j, we can solve the LP relaxations corresponding to branching options $x_j \leq \lfloor x^{\text{LP}} \rfloor$ and $x_j \geq \lceil x^{\text{LP}} \rceil$ to obtain LP relaxation objective values Z_j^D and Z_j^U, respectively. We then choose the fractional variable x_j that leads to subproblems with the best LP relaxation objective values, defined as

$$\arg \max_{j \in \{1,\ldots,n\}} \min \left\{ Z_j^D, Z_j^U \right\},$$

assuming a minimisation problem. Strong branching thus compares the worst LP relaxation bound for each fractional x_j and picks the fractional variable for which this value is the best.

As one might suspect, there is a trade-off related to the observed reduction in the number of nodes explored, given by the more prominent improvement of the dual bound, and how computationally intensive is the method. There are, however, ideas that can exploit this trade-off more efficiently. First, the solution of the subproblems might yield information related to infeasibility and pruning by limit, which can be used in favour of the method.

Another idea is to limit the number of simplex iterations performed when solving the subproblems associated with each branching option. This allows for using an approximate solution of the subproblems and potential savings in computational efforts. Some solvers offer a parameter that allows the user to set this iteration limit value.

Pseudo-cost branching

Pseudo-cost branching relies on the idea of using past information from the search process to estimate gains from branching on a specific variable. Because of this reliance on past information, the method tends to be more reliable later in the search tree, where more information has been accumulated on the impact of choosing a variable for branching.

These improvement estimates are the so-called *pseudo-costs*, which compile an estimate of how much the dual (LP relaxation) bound has improved per fractional unit of the variable that has been reduced. More specifically, let

$$f_j^- = x_j^{\text{LP}} - \lfloor x_j^{\text{LP}} \rfloor \text{ and } f_j^+ = \lceil x_j^{\text{LP}} \rceil - x_j^{\text{LP}}. \tag{11.2}$$

Then, we can define the quantities Ψ_j^- and Ψ_j^+ to be the average improvement in the dual bound observed per fractional unit reduced and increased, respectively, whenever the variable x_j has been selected for branching, i.e, for each branching direction. Notice that this requires that several subproblems to be solved for a reliable estimate to be available.

Then, we can define the quantities

$$\Delta_j^- = f_j^- \Psi_j^- \text{ and } \Delta_j^+ = f_j^+ \Psi_j^+ \tag{11.3}$$

which represent the estimated change to be observed when selecting the variable x_j for branching, based on the current fractional parts f_j^- and f_j^+. In effect, these are considered in a branching score, with the branching variable being selected as, for example,

$$j = \underset{j=1,\ldots,n}{\text{argmax}} \left\{ \alpha \min \left\{ \Delta_j^-, \Delta_j^+ \right\} + (1-\alpha) \max \left\{ \Delta_j^-, \Delta_j^+ \right\} \right\}.$$

where $\alpha \in [0, 1]$. Setting the value of α trades off two aspects. Assume a maximisation problem. Then, setting α closer to zero will slow down *degradation*, which refers to the decrease of the upper bound (notice that the dual bound is decreasing and thus, Δ^+ and Δ^- are negative). This strategy improves the chances of finding a good feasible solution on the given branch, and, in turn, potentially fostering pruning by bound. In contrast, setting α closer to one increases the rate of decrease (improvement) of the dual bound, which can be helpful for fostering pruning once a good global primal bound is available. Some solvers allow for considering alternative branching score functions.

As one might suspect, it might take several iterations before reliable estimates Ψ^+ and Ψ^- are available. The issue with unreliable pseudo-costs can be alleviated with the use of a hybrid strategy known as *reliability* branching[1]. In that, variables that are deemed unreliable for not having been selected for branching a minimum number of times $\eta \in [4, 8]$, have strong branching employed instead.

GUB branching

Constraints of the form

$$\sum_{j=1}^{k} x_j = 1$$

are referred to as special ordered sets 1 (or SOS1) which, under the assumption that $x_j \in \{0, 1\}$, $\forall j \in \{1, \ldots, k\}$ implies that only one variable can take value different than zero. Notice you may

[1] Tobias Achterberg, Thorsten Koch, and Alexander Martin (2005), Branching rules revisited, Operations Research Letters

have SOS1 sets involving continuous variables, which, in turn, would require the use of binary variables to be modelled appropriately.

Branching on these variables might lead to unbalanced branch-and-bound trees. This is because the branch in which x_j is set to a value different than zero, immediately defines the other variables to be zero, leading to an early pruning by optimality or infeasibility. In turn, unbalanced trees are undesirable since they preclude the possibility of parallelisation and might lead to issues related to searches that focus on finding leaf notes quickly.

To remediate this, the idea of using a *generalised upper bound* is employed, leading to what is referred to as GUB branching (with some authors referring to this as SOS1 branching). A generalised upper bound is an upper bound imposed on the sum of several variables. In GUB branching, branching for binary variables is imposed considering the following rule:

$$S_1 = S \cap \{x : x_{j_i} = 0, \forall i \in \{1, \ldots, r\}\}$$
$$S_2 = S \cap \{x : x_{j_i} = 0, \forall i \in \{r+1, \ldots, k\}\}$$

where $r = \text{argmax}_{t \in \{1,\ldots,k\}} \sum_{j=1}^{t} x_j \leq 0.5$. Notice the upper bounding on some of the variables, which inspires the name GUB branching. That is, only a subset of the variables are forced to be zero, while several others are left unconstrained, which favours more balanced search trees. As a final remark, constraints of the form of

$$\sum_{j=1}^{k} x_j \leq 1$$

can also benefit from the use of GUB branching, with the term GUB being perhaps better suited in this case.

11.5 Node selection

We now focus on the strategies associated with selecting the next subproblem to be solved. As we have seen in Chapter 9, the order in which the subproblem's LP relaxations are solved can have a major impact on the total number of nodes explored, and, consequently, on the efficiency of the method. The alternative strategies for node selection typically trade off the following:

1. Focus on finding primal feasible solutions earlier, to foster pruning by bound.
2. Alternatively, focus on improving the dual bound faster, hoping that once an integer solution is found, more pruning by bound is possible.
3. Increase *ramp-up*, which means increasing the number of unsolved nodes in the list of subproblems so that these might be solved in parallel. For that, the nodes must be created, and the faster they are opened, the earlier parallelisation can benefit the search.
4. Minimise computational effort by minimising the overhead associated with changing the subproblems to be solved. That means that children nodes are preferred over other nodes, in a way to minimise the changes needed to assemble a starting basis for the dual simplex method.

You might notice that points 1 and 2 are conflicting since while the former would benefit from a *breadth*-focused search (that is, having wider trees earlier is preferable than deeper trees) the latter would benefit from searches that dive *deeply* into the tree searching from leaf nodes. Points 3 and

4 pose exactly the same dilemma: the former benefits from breadth-focusing searches while the latter benefits from depth-focusing searches.

The main strategies for node selection are, to a large extent, ideas to emphasise each or a combination of the above.

Depth-first search (DFS) and breadth-first search (BFS)

A depth-first search focus on diving down into the search tree, prioritising nodes at lower levels. It has the effect of increasing the chances of finding leaves, and potentially primal feasible solutions, earlier. Furthermore, because the problems being successively solved are very similar, with the exception of one additional branching constraint, the dual simplex methods can be efficiently restarted and often fewer iterations are needed to find the optimal of the children's subproblem relaxation. On the other hand, as a consequence of the way the search is carried out, it is slower in populating the list of subproblems.

In contrast, breadth-first search gives priority to nodes at higher levels in the tree, ultimately causing a horizontal spread of the search tree. This has as a consequence a faster improvement of the dual bound, at the expense of potentially delaying the generation of primal feasible solutions. This also generates more subproblems quickly, fostering diversification (more subproblems and potentially more information to be re-utilised in repeated rounds of preprocessing) and opportunities for parallelisation.

Best bound

Best bound consists of the strategy of choosing the next node to be solved by selecting that with the best dual (LP relaxation) bound. It leads to a breadth-first search pattern, but with the flexibility to allow potentially good nodes that are in deeper levels of the tree to be selected.

Ultimately, this strategy fosters a faster improvement of the dual bound, but with a higher overhead on the set-up of the subproblems, since they can be quite different from each other in terms of their constraints. One way to mitigate this overhead is to perform *diving* sporadically, which consists of, after choosing a node by best bound, temporarily switching to a DFS search for a few iterations.

Best estimate or best projection

Uses a strategy similar to that employing pseudo-costs to choose which variable to branch on. However, instead of focusing on objective function values, it uses estimates of the node progress towards feasibility relative to its bound degradation.

To see how this works, assume that the parent node has been solved, and a dual bound z_D is available. Now, using our estimates in (11.3), we can calculate an estimate of the potential primal feasible solution value E, given by

$$E = z_D + \sum_{j=1}^{n} \min\left\{\Delta_j^-, \Delta_j^+\right\}.$$

The expression E is an estimate of what is the best possible value an integer solution could have if it were to be generated by rounding the LP relaxation solution. These estimates can also take into account the feasibility per se, trying to estimate feasibility probabilities considering known feasible solutions and how fractional the subproblem LP relaxation solution is.

11.6 Primal heuristics

The last element of MIP solvers we must consider is the set of primal heuristics they MIP solver has available. The term "primal" refers to the fact that these are methods geared towards either (i) building primal feasible solutions normally from a solution obtained from a relaxation; or (ii), geared towards improving on previously known primal solutions. The former strategy is often referred to as *constructive* heuristics, while the latter is called *improvement* heuristics.

The name heuristic refers to the fact that these are methods that are not guided by any optimality certificates per se, but focus only on performing local improvements repeatedly (or within a neighbourhood or a current reference solution), according to a given metric of solution difference.

Primal heuristics play three main important roles in MIP solver algorithms. First, they are employed in the preprocessing phase to verify whether the model can be proved to be feasible, by constructing a primal feasible solution. Second, constructive heuristics are very powerful in generating feasible solutions during the branch-and-bound phase, meaning that it can make primal feasible solutions available *before* they are found at leaf nodes when pruning by optimality (i.e., with integer LP relaxation solution), and therefore fostering early pruning by bound. Lastly, heuristics are a powerful way to obtain reasonably (often considerably) good solutions, which in practical cases might be sufficient, given computational or time limitations and precision requirements.

11.6.1 Diving heuristics

Diving heuristics are used in combination with node selection strategies that search in breadth (instead of depth). In simple terms, it consists of performing a local depth search at the node being considered with no or very little backtracking, with the hope of reusing the subproblem structure while searching for primal feasible solutions. The main difference is that the subproblems are generated in an alternative tree in which branching is based on rounding and fixing instead of the standard branching we have seen in Chapter 9.

Once the heuristic terminates, the structure is discarded, but the solution, if found, is kept. Notice that the diving can also be preemptively aborted if it either renders an infeasible subproblem or if it leads to a relaxation with a worse bound than a known primal bound from an incumbent solution. Another common termination criterion consists of limiting the total number of LP iterations solving the subproblems or the total number of subproblems solved.

The most common types of rounding employed in diving heuristics include *fractional diving*, in which the variable selected for rounding is simply that with the smallest fractional component, i.e., x_j is chosen, such that the index j is given by

$$\underset{i=1,\ldots,n}{\operatorname{argmax}} \left\{ \min \left\{ x_i - \lfloor x_i \rfloor, \lceil x_i \rceil - x_i \right\} \right\}.$$

Another common idea consists of selecting the variables to be rounded by considering a reference solution, which often is an incumbent primal feasible solution. This *guided dive* is then performed by choosing the variable with the smallest fractional value when compared against this reference solution.

A third idea consists of taking into account the number of *locks* associated with the variable. The locks refer to the number of constraints that are potentially made infeasible by rounding up or down a variable. This potential infeasibility stems from taking into count the coefficient of the variable, the type of constraint and whether rounding it up or down can potentially cause infeasibility. This is referred to as *coefficient diving*.

11.6.2 Local searches and large-neighbourhood searches

Local and large-neighbourhood searches are, contrary to most diving heuristics, improvement heuristics. In these, a reference solution is used to search for new solutions within its *neighbourhood*. This is simply the idea of limiting any solutions found to share a certain number of elements (for example, having the same values in some components) with the reference solution. We say that a solution is a k-neighbour solution if they share k components.

Local search and large-neighbourhood search simply differ in terms of scope; the former allows for only localised change while the latter considers wider possibilities for divergence from the reference solution.

Some of these searches are seldom used and often turned off by default in professional-grade solvers. They tend to be expensive from a computational standpoint because they require considerable extra work. Most of them are based on fixing and/or relaxing the integrality of a number of variables, adding extra constraints and/ or changing objective functions, and resolving which can be considerably onerous. Let us present the most common heuristics found in professional grade solvers.

Relaxation-induced neighbourhood search (RINS) and Relaxation-enforced neighbourhood search

The relaxation-induced neighbourhood search (or RINS) is possibly the most common improvement heuristic available in modern implementations[2]. The heuristic tries to find solutions that present a balance between proximity of the current LP solution, hoping this would improve the solution quality, and proximity to an incumbent (i.e., best-known primal feasible) solution, emphasising feasibility.

In a nutshell, the method consists of the following. After solving the LP relaxation of a node, suppose we obtain the solution x^{LP}. Also, assume we have at hand an incumbent (integer feasible) solution $\overline{x}$. Then, we form an auxiliary MIP problem in which we fix all variables coinciding between x^{LP} and $\overline{x}$. This can be achieved by using the constraints

$$x_j = \overline{x}_j,\ \forall j \in \{1, \ldots, n\} : \overline{x}_j = x_j^{\mathrm{LP}},$$

which, in effect, fixes these variables to integer values and removes them from the problem, as they can be converted to parameters (or input data). Notice that this constrains the feasible space to be in the (potentially large) neighbourhood of the incumbent solution. In later iterations, when more of the components of the relaxation solutions x^{LP} are integer, this becomes a more localised search, with fewer degrees of freedom. Finally, this additional MIP is solved and, in case an optimal solution is found, a new incumbent solution might become available.

In contrast, relaxation-enforced neighbourhood search (or RENS) is a constructive heuristic, which has not yet seen a wider introduction in commercial-grade solvers, though it is available in CBC and SCIP[3].

The main differences between RINS and RENS are the fact that no incumbent solution is considered (hence the dropping of the term "induced") but rather the LP relaxation solution x^{LP} fully defines the neighbourhood (explaining the name "enforced").

[2] Emilie Danna, Edward Rothberg, and Claude Le Pape (2005), Exploring relaxation induced neighborhoods to improve MIP solutions, Mathematical Programming

[3] Timo Berthold (2014), RENS: The optimal rounding, Mathematical Programming Computation

Once again, let us assume we obtain the solution x^{LP}. And again, we fix all integer-valued variables in x^{LP}, forming a large neighbourhood

$$x_j = x_j^{\text{LP}},\ \forall j \in \{1, \ldots, n\} : x_j^{\text{LP}} \in \mathbb{Z}.$$

One key difference is how the remaining variables are treated. For those components that are fractional, the following bounds are imposed

$$\lfloor x_j \rfloor \leq x_j \leq \lceil x_j \rceil,\ \forall j \in \{1, \ldots, n\} : x_j^{\text{LP}} \notin \mathbb{Z}.$$

Notice that this in effect makes the neighbourhood considerably smaller around the solution x^{LP}. Then, the MIP subproblem with all these additional constraints is solved and a new incumbent solution may be found.

Local branching

The idea of local branching is to allow the search to be performed in a neighbourhood of controlled size, which is achieved by the use of an L_1-norm[4]. The size of the neighbourhood is controlled by a divergence parameter Δ, which in the case of binary variables, amounts to being the Humming distance between the variable vectors.

In its most simple form, it can be seen as the following idea. From an incumbent solution $\overline{x}$, one can generate and impose the following neighbourhood-inducing constraint

$$\sum_{j=1}^{n} |x_j - \overline{x}_j| \leq \Delta$$

and then solve the resulting MIP.

The original use of local branching (as the name suggests) as proposed is to use this constraint directly to form a branching rule in an auxiliary tree search. However, most solvers use it by means of subproblems as described above.

Feasibility pump

Feasibility pump is a constructive heuristic that, contrary to the previous heuristics, has made inroads in most professional-grade solvers and is often employed by default. The focus is exclusively on trying to find a first primal feasible solution. The idea consists of, from the LP relaxation solution x^{LP}, performing alternate steps of rounding and solving a projection step, which happens to be an LP problem.

Starting from the x^{LP}, the method starts by simply rounding the components to obtain an integer solution $\overline{x}$. If this rounded solution is feasible, the algorithm terminates. Otherwise, we perform a projection step by replacing the LP relaxation objective function with

$$f^{\text{aux}}(x) = \sum_{j=1}^{n} |x_j - \overline{x}_j|$$

and resolving it. This is called a projection because it is effectively finding a point in the feasible region of the LP relaxation that is the closest to the integer solution $\overline{x}$. This new solution x^{LP} is once again rounded and the process repeats until a feasible integer solution is found.

[4]Matteo Fischetti and Andrea Lodi (2003), Local branching, Mathematical Programming

It is known that feasibility pump can suffer from cycling, that is, repeatedly finding the same x^{LP} and $\overline{x}$ solutions. This can be alleviated by performing random perturbations on some of the components of $\overline{x}$.

Feasibility pump is an extremely powerful method and plays a central role in many professional-grade solvers. It is also useful in the context of mixed-integer nonlinear programming models. More recently, variants have been developed[5], taking into account the quality of the projection (i.e., taking into account also the original objective function) and discussing theoretical properties of the methods and its convergence guarantees.

[5]Timo Berthold, Andrea Lodi, and Domenico Salvagnin (2018), Ten years of feasibility pump, and counting, EURO Journal on Computational Optimization

11.7 Exercises

Problem 11.1: Preprocessing and primal heuristics

(a) *Tightening bounds and redundant constraints*

Consider the LP below,

$$\begin{aligned} \text{max. } & 2x_1 + x_2 - x_3 \\ \text{s.t.: } & 5x_1 - 2x_2 + 8x_3 \leq 15 \\ & 8x_1 + 3x_2 - x_3 \leq 9 \\ & x_1 + x_2 + x_3 \leq 6 \\ & 0 \leq x_1 \leq 3 \\ & 0 \leq x_2 \leq 1 \\ & x_3 \geq 1. \end{aligned}$$

Derive tightened bounds for variables x_1 and x_3 from the first constraint and eliminate redundant constraints after that.

(b) *Primal heuristics (RINS)*

Consider the formulation UFL-W:

$$\text{(UFL-W)} : \min_{x,y} \sum_{j \in N} f_j y_j + \sum_{i \in M} \sum_{j \in N} c_{ij} x_{ij} \tag{11.4}$$

$$\text{s.t.: } \sum_{j \in N} x_{ij} = 1, \ \forall i \in M, \tag{11.5}$$

$$\sum_{i \in M} x_{ij} \leq m y_j, \ \forall j \in N, \tag{11.6}$$

$$x_{ij} \geq 0, \qquad \forall i \in M, \ \forall j \in N, \tag{11.7}$$

$$y_j \in \{0,1\}, \ \forall j \in N, \tag{11.8}$$

where f_j is the cost of opening facility j, and c_{ij} is the cost of satisfying client i's demand from facility j. Consider an instance of the UFL with opening costs $f = (21, 16, 30, 24, 11)$ and client costs

$$(c_{ij}) = \begin{pmatrix} 6 & 9 & 3 & 4 & 12 \\ 1 & 2 & 4 & 9 & 2 \\ 15 & 2 & 6 & 3 & 18 \\ 9 & 23 & 4 & 8 & 1 \\ 7 & 11 & 2 & 5 & 14 \\ 4 & 3 & 10 & 11 & 3 \end{pmatrix}$$

In the UFL problem, the facility production capacities are assumed to be large, and there is no budget constraint on how many facilities can be built. The problem thus has a feasible solution if at least one facility is opened. We choose an initial feasible solution $\bar{y} = (0, 1, 0, 0, 0)$.

Try to improve the solution by using relaxation-induced neighbourhood search (RINS) and construct a feasible solution using relaxation-enforced neighbourhood search (RENS).

Problem 11.2: Variable and node selection

Using Branch-and-Bound, solve the UFL-W problem from the exercises in Chapter 9:

$$\begin{aligned}
(\text{UFL-W}) : \min_{x,y} \quad & \sum_{j \in N} f_j y_j + \sum_{i \in M} \sum_{j \in N} c_{ij} x_{ij} && && (11.9)\\
\text{s.t.:} \quad & \sum_{j \in N} x_{ij} = 1, && \forall i \in M, && (11.10)\\
& \sum_{i \in M} x_{ij} \leq m y_j, && \forall j \in N, && (11.11)\\
& x_{ij} \geq 0, && \forall i \in M, \forall j \in N, && (11.12)\\
& y_j \in \{0,1\}, && \forall j \in N, && (11.13)
\end{aligned}$$

with opening costs $f = (4, 3, 4, 4, 7)$ and client costs

$$(c_{ij}) = \begin{pmatrix} 12 & 13 & 6 & 0 & 1 \\ 8 & 4 & 9 & 1 & 2 \\ 2 & 6 & 6 & 0 & 1 \\ 3 & 5 & 2 & 1 & 8 \\ 8 & 0 & 5 & 10 & 8 \\ 2 & 0 & 3 & 4 & 1 \end{pmatrix}.$$

- To obtain a good start for branching, use strong branching at the root node. All following branching variables should be selected using maximum infeasibility.
- For node selection, use the best bound selection to minimze the number of evaluated nodes.

Chapter 12

Decomposition Methods

12.1 Large-scale problems

In this chapter, we consider the notion of *decomposition*, which consists of a general term used in the context of mathematical programming to refer to solution methods that utilise some separability mechanism to more efficiently solve large-scale problems.

In general, decomposition methods are based on the premise that it is more efficient, under a computational standpoint, to repeatedly resolve a (collection of) smaller instances of a problem than to solve the full-scale original problem. More recently, with the widespread adoption of multithreaded processors and computing clusters with multiple nodes, decomposition methods have become attractive as parallelisation strategies, which can yield considerable computational savings.

The decomposition methods presented here utilise the explicit representation of polyhedral sets, as stated in Theorem 12.1, to iteratively reconstruct the full-scale problem, with the hope that the structure containing the optimal vertex will be successfully reconstructed before all of the problem itself is reconstructed. It turns out that this is the case in many applications, which is precisely the feature that renders these methods very efficient in some contexts. This is the class of methods we are going to analyse in this chapter, first the Dantzig-Wolfe decomposition and related column generation, and then its equivalent dual method, generally known as Benders' decomposition.

In either case, decomposition methods are designed in a way that they seek to break problems into easier parts by removing linking elements. Specifically, let

$$(P) : \text{min. } \left\{ c^\top x : x \in X \right\},$$

where $X = \bigcap_{k=1}^{K} X_k$, for some $K > 0$, and

$$X_k = \left\{ x_k \in \mathbb{R}_+^{n_k} : D_k x_k = d_k \right\}, \forall k \in \{1, \ldots, K\}.$$

That is, X is the intersection of K standard-form polyhedral sets. Our objective is to devise a way to break into K separable parts that can be solved separately and recombined as a solution for P. In this case, this can be straightforwardly achieved by noticing that P can be equivalently stated as

$$\begin{array}{rlllll} (P) : \text{max. } _x & c_1^\top x_1 & + \cdots + & c_K^\top x_K & & \\ \text{s.t.:} & D_1 x_1 & & & & = d_1 \\ & & \ddots & & & \vdots \\ & & & D_K x_K & & = d_K \\ & x_1, & \ldots, & x_K & & \in \mathbb{R}_+^n \end{array}$$

has a structure that immediately allows for separation. That is, P could be solved as K independent problems

$$P_k : \text{min. } \left\{c_k^\top x_k : x_k \in X_k\right\}$$

in parallel and then combine their individual solutions onto a solution for P, simply by making $\overline{x} = [x_k]_{k=1}^K$ and $c^\top \overline{x} = \sum_{i=1}^K c_k^\top \overline{x}_k$. Notice that, if we were to assume that the solution time scales linearly (it does not; it grows faster than linear) and $K = 10$, then solving P as K separated problems would be ten times faster (that is not true; there are bottlenecks and other considerations to take into account, but the point stands).

Unfortunately, *complicating structures* often compromise this natural separability, preventing one from being able to directly exploit this idea. Specifically, two types of complicating structures can be observed. The first is of the form of *complicating constraint*. That is, we observe that a constraint is such that it connects variables from (some of) the subsets X_k. In this case, we would notice that P has an additional constraint of the form

$$A_1 x_1 + \cdots + A_K x_K = b,$$

which precludes separability, since the problem structure becomes

$$\begin{array}{rccccc} P' : \text{max. } _x & c_1^\top x_1 & +\cdots+ & c_K^\top x_K & & \\ \text{s.t.:} & A_1 x_1 & +\cdots+ & A_K x_K & = b & \\ & D_1 x_1 & & & = d_1 & \\ & & \ddots & & \vdots & \\ & & & D_K x_K & = d_K & \\ & x_1, & \ldots, & x_K & \in \mathbb{R}^n_+. & \end{array}$$

The other type of complicating structure is the case in which the same set of decision variable is present in multiple constraints, or multiple subsets X_k. In this case, we observe that variables of a subproblem $k \in \{1, \ldots, K\}$ has nonzero coefficient in another subproblem $k' \neq k$, $k' \in \{1, \ldots, K\}$. Hence, problem P takes the form of

$$\begin{array}{rccccc} P'' : \text{max. } _x & c_0^\top x_0 + & c_1^\top x_1 & +\cdots+ & c_K^\top x_K & \\ \text{s.t.:} & A_1 x_0 + & D_1 x_1 & & & = d_1 \\ & \vdots & & \ddots & & \vdots \\ & A_K x_0 + & & & D_K x_K & = d_K \\ & x_0, & x_1, & \ldots, & x_K & \in \mathbb{R}^n_+. \end{array}$$

The challenging aspect is that a specific method becomes more suitable depending on the complicating structure. Therefore, being able to identify these structures is one of the key success factors in terms of the chosen method's performance. As a general rule, problems with complicating constraints (as P') are suitable to be solved by a delayed variable generation method such as column generation. Analogously, problems with complicating variables (P'') are better suited for employing delayed constraint generation methods such as Benders decomposition.

The development of professional-grade code employing decomposition methods is a somewhat recent occurrence. The commercial solver CPLEX offers a Benders decomposition implementation that requires the user to specify the separable structure. On the other hand, although there are some available frameworks for implementing column generation-based methods, these tend to be more ad hoc occurrences yet often reap impressive results.

12.2 Dantzig-Wolfe decomposition and column generation*

Before we move forward, we need to discuss some technical results that will be useful for describing the Dantzig-Wolfe and Benders decomposition methods. Those are results mostly based on the notion of linear duality from Chapters 5 and 6.

12.2.1 Resolution theorem

A polyhedral set P in standard form can be represented in two manners: either by (i) a finite set of linear constraints; or (ii) by combinations of its extreme points and extreme rays. Clearly, the first representation is far more practical than the second. That is, the second representation is an *explicit* representation that would require knowing beforehand each extreme point and extreme ray forming the polyhedral set.

Notice that the first representation, which we have relied on so far, has extreme points and extreme rays only implicitly represented. However, we will see that this explicit representation has an important application in the devising of alternative solution methods for large-scale linear programming problems. This fundamental result is stated in Theorem 12.1.

Theorem 12.1 (Resolution theorem)**.** *Let $P = \{x \in \mathbb{R}^n : Ax \geq b\}$ be a nonempty polyhedral set with at least one extreme point. Let $\{x_i\}_{i=1}^{k}$ be the set with all extreme points, and $\{w\}_{j=1}^{r}$ be the set of all extreme rays of P. Then $P = Q$, where*

$$Q = \left\{ \sum_{i=1}^{k} \lambda_i x^i + \sum_{j=1}^{r} \theta_j w^j : \lambda_i \geq 0,\ \theta_j \geq 0,\ \sum_{i=1}^{k} \lambda_i = 1 \right\}.$$

Theorem 12.1 has an important consequence, as it states that bounded polyhedra, i.e., a polyhedral set that has no extreme rays, can be represented by the convex hull of its extreme points. For now, let us look at an example that illustrates the concept.

Consider the polyhedral set P given by

$$P = \{x_1 - x_2 \geq -2; x_1 + x_2 \geq 1, x_1, x_2 \geq 0\}.$$

The recession cone $C = \mathbf{recc}(P)$ is described by $d_1 - d_2 \geq 0$, $d_1 + d_2 \geq 0$ (from $Ad = 0$), and $d_1, d_2 \geq 0$, which can be simplified as

$$C = \left\{(d_1, d_2) \in \mathbb{R}^2 : 0 \leq d_2 \leq d_1\right\}.$$

We can then conclude that the two vectors $w^1 = (1, 1)$ and $w_2 = (1, 0)$ are extreme rays of P. Moreover, P has three extreme points: $x_1 = (0, 2)$, $x_2 = (0, 1)$, and $x_3 = (1, 0)$.

Figure 12.1 illustrates what is stated in Theorem 12.1. For example, a representation for the point $y = (2, 2) \in P$ is given by

$$y = \begin{bmatrix} 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix},$$

that is, $y = x^2 + w^1 + w^2$. Notice, however, that y could also be represented as

$$y = \begin{bmatrix} 2 \\ 2 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 0 \\ 1 \end{bmatrix} + \frac{1}{2}\begin{bmatrix} 1 \\ 0 \end{bmatrix} + \frac{3}{2}\begin{bmatrix} 1 \\ 1 \end{bmatrix},$$

with then $y = \frac{1}{2}x^2 + \frac{1}{2}x^3 + \frac{3}{2}w^1$. Notice that this implies that the representation of each point is not unique.

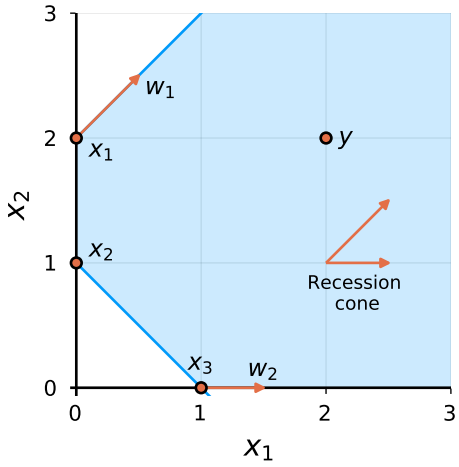


Figure 12.1: Example showing that every point of $P = \{x_1 - x_2 \geq -2; x_1 + x_2 \geq 1, x_1, x_2 \geq 0\}$ can be represented as a convex combination of its extreme point and a linear combination of its extreme rays

12.2.2 Dantzig-Wolfe decomposition

We start with the Dantzig-Wolfe decomposition, which consists of an alternative approach for reducing memory requirements when solving large-scale linear programming problems. Then, we show how this can be expanded further with the notion of delayed variable generation to yield a truly decomposed problem.

As before, let $P_k = \{x_k \geq 0 : D_k x_k = d_k\}$, with $P_k \neq \emptyset$ for $k \in \{1, \ldots, K\}$. Then, the problem P can be reformulated as:

$$\begin{aligned} \text{min.} \quad & \sum_{k=1}^{K} c_k^\top x_k \\ \text{s.t.:} \quad & \sum_{k=1}^{K} A_k x_k = b \\ & x_k \in P_k, \ \forall k \in \{1, \ldots, K\}. \end{aligned}$$

Notice that P has a complicating constraint structure, due to the constraints $\sum_{k=1}^{K} A_k x_k = b$. In order to devise a decomposition method for this setting, let us first assume that we have available for each of the sets P_k, $k \in \{1, \ldots, K\}$, (i) all extreme points, represented by x_k^j, $\forall j \in J_k$; and (ii) all extreme rays w_k^r, $\forall r \in R_k$. As one might suspect, this is in principle a demanding assumption, but one that we will be able to drop later on.

Using the Resolution theorem (Theorem 12.1), we know that any element of P_k can be represented as

$$x_k = \sum_{j \in J_k} \lambda_k^j x_k^j + \sum_{r \in R_k} \theta_k^r w_k^r, \tag{12.1}$$

where $\lambda_k^j \geq 0$, $\forall j \in J_k$, are the coefficients of the convex combination of extreme points, meaning that we also observe $\sum_{j \in J_k} \lambda_k^j = 1$, and $\theta_k^r \geq 0$, $\forall r \in R_k$, are the coefficients of the conic combination of the extreme rays.

Using the identity represented in (12.1), we can reformulate P onto the *main problem* P_M as follows.

$$
\begin{aligned}
(P_M): \min. \quad & \sum_{k=1}^{K} \left(\sum_{j \in J_k} \lambda_k^j c_k^\top x_k^j + \sum_{r \in R_k} \theta_k^r c_k^\top w_k^r \right) \\
\text{s.t.:} \quad & \sum_{k=1}^{K} \left(\sum_{j \in J_k} \lambda_k^j A_k x_k^j + \sum_{r \in R_k} \theta_k^r A_k w_k^r \right) = b && (12.2) \\
& \sum_{j \in J_k} \lambda_k^j = 1, \ \forall k \in \{1, \ldots, K\} && (12.3) \\
& \lambda_k^j \geq 0, \theta_k^r \geq 0, \forall j \in J_k, r \in R_k, k \in \{1, \ldots, K\}.
\end{aligned}
$$

Notice that (12.2) and (12.3) can be equivalently represented as

$$
\sum_{k \in K} \left(\sum_{j \in J_k} \lambda_k^j \begin{bmatrix} A_k x_k^j \\ e_k \end{bmatrix} + \sum_{r \in R_k} \theta_k^r \begin{bmatrix} A_k w_k^r \\ 0 \end{bmatrix} \right) = \begin{bmatrix} b \\ 1 \end{bmatrix},
$$

where e_k is the unit vector (i.e., with 1 in the k^{th} component, and 0 otherwise). Notice that P_M has as many variables as the number of extreme points and extreme rays of P, which is likely to be prohibitively large.

However, we can still solve it if we use a slightly modified version of the revised simplex method. To see that, let us consider that b is a m-dimensional vector. Then, a basis for P_M would be of size $m+K$, since we have the original m constraints plus one for each convex combination (arising from each subproblem $k \in K$). This means that we are effectively working with $(m+K) \times (m+K)$ matrices, i.e., the basic matrix B and its inverse B^{-1}. Another element we need is the vector of simplex multipliers p, which is a vector of dimension $(m+K)$.

The issue with the representation adopted in P_M arises when we are required to calculate the reduced costs of *all* the nonbasic variables, since this is the critical issue for its tractability. That is where the method provides a clever solution. To see that, notice that the vector p is formed by components $p^\top = (q, r_1, \ldots, r_K)^\top$, where q represent the m dual variables associated with (12.2), and r_k, $\forall k \in \{1, \ldots, K\}$, are the dual variables associated with (12.3).

The reduced costs associated with the extreme-point variables λ_k^j, $j \in J_K$, is given by

$$
c_k^\top x_k^j - [q^\top \ r_1 \ \ldots \ r_K] \begin{bmatrix} A_k x_k^j \\ e_k \end{bmatrix} = (c_k^\top - q^\top A_k) x_k^j - r_k. \tag{12.4}
$$

Analogously, the reduced cost associated with extreme-ray variables θ_k^r, $r \in R_k$, is

$$
c_k^\top w_k^r - [q^\top \ r_1 \ \ldots \ r_K] \begin{bmatrix} A_k w_k^r \\ 0 \end{bmatrix} = (c_k^\top - q^\top A_k) w_k^r. \tag{12.5}
$$

The main difference is how we assess the reduced costs of the non-basic variables. Instead of explicitly calculating the reduced costs of all variables, we instead rely on an optimisation-based approach to consider them only *implicitly*. For that, we can use the subproblem

$$
\begin{aligned}
(S_k): \min_x \quad & \overline{c}_k = (c_k^\top - q^\top A_k) x_k \\
\text{s.t.:} \quad & x_k \in P_k,
\end{aligned}
$$

which can be solved in parallel for each subproblem $k \in \{1, \ldots, K\}$. The subproblem S_k is known as the *pricing problem*. For each subproblem $k = 1, \ldots, K$, we have the following cases.

We might observe that $\bar{c}_k = -\infty$. In this case, we have found an *extreme ray* w_k^r satisfying $(c_k^\top - q^\top A_k)w_k^r < 0$. Thus, the reduced cost of the associated extreme-ray variable θ_k^r is negative. If that is the case, we must generate the column

$$\begin{pmatrix} A_k w_k^r \\ 0 \end{pmatrix}$$

associated with θ_k^r and make it enter the basis.

Otherwise, being S_k bounded, i.e., $\bar{c}_k < \infty$, two other cases can occur. The first is the case in which $\bar{c}_k < r_k$. Therefore, we found an extreme point x_k^j satisfying $(c_k^\top - q^\top A_k)x_k^j - r_k < 0$. Thus, the reduced cost associated with the extreme-point variable λ_k^j is negative and, analogously, we must generate the column

$$\begin{pmatrix} A_k x_k^j \\ e_k \end{pmatrix}$$

associated with λ_k^j and make it enter the basis.

The last possible case is when we observe that $r_k < \bar{c}_k < \infty$. In this case, the pricing problem could not identify a beneficial variable to be made basic, and therefore there is not an extreme point or ray with negative reduced cost for subproblem k. If this condition holds for all $k = 1, \ldots, K$, then all necessary extreme points and rays to characterise the region where the optimal extreme point lies (or one of the extreme points, in the case of multiple solutions) have been found and the optimal solution can be recovered.

Algorithm 7 summarises the Dantzig-Wolfe method. The two most remarkable features of the method are (i) the fact that columns are not explicitly represented, but generated "on demand" and (ii) the fact that the pricing problem requires the solution of another linear programming problem. Analogously to the simplex method, it might be necessary to employ a "Phase 1" approach to obtain an initial basis to start the algorithm.

Under a theoretical standpoint, the Dantzig-Wolfe method is equally efficient as the revised simplex method. There are however two settings where the decomposition is most favourable. The first, consists of applications in which the pricing problem can be solved in a closed-form, without invoking a method to solve an additional linear programming subproblem. There are a few examples in which this happens to be the case and certainly many others yet to be discovered.

Secondly, the memory requirements of the Dantzig-Wolfe decomposition makes it an interesting approach for very large-scale problems. The original simplex method requires an amount of memory space that is $O((m + K \times m_0)^2)$, where m_0 is the number of rows of D_k, for $\forall k \in \{1, \ldots, K\}$. This is essentially the size of the inverse basic matrix B^{-1}. In contrast, the Dantzig-Wolfe reformulation requires $O((m + K)^2) + K \times O(m_0^2)$ of memory space, with the first term referring to the main problem inverse basic matrix and the second to the pricing problems basic matrices. For example, for a problem in which $m = m_0$ and much larger than, say, $K = 10$, this implies that the memory space required by the Dantzig-Wolfe reformulation is 100 times smaller, which can substantially enlarge the range of large-scale problems that can be solved for the same amount of computational resources available.

Algorithm 7 Dantzig-Wolfe decomposition

initialise. Let B be a BFS for P_M and set $l \leftarrow 0$.
repeat
  for $k \in \{1, \ldots, K\}$ **do**
    solve S_k and let $\overline{c}_k = \min_x \{S_k\}$
    if $\overline{c}_k = -\infty$ **then**
      obtain extreme ray w_k^r and make $R_k^l = R_k^l \cup \{w_k^r\}$.
      generate column $(A_k w_k^r, 0)$ to become basic.
    else if $\overline{c}_k < r_k < \infty$ **then**
      obtain extreme point x_k^j and make $J_k^l = J_k^l \cup \left\{x_k^j\right\}$.
      generate column $(A_k x_k^j, e_k)$ to become basic.
    end if
  end for
  select one of the generated columns to replace one of the columns of B and update B accordingly.
  $l \leftarrow l + 1$.
until $\overline{c}_k > r_k$ for all $k \in \{1, \ldots, K\}$
return B

12.2.3 Delayed column generation

The term column generation can also refer to a related, and perhaps more widely known, variant of the Dantzig-Wolfe decomposition. In that, the main problem P_M is also repeatedly solved, each time being incremented by an additional variable (or variables) associated with the column(s) identified with negative reduced costs in the pricing problem S_k, $k \in \{1, \ldots, K\}$. This is particularly useful for problems with an exponentially increasing number of variables, or with a large number of variables associated with the complicating constraints (i.e., when m is a large number).

Algorithm 8 Column generation algorithm

1: **initialise.** Let $\tilde{X}_k \subset X_k$, for $k \in \{1, \ldots, K\}$, and set $l \leftarrow 0$.
2: **repeat**
3: solve P_M^l to obtain $\lambda^{*l} = (\lambda_1^{*l}, \ldots, \lambda_K^{*l})$ and duals $(q^{*l}, \left\{r_k^{*l}\right\}_{k=1}^K)$.
4: **for** $k \in \{1, \ldots, K\}$ **do**
5: solve the pricing problem

$$\overline{x}_k^{*l} \leftarrow \operatorname{argmin} \left\{c_k^\top x_k - q^{*l}(A_k x_k) - r_k^{*l} : x_k \in P_k\right\}.$$

6: **if** $\overline{c}_k = c_k^\top \overline{x}_k^{*l} - q^{*l}(A_k \overline{x}_k^{*l}) < r_k^{*l}$ **then** $\tilde{X}_k \leftarrow \tilde{X}_k \cup \left\{\overline{x}_k^{*l}\right\}$
7: **end if**
8: **end for**
9: $l \leftarrow l + 1$.
10: **until** $\overline{c}_k > r_k$ for all $k \in \{1, \ldots, K\}$
11: **return** λ^{*l}.

The (delayed) column generation method is presented in Algorithm 8. Notice in Line 6 the step that is generating new columns in the main problem P_M, represented in the statement $\tilde{X}_k \leftarrow \tilde{X}_k \cup \left\{\overline{x}_k^{*l}\right\}$. That is precisely when new variables λ_k^t are introduced in the P_M with coefficients represented by

the column

$$\begin{pmatrix} c_k \overline{x}_k^{*l} \\ A_k \overline{x}_k^{*l} \\ e_k \end{pmatrix}.$$

Notice that the unbounded case is not treated to simplify the pseudocode, but could be trivially adapted to return extreme rays to be used in P_M, like the previous variant presented in Algorithm 7. Also, notice that the method is assumed to be initialised with a collection of columns (i.e., extreme points) $\tilde{X}_k$, which can normally be obtained from inspection or using a heuristic method.

We finalise showing that the Dantzig-Wolfe and column generation methods can provide information related to its own convergence. This means that we have access to an optimality bound that can be used to monitor the convergence of the method and allow for a preemptive termination given an acceptable tolerance. This bounding property is stated in Theorem 12.2.

Theorem 12.2. *Suppose P is feasible with finite optimal value z. Let $\overline{z}$ be the optimal cost associated with P_M at a given iteration l of the Dantzig-Wolfe method. Also, let r_k be the dual variable associated with the convex combination of the k^{th} subproblem and z_k its optimal cost. Then*

$$z + \sum_{k=1}^{K} (z_k - r_k) \leq z \leq \overline{z}.$$

Proof. We know that $z \leq \overline{z}$, because a solution for P_M is primal feasible and thus feasible for P.

Now, consider the dual of P_M

$$\begin{aligned} (D_M): \ \text{max.} \ & q^\top b + \sum_{k=1}^{K} r_k \\ \text{s.t.:} \ & q^\top A_k x_k^j + r_k \leq c_k^\top x_k^j, \ \forall j \in J_k, \forall k \in \{1, \ldots, K\} \\ & q^\top A_k w_k^r \leq c_k^\top w_k^r, \ \forall r \in R_k, \forall k \in \{1, \ldots, K\} \end{aligned}$$

We know that strong duality holds, and thus $z = q^\top b + \sum_{k=1}^{K} r_k$ for dual variables $(q, r_1, \ldots, r_K)$.

Now, since z_k are finite, we have $\min_{j \in J_k}(c_k^\top x_k^j - q^\top A_k x_k^j) = z_k$ and $\min_{r \in R_k}(c_k^\top w_k^r - q^\top A_k w_k^r) \geq 0$, meaning that $(q, z_1, \ldots, z_K)$ is feasible to D_M. By weak duality, we have that

$$z \geq q^\top b + \sum_{k=1}^{K} z_k = q^\top b + \sum_{k=1}^{K} r_k + \sum_{k=1}^{K} (z_k - r_k) = z + \sum_{k=1}^{K} (z_k - r_k). \qquad \square$$

12.3 Benders decomposition

Benders decomposition is an alternative decomposition method, suitable for settings in which *complicating variables* are present. Differently than the Dantzig-Wolfe decomposition, the method presumes from the get-go the employment of delayed constraint generation.

Benders decomposition has made significant inroads into practical applications. Not only is it extensively used in problems related to multi-period decision-making under uncertainty, but it is also available in the commercial solver CPLEX to be used directly, only requiring the user to indicate (or annotate) which are the complicating variables.

12.3.1 Parametric optimisation problems

Before we proceed, let us develop the idea of having optimisation problems stated as functions of the input data. Specifically, let

$$P(b) = \{x \in \mathbb{R}^n : Ax = b, x \geq 0\}$$

be defined as a function of the input vector $b \in \mathbb{R}^m$. That is, $P(b)$ is the set of feasible solutions when the right-hand side is set to b. Then, let $S = \{b \in \mathbb{R}^m : P(b) \neq \emptyset\}$ be the set of all vectors b for which P has at least one feasible solution. Being so, we can restate the set S as

$$S = \{Ax : x \geq 0\}.$$

That is, the set S is formed by the conic combination of the columns of A for which x is nonnegative (or, in other words, that $P(b)$ is feasible). Also, notice that this is a convex set, as discussed in Section 6.2.1. For any $b \in S$, we can define the function

$$F(b) = \min_x \left\{c^\top x : x \in P(b)\right\}, \tag{12.6}$$

which takes as an argument b and returns the optimal value of the parametrised optimisation problem. Notice that evaluating F requires that an optimisation problem is solved.

Now, let us assume that the dual feasibility set

$$S_D = \left\{p \in \mathbb{R}^m : p^\top A \leq c\right\}$$

is not empty. That implies that $F(b)$ is finite for every $b \in S$ since different b's simply imply that different objective functions $p^\top b$ over S_D are considered. Our main objective here is to understand the structure of the function $F : S \to \mathbb{R}$.

Let $\bar{b} \in S$ be a particular b. Suppose that there exists a nondegenerate optimal BFS x_B with basis B. Then, we have that $x_B = B^{-1}\bar{b}$, $F(\bar{b}) = c_B^\top x_B = c_B^\top B^{-1}\bar{b}$ and that all reduced costs are nonnegative.

Now, if we change $\bar{b}$ to b such that the difference is sufficiently small, $B^{-1}\bar{b}$ remains positive and, consequently, x_B remains a basic feasible solution. Furthermore, since b does not affect the reduced costs (recall that our optimality condition is given by $\bar{c} = c^\top - c_B^\top B^{-1}A \geq 0$), they remain nonnegative. Notice that this is the same argument we used in Section 6.1.3 when we discussed changes in the input data in the context of sensitivity analysis.

The optimal value $F(b)$ associated with this new b sufficiently close to $\bar{b}$

$$F(b) - F(\bar{b}) = c_B B^{-1}(b - \bar{b}) = p^\top (b - \bar{b})$$

where $p = c_B B^{-1}$ is the optimal solution of the dual problem

$$\begin{aligned} &\text{max. } p^\top b \\ &\text{s.t.: } p^\top A \leq c. \end{aligned}$$

This allows us to observe two important characteristics of F. First, in the vicinity of $\bar{b}$, $F(b)$ is a linear function of b, Also, p represents a *gradient* of F. This, again, is an alternative view to the conclusions we drew earlier in Section 6.1.3, and, in a way, further strengthens the idea of the optimal dual variables p representing marginal values regarding changes in the components of b.

With the above, we can formalise an important result regarding the convexity of the function $F(b)$.

Theorem 12.3 (Convexity of $F(b)$)**.** *The optimal value function $F(b)$ is convex in b on the set S.*

Proof. Let $b^i \in S$ and x^i be their associated optimal solution, for $i = 1, 2$. Thus, $F(b^i) = c^\top x^i$, for $i = 1, 2$. Let $\lambda \in [0, 1]$. The vector $\overline{x} = \lambda x^1 + (1 - \lambda)x^2$ is nonnegative, and $A\overline{x} = \lambda b^1 + (1 - \lambda)b^2$. Thus, $\overline{x}$ is a feasible solution when b is set to $\lambda b^1 + (1 - \lambda)b^2$. Therefore,

$$F(\lambda b^1 + (1 - \lambda)b^2) \leq c^\top \overline{x} = \lambda c^\top x^1 + (1 - \lambda)c^\top x^2 = \lambda F(b^1) + (1 - \lambda)F(b^2),$$

which is the definition of a convex function. The first inequality is valid because $\overline{x}$ is a feasible solution when b is set to $\lambda b^1 + (1 - \lambda)b^2$. □

12.3.2 Properties of the optimal value function $F(b)$

Let us again consider the dual problem D

$$\begin{aligned} D : \text{max. } & p^\top b \\ \text{s.t.: } & p^\top A \leq c, \end{aligned}$$

which is again assumed to be feasible. For any b in S, $F(b)$ is finite, and by strong duality (cf. Theorem 5.3), we have that $F(b) = p^\top b$. Because of our outstanding assumption that A has m linearly independent rows (and therefore, m linearly independent columns), Theorem 3.5 guarantees that the feasible region of D has at least one extreme point.

Let us suppose that we know all the extreme points $p^1, \ldots, p^K$ of D. As the optimal solution for D must be an extreme point, we can redefine F as

$$F(b) = \max_{i=1,\ldots,K} (p^i)^\top b, \forall b \in S. \tag{12.7}$$

Specifically, F is the maximum of a finite collection of linear functions, and thus, piecewise linear. Furthermore, within the region where $F(b)$ is linear, or, as we have seen before, the change in b is such that the corresponding optimal basis B of the primal does not change, $F(b) = (p^i)^\top b$ where p^i is the corresponding dual cost.

Finally, we must consider the lack of differentiability of F. Notice that specific values of b will indicate the point at which the optimal basis B changes, which implies a change in p. These "junctions" represent the points at which D has multiple (i.e., not unique) solutions, which, as we have also seen, implies that the primal problem becomes degenerate.

Let us assume that we change b in a particular way, i.e., $b = \overline{b} + \theta d$, where $\theta \in \mathbb{R}$. We can then redefine F by letting

$$f(\theta) = F(\overline{b} + \theta d).$$

From (12.7), we obtain

$$f(\theta) = \max_{i=1,\ldots,K} (p^i)^\top (\overline{b} - \theta d), \overline{b} + \theta d \in S$$

which represents the optimal cost as a function of the scalar θ. In fact, $f(\theta)$ represents a section of the function F in the direction given by the vector d, which is thus also piecewise linear and convex (and can be plotted). Figure 12.2 illustrates the function f projected onto direction d as a function of θ.

To finalise, we return to one remaining issue associated with differentiability. Although p can be seen as a gradient of $F(b)$ at b, we know that F is not differentiable everywhere. To circumvent that, we require a generalisation of the concept of gradients, which is given by the notion of subgradients.

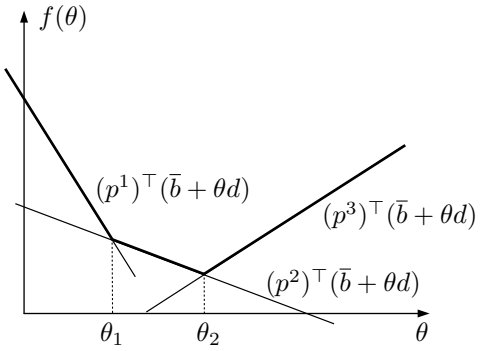


Figure 12.2: The optimal cost function F as a function of b in the direction d. The feasibility set S_D has three extreme points p^1, p^2, and p_3, each associated with a hyperplane $(p^i)^\top(\bar{b}+\theta d)$

Definition 12.4. *Let F be a convex function on the convex set S. Let $\bar{b} \in S$ The vector p is a subgradient of of F at $\bar{b}$ if*

$$F(\bar{b}) + p^\top(b - \bar{b}) \leq F(b), \forall b \in S. \tag{12.8}$$

Figure 12.3 illustrates the notion of a subgradient. Notice that, in general, the subgradient is a singleton, composed only of the gradient (or normal vector) of the hyperplane, which is given by the vector p. At the nondifferentiable points (the junctions), the subgradient comprises all hyperplanes defined between the two hyperplanes intersecting (or all nonnegative linear combinations of the normal vectors of the intersecting hyperplanes). For these values of b, we notice that the dual problem has multiple solutions, implying that the referring BFS to the primal problem is degenerate.

The last result we need is to show that the optimal solution of the dual problem p^* is in fact a subgradient of $F(\bar{b})$ at $\bar{b}$.

Theorem 12.5. *Suppose that the linear programming problem $P = min.\ \{c^\top x : Ax = \bar{b}, x \geq 0\}$ is feasible and the optimal cost is finite. Then, a vector $p \in \mathbb{R}^m$ is an optimal solution to the dual problem if and only if it is a subgradient of the optimal value function F at $\bar{b}$.*

Proof. Recall that F is defined on the set $S = \{b \in \mathbb{R}^m : P(b) \neq \emptyset\}$, and that $P(b) = \{x \in \mathbb{R}^n : Ax = b, x \geq 0\}$. Suppose p is an optimal solution to the dual problem D. Then, strong duality implies that $p^\top \bar{b} = F(\bar{b})$. Consider now an arbitrary $b \in S$. For any feasible solution $x \in P(b)$, we have from weak duality that

$$p^\top b \leq c^\top x \Rightarrow p^\top b \leq \min_{x \in P(b)} c^\top x = F(b).$$

Notice that this implies that $p^\top b - p^\top \bar{b} \leq F(b) - F(\bar{b})$, which in turn yields that p is a subgradient of F at $\bar{b}$ since it rearranges to

$$F(\bar{b}) + p^\top(b - \bar{b}) \leq F(b).$$

Let us consider the converse. Assume that p is a subgradient of F at $\bar{b}$. Thus, we have

$$F(\bar{b}) + p^\top(b - \bar{b}) \leq F(b),\ \forall b \in S. \tag{12.9}$$

Let choose an $x \geq 0$, and let $b = Ax$, meaning that $x \in P(b)$ and that $F(b) \leq c^\top x$. Using (12.9), we obtain

$$p^\top Ax = p^\top b \leq F(b) - F(\bar{b}) + p^\top \bar{b} \leq c^\top x - F(\bar{b}) + p^\top \bar{b}.$$

Since $x \geq 0$, this implies that $p^\top A \leq c$, showing that p is a dual feasible solution. Also, for $x = 0$, we obtain $F(\overline{b}) \leq p^\top \overline{b}$. Now, using weak duality, we have that a dual feasible solution p' satisfies $(p')^\top \overline{b} \leq F(\overline{b})$. Combining the two, we show that p is dual optimal, since

$$(p')^\top \overline{b} \leq p^\top \overline{b}.$$

□

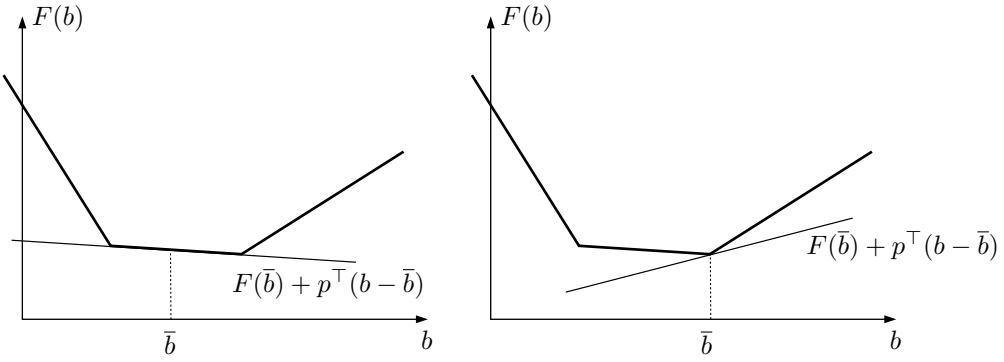


Figure 12.3: The subgradients of the function F at $\overline{b}$. On the left, the unique subgradient of F at $\overline{b}$ is the gradient of the affine function $F(\overline{b}) + p^\top(b - \overline{b})$. On the right, the gradient of the affine function $F(\overline{b}) + p^\top(b - \overline{b})$ at $\overline{b}$ is contained in a subgradient for F at $\overline{b}$.

Therefore, more generally, we can say that at the breakpoints, F has multiple subgradients while everywhere else, the subgradients are unique and correspond to the gradients of F.

12.3.3 Benders decomposition

Let us now return to the Benders decomposition method. Once again, let the problem P be defined as

$$\begin{aligned}(P) : \min_{x,y}\ & c^\top x + \sum_{k=1}^{K} f_k^\top y_k \\ & Ax = b \\ & C_k x + D_k y_k = e_k,\ k \in \{1, \ldots, K\} \\ & x \geq 0, y_k \geq 0,\ k \in \{1, \ldots, K\}\,.\end{aligned}$$

Notice that this is equivalent to the problem P' presented in Section 12.1, but with a notation modified to make it easier to track how the terms are separated in the process. We can see that P has a set of complicating variables x, which becomes obvious when we recast the problem as

$$\begin{array}{ccccccccccc}
c^\top x & + & f_1^\top y_1 & + & f_2^\top y_2 & + & \dots & + & f_k^\top y_k & & \\
Ax & & & & & & & & & = & b \\
C_1 x & + & D_1 y_1 & & & & & & & = & e_1 \\
C_2 x & & & + & D_2 y_2 & & & & & = & e_2 \\
\vdots & & \vdots & & & & \ddots & & & & \vdots \\
C_K x & & & & & & & + & D_k y_k & = & e_K \\
x & & y_1 & & y_2 & & \dots & & y_k & \geq & 0
\end{array}$$

This structure is sometimes referred to as block-angular, referring to the initial block of columns on the left (as many as there are components in x) and the diagonal structure representing the elements associated with the variables y. In this case, notice that if the variable x were to be removed, or fixed to a value $x = \overline{x}$, the problem becomes separable in K independent parts

$$\begin{aligned}
(S_k) : \min_y \ & f_k^\top y_k \\
\text{s.t.: } & D_k y_k = e_k - C_k \overline{x} \\
& y_k \geq 0.
\end{aligned}$$

Notice that these subproblems $k \in \{1, \dots, K\}$ can be solved in parallel and, in certain contexts, might even have analytical closed-form solutions. The part missing is the development of a coordination mechanism that would allow for iteratively updating the solution $\overline{x}$ based on information emerging from the solution of the subproblems $k \in \{1, \dots, K\}$.

To see how that can be achieved, let us reformulate P as

$$\begin{aligned}
(P_R) : \min_x \ & c^\top x + \sum_{k=1}^{K} z_k(x) \\
\text{s.t.: } & Ax = b \\
& x \geq 0.
\end{aligned}$$

where, for $k \in \{1, \dots, K\}$,

$$z_k(x) = \min_y \ \left\{ f_k^\top y_k : D_k y_k = e_k - C_k x \right\}.$$

Notice the resemblance between $z_k(x)$ and the optimal value function $F(b)$ introduced in Section 12.3.1. This is because the subproblems become parametric optimisation problems but as a function of x. That is, in this case, the subproblems are assumed to have $x = \overline{x}$ set as a parameter. Analogously, evaluating $z_k(x)$ requires solving the subproblem S_k, which, in turn, depends on x.

The Benders decomposition works by iteratively constructing the optimal value function. Instead of assuming that all dual extreme points are known, we collect them iteratively, forming an approximation of the optimal value function $z_k(x)$ that becomes increasingly precise as we collect more such extreme points. And, to find new dual extreme points, we can use our current approximation of the optimal value function $z_k(x)$ in P_R, which, once solved, returns us a new $\overline{x}$ to be used for finding a new dual extreme point. Notice that this procedure is akin to iteratively finding the linear segments that form the optimal value functions $z_k(x)$.

To formalise the discussion above, let us first consider the dual formulation of the subproblems $k \in \{1, \dots, K\}$, which is given by

$$\begin{aligned}
(S_k^D) : z_k^D = \max. \ & p_k^\top (e_k - C_k x) \\
\text{s.t.: } & p_k^\top D_k \leq f_k.
\end{aligned}$$

Next, let us denote the feasibility set of S_k^D as

$$P_k = \left\{p : p^\top D_k \leq f_k\right\}, \forall k \in \{1, \ldots, K\}, \tag{12.10}$$

and assume that each $P_k \neq \emptyset$ with at least one extreme point[1]. Relying on the resolution theorem (Theorem 12.1), we know that P_k can be represented by its extreme points p_k^i, $i \in I_k$ and extreme rays w_k^r, $r \in R_k$.

As we assume that $P_k \neq \emptyset$, two cases can occur when we solve S_k^D, $k \in \{1, \ldots, K\}$. Either S_k^D is unbounded, meaning that the relative primal subproblem is infeasible, or S_k^D is bounded, meaning that $z_k^D < \infty$.

From the first case, we can use Theorem 6.6 to conclude that primal feasibility (or a bounded dual value $z_k^D < \infty$) can only be attained if and only if

$$(w_k^r)^\top (e_k - C_k x) \leq 0, \ \forall r \in R_k. \tag{12.11}$$

Furthermore, we know that if S_k^D has a solution, it must lie on a vertex of P_k. So, having available the set of all extreme vertices p_k^i, $i \in I_k$, we have that if one can solve S_k^D, it can be equivalently represented as

$$(S_k^D) : z_k(x) = \max_{i \in I_k} (p_k^i)^\top (e_k - C_k x), \tag{12.12}$$

which can be equivalently reformulated as

$$\text{min.} \ \ \theta_k \tag{12.13}$$

$$\text{s.t.:} \ \theta_k \geq (p_k^i)^\top (e_k - C_k x), \ \forall i \in I_k. \tag{12.14}$$

Again, notice that (12.12) is equivalent to (**??**). Combining (**??**)–(12.14), we can reformulate P_R into a single-level equivalent form

$$\begin{aligned}
(P_R) : \min_x \ & c^\top x + \sum_{k=1}^{K} \theta_k \\
\text{s.t.:} \ & Ax = b \\
& (p_k^i)^\top (e_k - C_k x) \leq \theta_k, \ \forall i \in I_k, \forall k \in \{1, \ldots, K\} && (12.15) \\
& (w_k^r)^\top (e_k - C_k x) \leq 0, \ \forall r \in R_k, \forall k \in \{1, \ldots, K\} && (12.16) \\
& x \geq 0.
\end{aligned}$$

Notice that, just like the reformulation used for the Dantzig-Wolfe method presented in Section 12.2, the formulation of P_R is of little practical use since it requires the complete enumeration of (a typically prohibitive) number of extreme points and rays and is likely to be computationally intractable due to the large number of associated constraints. To address this issue, we can employ delayed constraint generation and iteratively generate only the constraints we observe to be violated. Notice that this can be alternatively interpreted as the idea of iteratively generating the segments of the optimal value function $z_k(x)$.

Following this idea, at a given iteration l, we have at hand a *relaxed main problem* P_M^l, which comprises only some of the constraints associated with the dual extreme points and rays obtained

[1] We have discussed in Section 12.3.2 why we can assume that P_k has at least one extreme point

until iteration l. The relaxed main problem can be stated as

$$\begin{aligned}(P_M^l) : z_{P_M}^l = \min_x.\ & c^\top x + \sum_{k=1}^{K} \theta_k \\ \text{s.t.: } & Ax = b \\ & (p_k^i)^\top (e_k - C_k x) \leq \theta_k,\ \forall i \in I_k^l, \forall k \in \{1, \ldots, K\} \\ & (w_k^r)^\top (e_k - C_k x) \leq 0,\ \forall r \in R_k^l, \forall k \in \{1, \ldots, K\} \\ & x \geq 0,\end{aligned}$$

where $I_k^l \subseteq I_k$, $\forall k \in \{1, \ldots, K\}$ represent subsets of extreme points p_k^i of P_k, and $R_k^l \subseteq R_k$ subsets of extreme rays w_k^r of P_k.

We can iteratively obtain these extreme points and rays from the subproblems S_k, $k \in \{1, \ldots, K\}$. To see that, let us first define that, at iteration l, we solve the main problem P_M^l and obtain a solution

$$\underset{x,\theta}{\operatorname{argmin}} \left\{ P_M^l \right\} = (\overline{x}^l, \overline{\theta}_1^l, \ldots \overline{\theta}_K^l).$$

We can then solve the subproblems S_k^l, $k \in \{1, \ldots, K\}$, for that fixed solution $\overline{x}^l$ and then observe if we can find additional constraints that were to be violated if they had been in the relaxed main problem in the first place. In other words, we can identify if the solution $\overline{x}^l$ allows for identifying additional extreme points p_k^i or extreme rays w_k^r of P_k that were not yet included in P_M^l.

Another way to interpret this notion of violation is to again think of the optimal value function. When we solve S_k^l, for each $k \in \{1, \ldots, K\}$, we are obtaining a "true" (as opposed to approximate) evaluation of the optimal value function $z_k(\overline{x}^l)$ which, when compared against the working approximation of z_k (valued as θ_k) in the main problem, provides a value that is greater than that of θ_k, that is,

$$\theta_k < (p_k^i)^\top (e_k - C_k \overline{x}^l). \tag{12.17}$$

This implies that the approximation of the optimal value function has a segment missing at $\overline{x}^l$, which is precisely the one given by $(p_k^i)^\top (e_k - C_k x)$.

Figure 12.4 illustrates how the optimal value function approximation is iteratively constructed.

To identify those violated constraints, first recall that the subproblem (in its primal form) is given by

$$\begin{aligned}(S_k^l) : \text{min. } & f^\top y \\ \text{s.t.: } & D_k y_k = e_k - C_k \overline{x}^l \\ & y_k \geq 0.\end{aligned}$$

Then, two cases can lead to generating violated constraints that must be added to the relaxed primal problem to form P_M^{l+1}. The first is when S_k^l is feasible. In that case, a dual optimal basic feasible solution p_k^{il} is obtained. If $(p_k^{il})^\top (e_k - C_k \overline{x}^l) > \overline{\theta}_k^l$, then we can conclude that we just formed a violated constraint of the form of (12.15). The second case is when S_k^l is infeasible, then an extreme ray w_k^{rl} of P_k is available, such that $(w_k^{rl})^\top (e_k - C_k \overline{x}^l) > 0$, violating (12.16).

Notice that the above can also be accomplished by solving the dual subproblems S_k^D, $k \in \{1, \ldots, K\}$, instead. In that case, the extreme point p_k^{il} is immediately available, and so are the extreme rays w_k^{rl} in case of unboundedness.

Algorithm 9 presents a pseudocode for the Benders decomposition. Notice that the method can benefit in terms of efficiency from the use of dual simplex, since we are iteratively adding violated

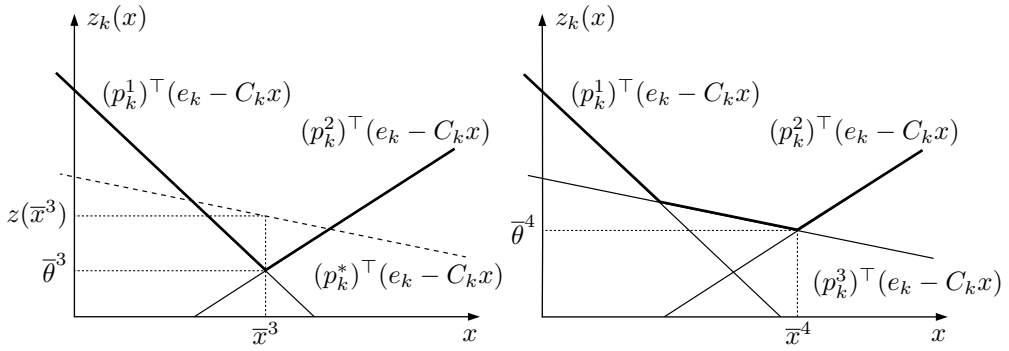


Figure 12.4: $z_k(x)$ is described by 3 line segments. At iteration $l = 3$, two are available. The solution to P_M^3 returns $\overline{\theta}^3$, which is lower than $z(\overline{x}^3)$, obtained solving S_k^D for $\overline{x}^l$. The solution $p_k^* = p_k^3$ from $z(\overline{x}^l)$ defines the missing segment $(p_k^*)^\top(e_k - C_k x)$. A new optimality cut is added, and in iteration $l = 4$, $\overline{x}^4$ is obtained from solving P_M^4. Notice that we would have $\overline{\theta}^4 = z(\overline{x}^4)$, meaning that the algorithm terminates.

constraints to the relaxed main problem P_M^l. Likewise, the dual of the subproblem S_k^l, S_k^{Dl} has only the objective function coefficients being modified at each iteration and, in light of the discussion in Section 6.1.3, can also benefit from the use of dual simplex. Furthermore, the loop represented by Line 4 can be parallelised to provide further computational performance improvements.

Algorithm 9 Benders decomposition

1: **initialise.** Let $P_i^l = W_j^l = \emptyset$, for $k \in \{1, \ldots, K\}$, and set $l \leftarrow 0$.
2: **repeat**
3: solve P_M^l to obtain $\left(\overline{x}^l, \left\{\overline{\theta}_k^l\right\}_{k=1}^K\right)$.
4: **for** $k \in \{1, \ldots, K\}$ **do**
5: solve S_k^{Dl}.
6: **if** S_k^{Dl} is unbounded **then**
7: obtain extreme ray w_j^k and make $W^l = W^l \cup \left\{w_j^k\right\}$.
8: **else**
9: obtain extreme point p_i^k and $P^l = P^l \cup \left\{p_i^k\right\}$
10: **end if**
11: **end for**
12: $l = l + 1$.
13: **until** $(p_k^i)^\top(e_k - C_k\overline{x}) \leq \overline{\theta}_k,\ \forall k \in \{1, \ldots, K\}$
14: **return** $\left(\overline{x}^l, \left\{\overline{\theta}_k^l\right\}_{k=1}^K\right)$

Notice that the algorithm terminates if no violated constraint is found. This in practice implies that $(p_k^i)^\top(e_k - C_k\overline{x}) \leq \overline{\theta}_k$ for all $k \in \{1, \ldots, K\}$, and thus $(\overline{x}, \left\{\overline{\theta}_k\right\}_{k=1}^K)$ is optimal for P. In a way, if one considers the dual version subproblem, S_k^D, one can notice that it is acting as an implicit search for values of p_k^i that can make $(p_k^i)^\top(e_k - C_k\overline{x})$ larger than $\overline{\theta}_k$, meaning that the current solution $\overline{x}$ violates (12.15) and is thus not feasible to P_M^l.

Also, every time one solves P_M^l, a dual (lower for minimisation) bound $LB^l = z_{P_M}^l$ is obtained. This is simply because the relaxed main problem is a relaxation of the problem P, i.e., it contains fewer constraints than the original problem P. A primal (upper) bound can also be calculated at every iteration, which allows for keeping track of the progress of the algorithm in terms of convergence and preemptively terminating it at any arbitrary optimality tolerance. That can be achieved by setting

$$\begin{aligned} UB^l &= \min \left\{ UB^{l-1}, c^\top \overline{x}^l + \sum_{k=1}^{K} f^\top \overline{y}_k^l \right\} \\ &= \min \left\{ UB^{l-1}, z_{P_M}^l - \sum_{k=1}^{K} \overline{\theta}_k^l + \sum_{k=1}^{K} z_k^{Dl} \right\}, \end{aligned}$$

where $(\overline{x}^l, \left\{\overline{\theta}_k^l\right\}_{k=1}^{K}) = \text{argmin}_{x,\theta} \left\{P_M^l\right\}$, $\overline{y}_k^l = \text{argmin}_y \left\{S_k^l\right\}$, and z_k^{Dl} is the objective function value of the dual subproblem S_k^D at iteration l. Notice that, differently from the lower bound LB^l, there are no guarantees that the upper bound UB^l will decrease monotonically. Therefore, one must compare the bound obtained at a given iteration l using the solution $(\overline{x}^l, \overline{y}_1^l, \ldots, \overline{y}_K^l)$ against an incumbent (or best-so-far) bound UB^{l-1}.

12.4 Exercises

Exercise 12.1: Dantzig-Wolfe decomposition

Consider the following linear programming problem:

$$
\begin{array}{lccccccc}
\text{min.} & & -x_{12} & & & -x_{22} & -x_{23} & \\
\text{s.t.:} & x_{11} & +x_{12} & +x_{13} & & & & = 20 \\
 & & & & x_{21} & +x_{22} & +x_{23} & = 20 \\
 & -x_{11} & & & -x_{21} & & & = \text{-}20 \\
 & & -x_{12} & & & -x_{22} & & = \text{-}10 \\
 & & & -x_{13} & & & -x_{23} & = \text{-}10 \\
 & x_{11} & & & & & +x_{23} & \leq 15 \\
 & x_{11,} & x_{12,} & x_{13,} & x_{21,} & x_{22,} & x_{23} & \geq 0
\end{array}
$$

We wish to solve this problem using Dantzig-Wolfe decomposition, where the constraint $x_{11}+x_{23} \leq 15$ is the only "coupling" constraint and the remaining constraints define a single subproblem.

(a) Consider the following two extreme points for the subproblem:

$$x^1 = (20, 0, 0, 0, 10, 10),$$

and

$$x^2 = (0, 10, 10, 20, 0, 0).$$

Construct a main problem in which x is constrained to be a convex combination of x^1 and x^2. Find the optimal primal and dual solutions for the main problem.

(b) Using the dual variables calculated in part a), formulate the subproblem and find its optimal solution.

(c) What is the reduced cost of the variable λ_3 associated with the extreme point x^3 obtained from solving the subproblem in part b)?

(d) Compute a lower bound on the optimal cost.

Exercise 12.2: Parametric optimization

Recall the paint factory problem

$$\begin{aligned}
\text{max. } & z = 5x + 4y && (12.18)\\
\text{s.t.: } & 6x + 4y \leq 24 && (12.19)\\
& x + 2y \leq 6 && (12.20)\\
& y - x \leq 1 && (12.21)\\
& y \leq 2 && (12.22)\\
& x, y \geq 0. && (12.23)
\end{aligned}$$

(a) Formulate a parametric optimization problem $F(x)$, maximizing the profit from selling interior paint (y) given the amount of exterior paint (x) produced. The solution to this problem must be feasible to the full paint factory problem.

(b) Solve the paint factory problem using Benders decomposition. Use x as the main problem variable and y as the subproblem variable.

Exercise 12.3: Benders decomposition

Consider a wholesaler company planning to structure its supply chain to the retailers of a given product. The company needs to distribute the production from many suppliers to a collection of distribution points from which the retailers can collect as much product as they need for a certain period. By default, a pay-as-you-consume contract between wholesaler and retailers is signed and, therefore, the demand at each point is unknown at the moment of shipping. Consider there is no penalty for any unfulfilled demand and any excess must be discarded from one period to the other. The following parameters are given:

- B_i: production cost at supplier i
- C_i: production capacity at supplier i
- D_{js}: total orders from distribution point j in scenario s
- T_{ij}: transportation cost between i and j
- R_j: revenue for sale at distribution point j
- W_j: disposal cost at distribution point j

Let the variables be:

- p_i: production at supplier i
- t_{ij}: amount of products transported between i and j
- l_{js}: amount of products sold from the distribution point j in scenario s
- w_{js}: amount of products discarded from the distribution point j in scenario s
- r_j: amount pre-allocated in the distribution point j

The model for minimising the cost (considering revenue as a negative cost) is given below,

$$
\begin{aligned}
\text{min. } & \sum_{i \in I} B_i p_i + \sum_{i \in I, j \in J} T_{ij} t_{ij} + \sum_{s \in S} P_s \left(\sum_{j \in J} (-R_j l_{js} + W_j w_{js}) \right) \\
\text{s.t.: } & p_i \leq C_i, \ \forall i \in I \\
& p_i = \sum_{j \in J} t_{ij}, \ \forall i \in I \\
& r_j = \sum_{i \in I} t_{ij}, \ \forall j \in J \\
& r_j = l_{js} + w_{js}, \ \forall j \in J, \forall s \in S \\
& l_{js} \leq D_{js}, \ \forall j \in J, \forall s \in S \\
& p_i \geq 0, \ \forall i \in I \\
& r_j \geq 0, \ \forall j \in J \\
& t_{ij} \geq 0, \ \forall i \in I, \forall j \in J \\
& l_{js}, w_{js} \geq 0, \ \forall j \in J, \forall s \in S.
\end{aligned}
$$

Solve an instance of the wholesaler's distribution problem proposed using Benders decomposition.

Bibliography

[1] Dimitris Bertsimas and John N Tsitsiklis. *Introduction to linear optimization*, volume 6. Athena Scientific Belmont, MA, 1997.

[2] John R Birge and Francois Louveaux. *Introduction to stochastic programming*. Springer Science & Business Media, 2011.

[3] Jacek Gondzio. Interior point methods 25 years later. *European Journal of Operational Research*, 218(3):587–601, 2012.

[4] Changhyun Kwon. *Julia Programming for Operations Research*. Changhyun Kwon, 2019.

[5] Hamdy A Taha. *Operations Research: An Introduction (7th Edition)*. Pearson/Prentice Hall, 2003.

[6] H Paul Williams. *Model building in mathematical programming*. John Wiley & Sons, 2013.