Chapter 1

Introduction

1.1 Optimisation

Optimisation is the mathematical procedure for finding the best choice of a set of variables subject to a range of constraints. It is a huge subject, and we will focus on two specific model problems as motivation.

The first is the *diet problem*. We have a population that we want to keep healthy. We assume they are isolated, or resource limited in some way: think of astronauts, or explorers at sea, or a population at war. That means we want to provide food that meets their essential nutritional needs (enough energy, protein, and so on) whilst minimising its cost (which might be in terms of money, or weight, or volume, or something else). The best choice has the smallest cost. The constraints we need to satisfy are the essential nutritional needs.

The second is the *shortest path problem*. We want to travel between two points with the minimum cost. We are thinking of a road or other transport network as the underlying example here. The best choice again has the smallest cost, which again might be in terms of money, or time, or distance. The constraints are the costs needed to go between any two underlying points, or along one single path, on the network.

1.1.1 Purpose of these notes

These notes will give the minimum expected material for this course. They are mainly here to develop the theory and the analysis. For (much) more detail see the books

1. *Operations Research*, Winston, QA264.2 WIN.

2. *Linear Programming*, Chvátal, QA265 CHV.

Neither are needed, but those (and other similar books in the library and elsewhere) will give many additional ideas, examples, and exercises.

1.1.2 Links to other modules

Whilst it may seem that a modules whose purpose is finding extreme values would naturally link with calculus, this is in fact only a small link. The main connection will be to Linear Algebra, particularly the use of matrices and vectors to formulate the problem and for key steps in the arguments. You will need to be comfortable with matrices and matrix operations, linear independence, and basis vectors. If in doubt refer to your Linear Algebra notes.

Chapter 2

Linear programming

2.1 The problem

We start with a minimal *diet problem.* We have three types of food available to us: bread, milk, and eggs. We have three nutrients we need above certain minimum levels: carbohydrates, proteins, and vitamins. We want to minimise the cost of the food we need whilst ensuring we get enough nutrients.

2.1.1 The problem in words

The example numbers we will use are as follows. The unit cost (cost in GB pounds per 100 gram serving) of each food type is

Food	Cost
Bread	2.5
Milk	1.2
Eggs	0.8

The amount (in grams) of each nutrient per unit (100 gram serving) of food is

Food	Carbs	Proteins	Vitamins
Bread	300	5	0.07
Milk	30	50	0.02
Eggs	20	90	0.12

To stay healty, the daily diet should contain at least the minimum quantity of each nutrient, which is

Nutrient	Minimum quantity
Carbs	300
Proteins	600
Vitamins	1

2.1.2 The problem as equations

We will now abstract away the details to construct a problem to solve.

The variables we care about are the number of servings of each food stuff. We denote these x_B, x_M, x_E for the number of servings of bread, milk, and eggs respectively. These are real numbers (we can eat a fraction of a serving), but cannot be negative (we cannot eat a negative amount of food). Hence

$$x_B, x_M, x_E \geq 0. \tag{2.1}$$

The total cost of the diet is the sum of the unit costs multiplied by the number of servings in the diet. This is the *objective function* that we want to minimise. Hence

$$\min 2.5x_B + 1.2x_M + 0.8x_E \tag{2.2}$$

is the total cost to be minimised.

Finally, we have the constraints, which are the nutritional needs for the diet. Each separate nutrient gets its own equation. The total amount of each nutrient in the diet is the amount of nutrients per unit serving multiplied by the number of servings. Therefore the constraints are

$$\begin{array}{rcrcrcll} 300x_B & + & 30x_M & + & 20x_E & \geq & 300 & : \text{minimum carbs,} \\ 5x_B & + & 50x_M & + & 90x_E & \geq & 600 & : \text{minimum proteins,} \\ 0.07x_B & + & 0.02x_M & + & 0.12x_E & \geq & 1 & : \text{minimum vitamins.} \end{array} \tag{2.3}$$

Typically the mathematical problem is summarized as

$$\begin{array}{lrcrcrcl} \min & 2.5x_B & + & 1.2x_M & + & 0.8x_E & & \\ \text{subject to} & 300x_B & + & 30x_M & + & 20x_E & \geq & 300 \\ & 5x_B & + & 50x_M & + & 90x_E & \geq & 600 \\ & 0.07x_B & + & 0.02x_M & + & 0.12x_E & \geq & 1 \\ & x_B, & & x_M, & & x_E & \geq & 0. \end{array} \tag{2.4}$$

2.1.3 A more general form

The specific coefficients are needed to solve the problem, but obscure some of the mathematical structure. We introduce more general notation to make the structure clearer. First, the number of constraints will be denoted m (this is the number of nutrients in this example). Next, the number of foods will be denoted n (this is the number of variables in the objective function). Then the problem can be written as

$$\begin{array}{lrcll} \min & \sum_{j=1}^{n} c_j x_j & & & \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j & \geq & b_j & \forall i = 1, \dots, m \\ & x_j & \geq & 0 & \forall j = 1, \dots, n. \end{array} \tag{2.5}$$

The coefficients are written a_{ij} (the amount of the i^{th} nutrient per unit of the j^{th} food), b_j (the daily required amount of the j^{th} nutrient), and c_j (the cost of the j^{th} food).

2.2 The graphical solution

We will simplify the problem even further to illustrate the key points of the solution method. Restrict to two variables (for example, consider only bread and eggs, as no milk is available). We will look at the simple problem

$$\begin{array}{lrcrcl} \max & 0.02x_A & + & 0.03x_B & & \\ \text{subject to} & x_A & + & x_B & \leq & 10 \\ & x_A & & & \leq & 7 \\ & & & x_B & \leq & 5 \\ & x_A, & & x_B & \geq & 0. \end{array} \tag{2.6}$$

This is a problem with three constraints ($m = 3$) and two variables ($n = 2$), and the coefficients can be read off from the equations (for example, $c = (0.02, 0.03)$). Note that key points of the problem are reversed compared to the diet problem: we are maximising, not minimising, and the constraints are less-than-or-equals rather than greater-than-or-equals.

Each of the constraints describes a line in $\mathbb{R}^2$ which splits the plane into two regions. In one region the constraint is satisfied; in the other it is not. We want to *draw* the region where all the constraints are satisfied. This is the *feasible region.* An example is in figure 2.1.

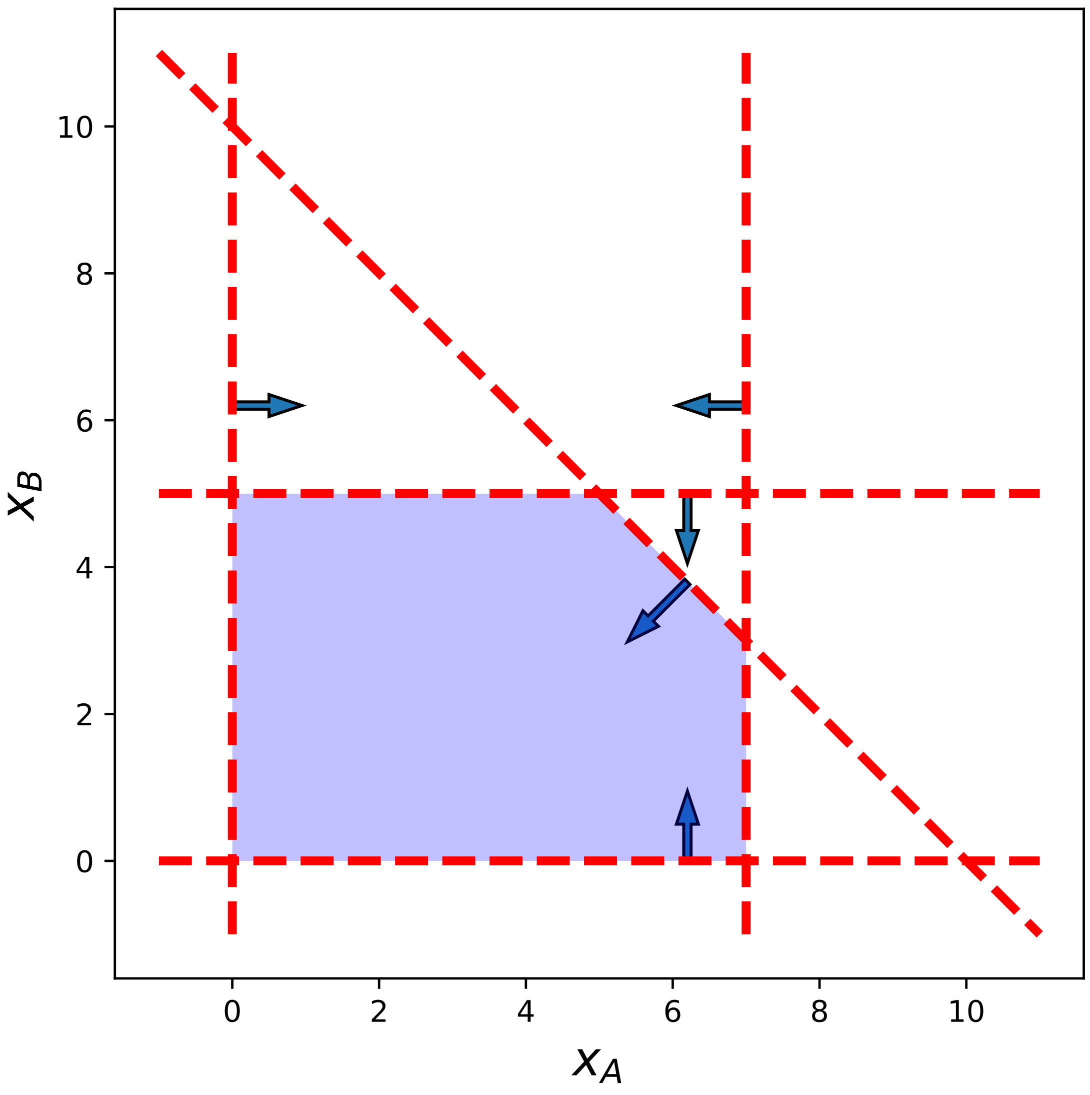

Figure 2.1: The feasible region (shaded blue) constructed from the constraints (red dashed lines, with arrows pointing to which side is feasible) for a problem with two variables x_A, x_B.

We want to find the point in the feasible region where the objective function is maximised (in this case: in the diet problem, we want to minimise). The *level curves* of the objective function are lines where $f(x) = 0.02x_A + 0.03x_B = z$ are constant. These are straight lines which we can also plot. We also need the line that is orthogonal to the level curves: this is the gradient of the objective function ∇f which, in this case, is $(0.02, 0.03)$. For linear problems we generally have $\nabla f = c$.

Our solution method is then as follows. Start from some point within the feasible region (for example, the origin). Move in the direction of the gradient ∇f that increases the objective function z. Draw level curves through this point. Each point that intersects the feasible region is a valid solution to the problem with the value z of the objective function. The last level curve (as we move along the gradient vector) that we can draw will give us the optimal value with its optimal solution x^*. This is illustrated in figure 2.2.

Note that the solution we find has to be at a *corner* or *vertex* of the feasible region. Whilst the graphical solution only works for problems with two variables, the intuition (using the gradient and looking at vertices of the feasible region) extends to the general case.

2.3 Properties

Before we construct the general solution method it is useful to look at the form and structure of the problem. This is because there are multiple ways of looking at the problem, each of which is useful in different cases. There are also rules to transform between the cases.

2.3.1 The different forms

Canonical form

This is the form used so far:

$$\begin{array}{lrcll} \max & \sum_{j=1}^{n} c_j x_j & & & \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j & \leq & b_j & \forall i = 1, \dots, m \\ & x_j & \geq & 0 & \forall j = 1, \dots, n. \end{array} \tag{2.7}$$

This is a maximisation problem, with the inequalities being less-than's.

Standard form

This is a new form:

$$\begin{array}{lrcll} \min & \sum_{j=1}^{n} c_j x_j & & & \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j & = & b_j & \forall i = 1, \dots, m \\ & x_j & \geq & 0 & \forall j = 1, \dots, n. \end{array} \tag{2.8}$$

This is a minimisation problem. We no longer have inequalities, but only equalities.

Simplified canonical form

This is the simplification of the canonical form we have used so far:

$$\begin{array}{lrcll} \max & \sum_{j=1}^{n} c_j x_j & & & \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j & \leq & b_j & \forall i = 1, \dots, m. \end{array} \tag{2.9}$$

The key difference with canonical form is that we do not impose the variables to be non-negative.

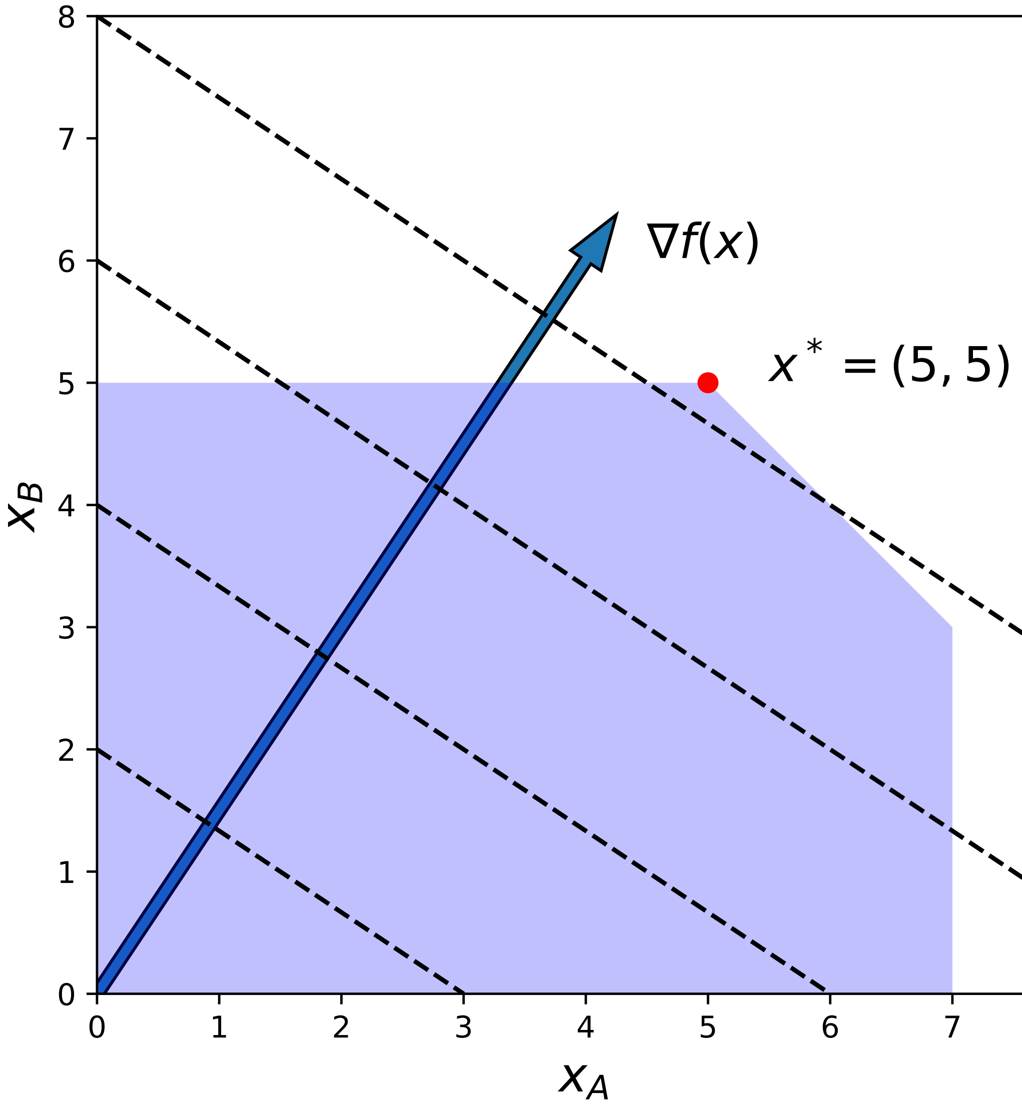

Figure 2.2: The optimal solution (red dot) is the last point in the feasible region (shaded blue) crossed by a level curve (black dashed lines) when moving in the direction of increasing objective function (the arrow pointing in the direction $\nabla f = c$).

Matrix form

This is the canonical form written in the more compact matrix notation:

$$\begin{array}{lrcl} \max & cx & & \\ \text{subject to} & Ax & \leq & b \\ & x & \geq & 0. \end{array} \tag{2.10}$$

Here $A \in \mathbb{R}^{m\times n}$ is a matrix with entries a_{ij}, and $c, x \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$ are vectors with entries c_j, x_j and b_j respectively. Note that in the objective function we have written

$$cx = c \cdot x = \sum_j c_j x_j \tag{2.11}$$

for the inner product between the vector c and the vector x.

This matrix form is useful for its compactness and the links to Linear Algebra concepts it allows us to make.

2.3.2 Transformation rules

There are a range of rules that allow us to transform between, for example, canonical and standard form, or to change a problem in neither standard form into one of the standard forms.

1. Convert between maximisation and minimisation:

$$\max cx \implies -\min(-cx), \tag{2.12}$$
$$\min cx \implies -\max(-cx). \tag{2.13}$$

2. Change direction of an inequality:

$$ax \leq b \implies -ax \geq -b, \tag{2.14}$$
$$ax \geq b \implies -ax \leq -b. \tag{2.15}$$

3. Turn an equation into two inequalities:

$$ax = b \implies ax \geq b \text{ and } ax \leq b. \tag{2.16}$$

4. Turn an inequality into an equation:

   (a) For $\leq$ inequalities, introduce $s \geq 0$, a *slack* variable. Then

$$ax \leq b \implies ax + s = b, \text{ with } s \geq 0. \tag{2.17}$$

   (b) For $\geq$ inequalities, introduce $s \geq 0$, a *surplus* variable. Then

$$ax \geq b \implies ax - s = b, \text{ with } s \geq 0. \tag{2.18}$$

5. Turn any variable x_j without a sign restriction into two variables $x_j^{\pm} \geq 0$, where $x_j = x_j^+ - x_j^-$.

It is important to note that a number of these steps introduce additional variables and constraints. The exact number of variables and constraints are therefore linked to the form in which we write the problem, and are not properties of the problem itself.

2.4 Geometry of linear programming

2.4.1 Setup

For this section we will work with the simplified canonical form in matrix notation,

$$\begin{array}{lccl} \max & cx & & \\ \text{subject to} & Ax & \leq & b \\ & x & & \text{free.} \end{array} \tag{2.19}$$

We can write any non-negativity constraint as $e_{(j)}x \leq 0$, where $e_{(j)}$ is a vector that is zero in all entries except the j^{th}, where it is 1. This additional constraint can be embedded in the matrix form as an additional row in $Ax \leq b$.

2.4.2 Purpose

Start by remembering the graphical solution method. We argued that the optimal solution had to lie on a vertex of the feasible region. We also argued that we could sort these points using the gradient of the objective function.

The purpose of this section is to prove (at some level of rigour) that this intuition extends away two variables, $n = 2$, where the feasible region is a subset of the plane $\mathbb{R}^2$, to the general case where the feasible region is a subset of $\mathbb{R}^n$. That is, we want to show that the optimal solution lies on a vertex of this feasible region.

2.4.3 Fundamental theorem of Linear Programming

Theorem 2.4.1 (Fundamental theorem). *Let $P = \{x \in \mathbb{R}^n \colon Ax \leq b\}$ and consider the linear program $\max\{cx \colon x \in P\}$. If P is non-empty, the linear program either admits an optimal solution x^* corresponding to one of its vertices, or it is unbounded.*

Before we consider the proof let us look at what this says. The region P defined in the theorem is the feasible region. We have constructed the subset of $\mathbb{R}^n$ in which the constraints are satisfied. As noted in the theorem, it may be that there are no points where the constraints are satisfied ("if P is non-empty"): we ignore that case. It is also possible that the feasible region is unbounded and the objective function can increase without limit: this case (where the linear program is unbounded) also needs checking.

To prove this theorem we need to be able to characterise any point in P in terms of its vertices in some way. To do this, we need a number of definitions and preliminary results.

Definitions

Consider a single inequality $a_i x \leq b_i$. This splits $\mathbb{R}^n$ into two pieces. Define $\{x \in \mathbb{R}^n \colon a_i x \leq b_i\}$ to be the *halfspace* of points satisfying the inequality. Further define $\{x \in \mathbb{R}^n \colon a_i x = b_i\}$ to be the *hyperplane* of points satisfying the inequality as an equation. The hyperplane bounds the halfspace. The vector a_i is orthogonal to the hyperplane and points out of the halfspace. This is illustrated in figure 2.3.

The feasible region $P = \{x \in \mathbb{R}^n \colon Ax \leq b\}$ is obtained by intersecting the m halfspaces for each inequality. The intersection of a finite number of halfspaces is called a *polyhedron.* Thus the feasible region is a polyhedron.

Given k points $x_1, \ldots, x_k \in \mathbb{R}^n$, their *convex combinations* are all the points $\{y\}$, with

$$y = \sum_{i=1}^{k} \lambda_i x_i, \tag{2.20}$$

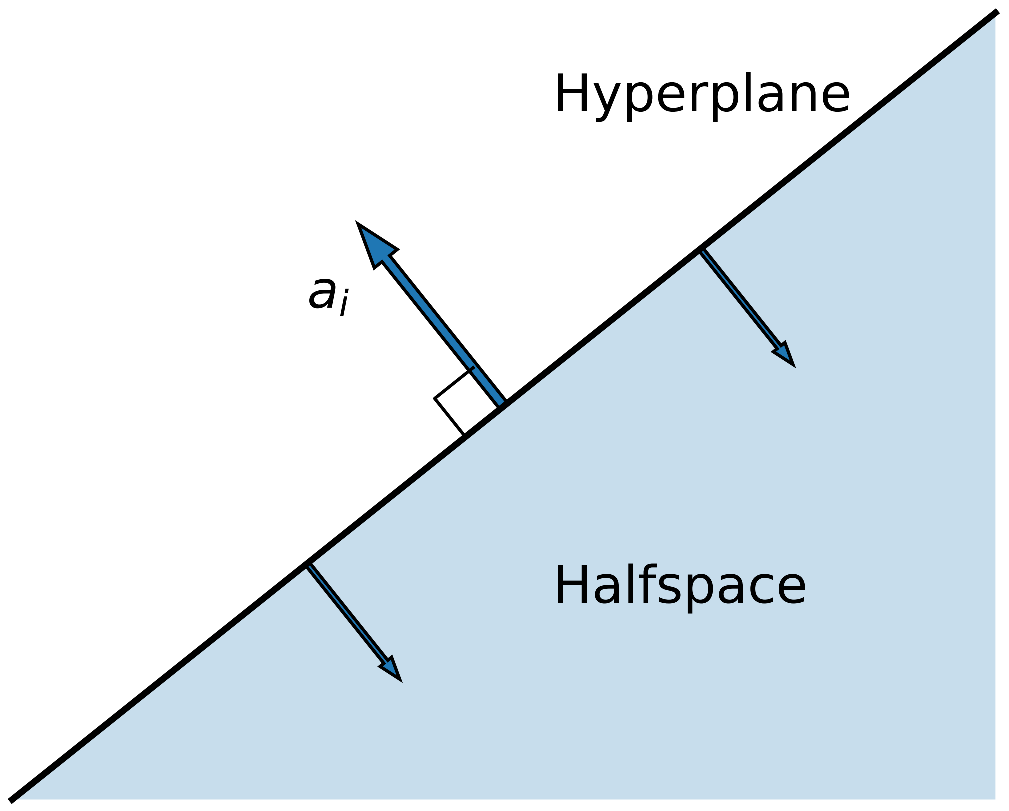

Figure 2.3: A hyperplane splitting $\mathbb{R}^n$ into two pieces. The halfspace (shaded blue) is where the inequality is satisfied. a_i is orthogonal to the hyperplane.

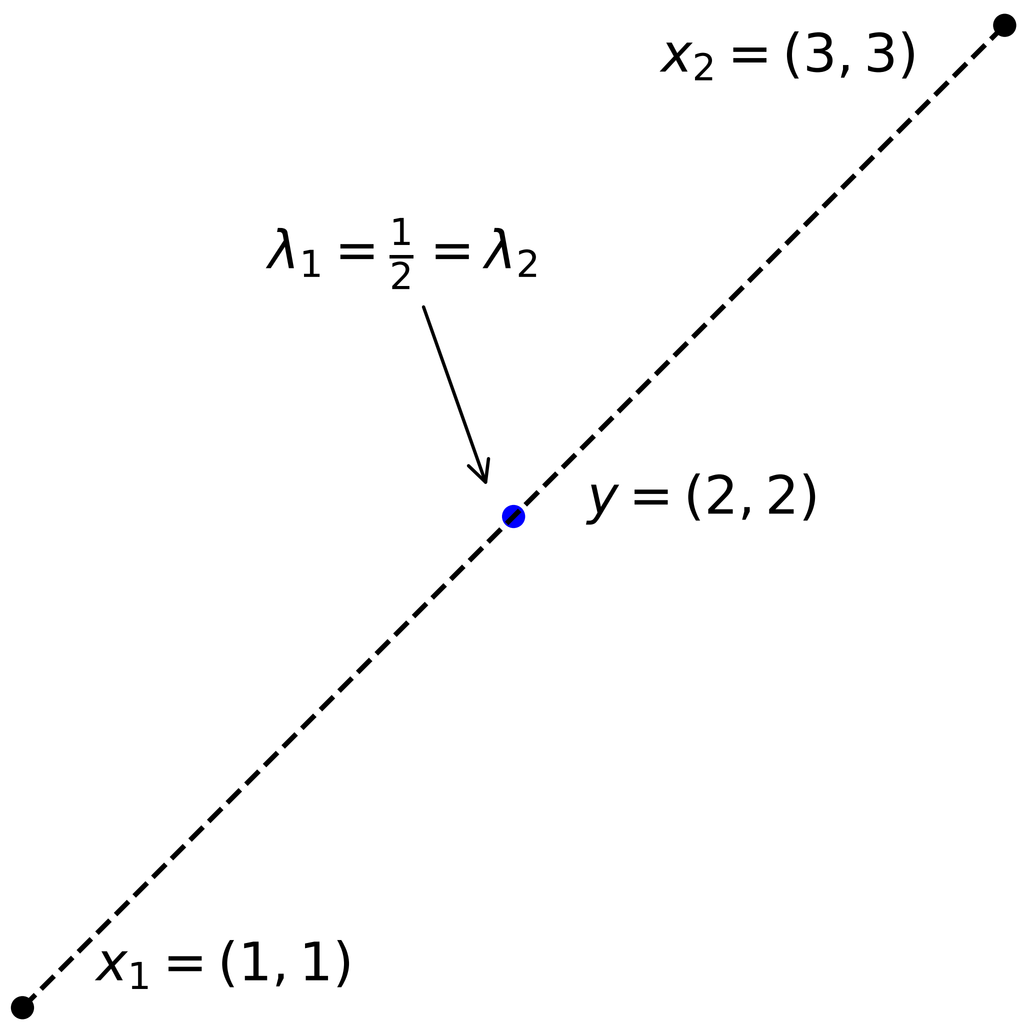

Figure 2.4: All convex combinations of two points lie on the line segment between those two points.

with the restrictions

$$\sum_{i=1}^{k} \lambda_i = 1, \tag{2.21}$$

$$\lambda_1, \ldots, \lambda_k \geq 0. \tag{2.22}$$

Intuitively, if the x_i points are the vertices of a face of the polyhedron, the convex combinations describe all points on the face. Illustrations of convex combinations are in figure 2.4 for two points and in figure 2.5 for three points.

The *conic combinations* are all the points without the restriction that the multipliers λ_i sum to one. Intuitively they generate the hyperplane in full. For two points this is illustrated in figure 2.6.

A set $S \subseteq \mathbb{R}^n$ is *convex* if it contains the convex combinations of all of its elements, that is

$$\lambda x_1 + (1 - \lambda) x_2 \in S \quad \forall x_1, x_2 \in S \text{ and } \forall \lambda \in [0, 1]. \tag{2.23}$$

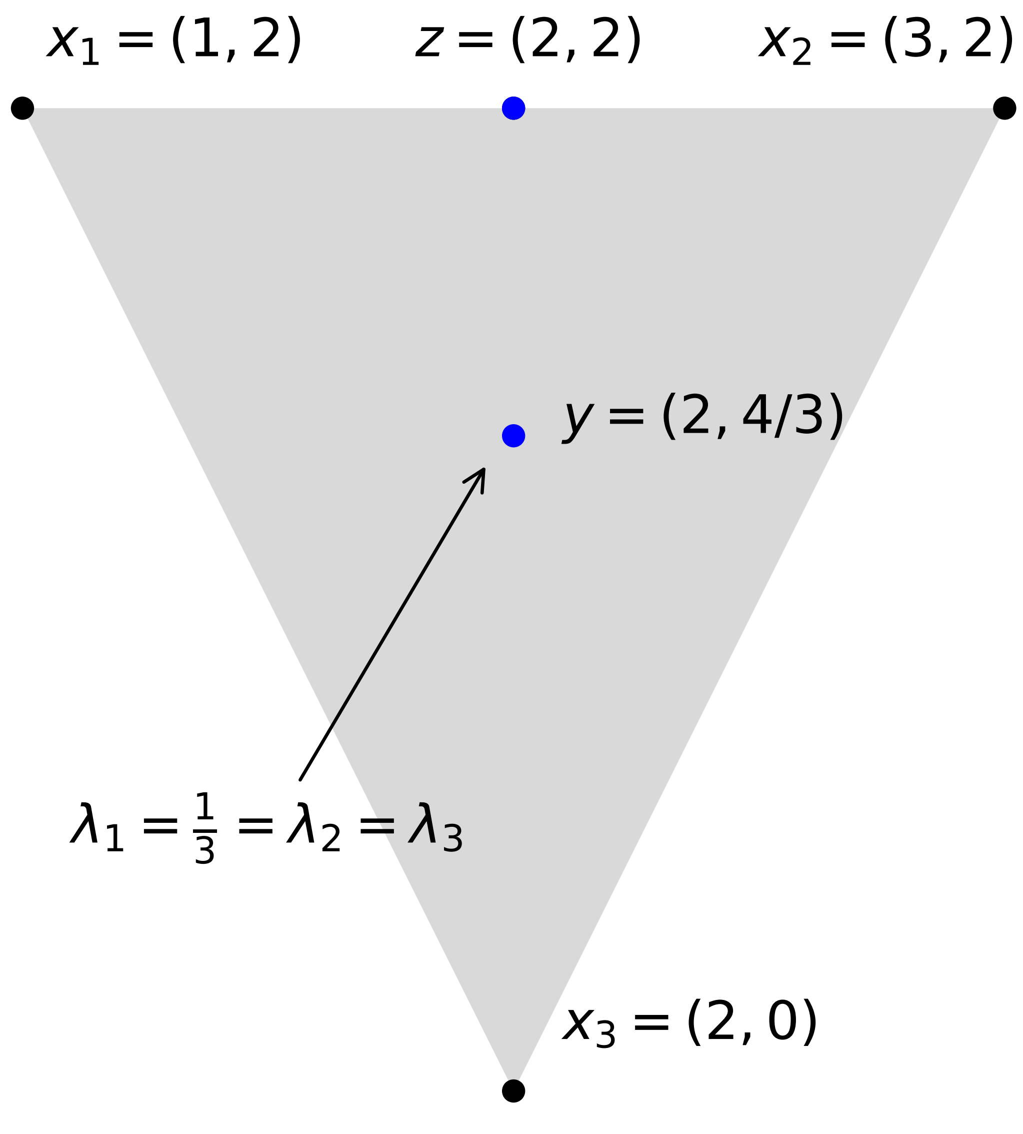

Figure 2.5: All convex combinations of three points form a triangle in a plane within $\mathbb{R}^n$.

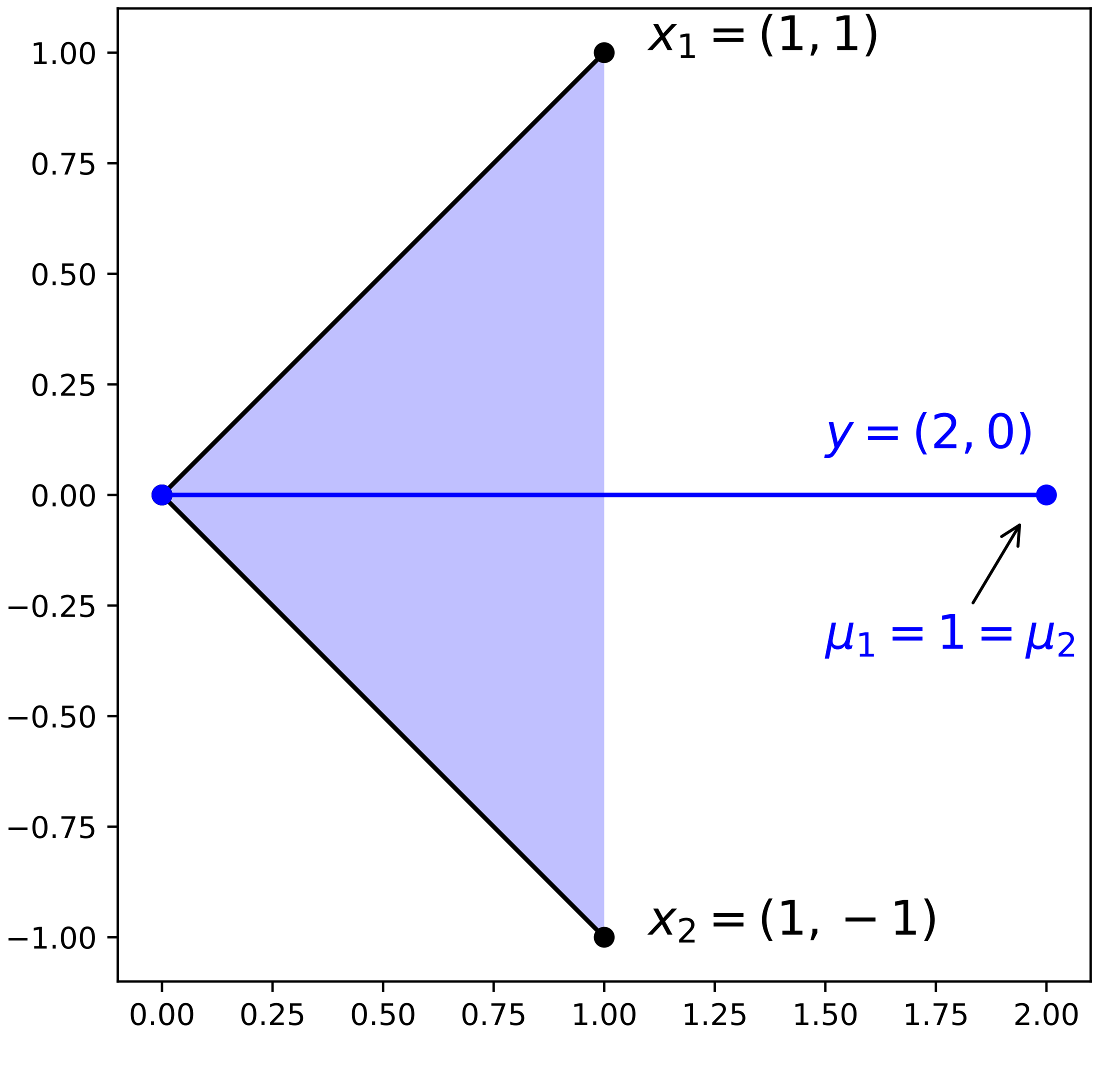

Figure 2.6: The conic combinations of two points generation a quadrant of $\mathbb{R}^2$.

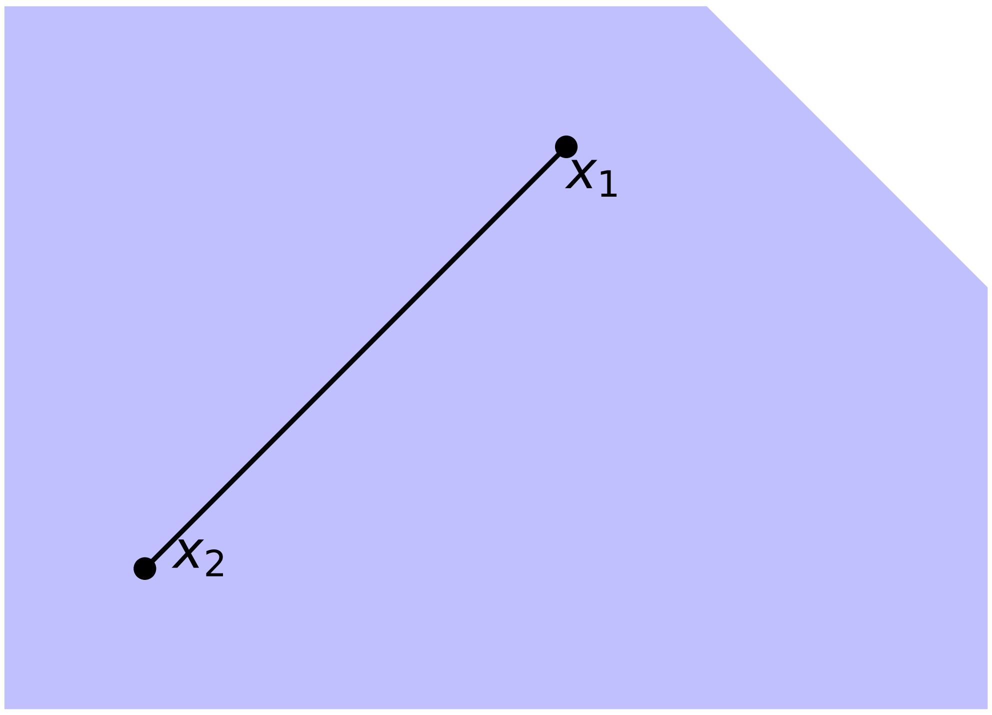

Figure 2.7: A convex set has all points linked by line segments that lie within the set.

We think of this as meaning that the straight line segment linking $x_1, x_2 \in S$ is contained within S. Within the plane $\mathbb{R}^2$ a circle is convex and a "C" shape is not. These are illustrated in figures 2.7 and 2.8 respectively.

For a given polyhedron P, a *vertex* v is a point that cannot be written as a convex combination of two other points x, y in P (with v, x, y all distinct). A *face* is any planar surface belonging to the boundary of P. A *facet* is any $n-1$-dimenional face. This is illustrated in figure 2.9.

For a set $S \subseteq \mathbb{R}^n$, the vector $r \in \mathbb{R}^n$ is a *ray* if

$$x_0 + \mu r \in S \quad \forall x_0 \in S \text{ and } \forall \mu \geq 0. \tag{2.24}$$

This requires that the *semiline* (starting from x_0 and moving in the direction of r) is completely contained in S (for all points $x_0 \in S$).

An *extreme ray* is a ray that cannot be expressed as a conic combination of two other distinct rays p, q, where r, p, q are all distinct. This is illustrated in figure 2.10.

A set is *bounded* if the norm of all vectors in the set is bounded. That is, $\exists \delta \geq 0$ such that $\|x\| \leq \delta \; \forall x \in S$.

We call a bounded polyhedron a *polytope*. This is illustrated in figure 2.11.

2.4.4 Key results

After that blizzard of definitions we can prove some key results.

Theorem 2.4.2. *Polytopes have no rays.*

Proof. If a polytope had a ray it would contain a semiline $x_0 + \mu r$ for all values of μ. By increasing μ we can increase $\|x\|$ without bound. This contradicts that a polytope must be bounded. $\square$

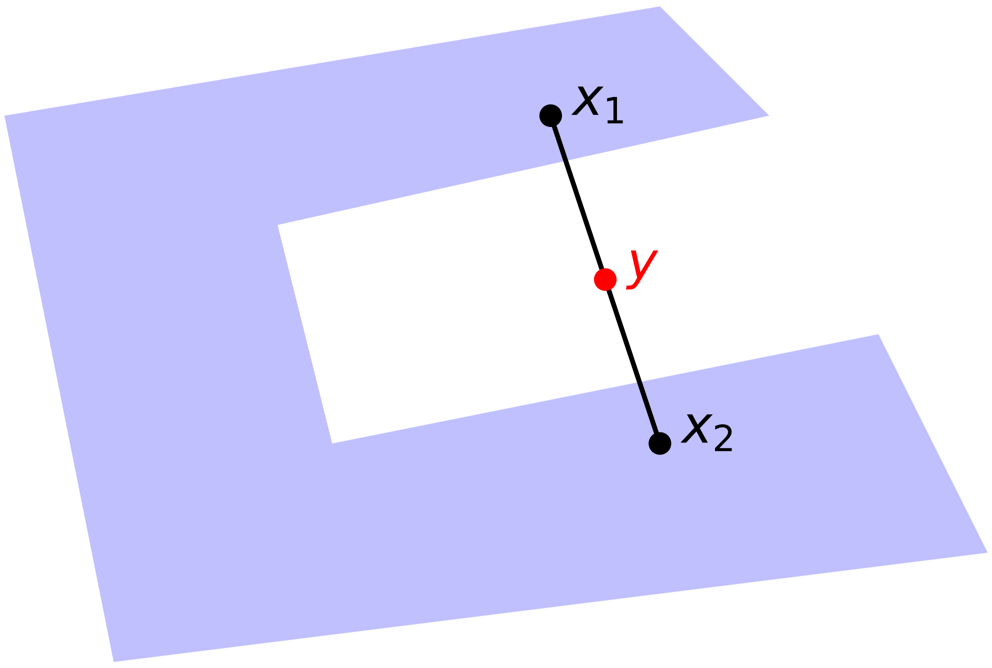

Figure 2.8: A non-convex set has some pair of points that are not linked by a line segment that lies within the set.

Theorem 2.4.3. *The feasible region is convex.*

Proof. Work by contradiction. Assume that P, the polyhedron which is the feasible region, is not convex. Then there exists $x, y \in P$ such that the line segment joining x, y has points not in P. Start at $x \in P$ and move along the line segment until we reach the boundary of P. By construction this point must be in a hyperplane. By definition all points on one side of the hyperplane are in P and all points on the other side are not. As the line segment has not yet left P we have not yet reached y, so y is on the other side of the hyperplane to x. Therefore $y \notin P$. This contradicts our assumption. $\square$

Convexity makes constructing an algorithm to solve the linear program much simpler. Non-convex problems are much harder to solve. However, our main concern is linking the vertices to points in the interior in order to prove the fundamental theorem.

Theorem 2.4.4 (Weyl-Minkowski theorem)**.** *For a subset $P \subseteq \mathbb{R}^n$, the following statements are equivalent:*

- *P is a non-empty polyhedron;*
- *There are two sets, one of $k \geq 1$ vertices $v_1, \ldots, v_k$, and another of $h \geq 0$ extreme rays $r_1, \ldots, r_h$, such that any point $x \in P$ can be written*

$$x = \sum_{i=1}^{k} \lambda_i v_i + \sum_{j=1}^{h} \mu_j r_j. \tag{2.25}$$

The proof of this theorem is complex and not our key purpose. The point, for our purpose, is that it allows us to express any point within P in terms of its vertices. This is illustrated in figure 2.12. This is what we needed to prove the fundamental theorem.

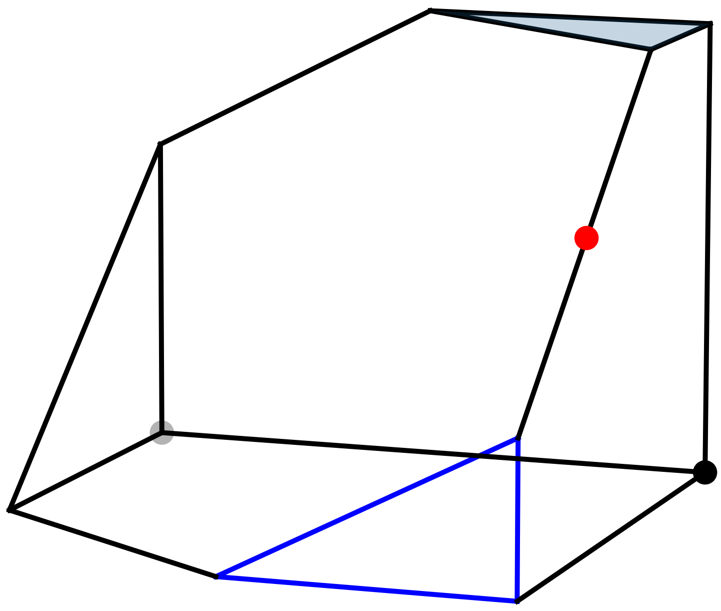

Figure 2.9: A vertex is a corner. The black circles are vertices; the red circle is not. A face is any planar part of the boundary. The blue lines are one dimenional faces; the blue shaded region is a two dimensional face. A facet is an $(n-1)$-dimesional face. The shaded region is a facet as well as a face, whilst the edges are not facets.

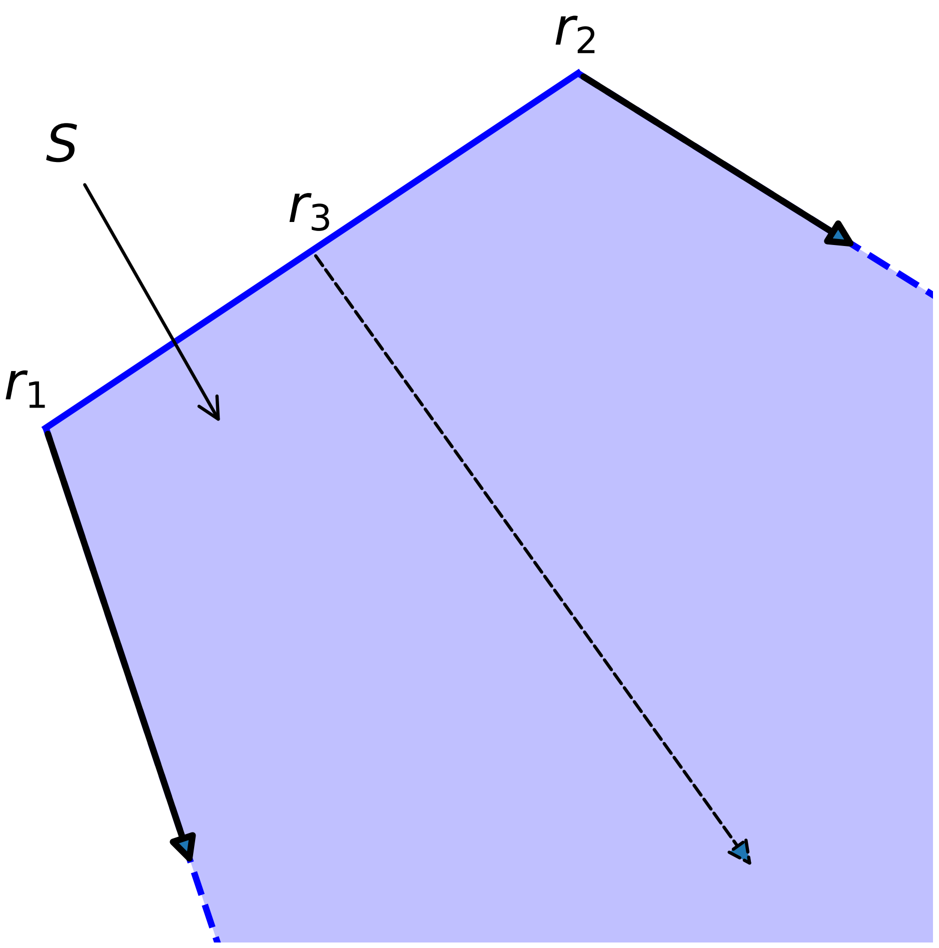

Figure 2.10: The given set S is the blue shaded region, assumed to extend indefinitely down and to the right. The semilines r_1 and r_2 are extreme rays. The semiline r_3 is a ray but not an extreme one.

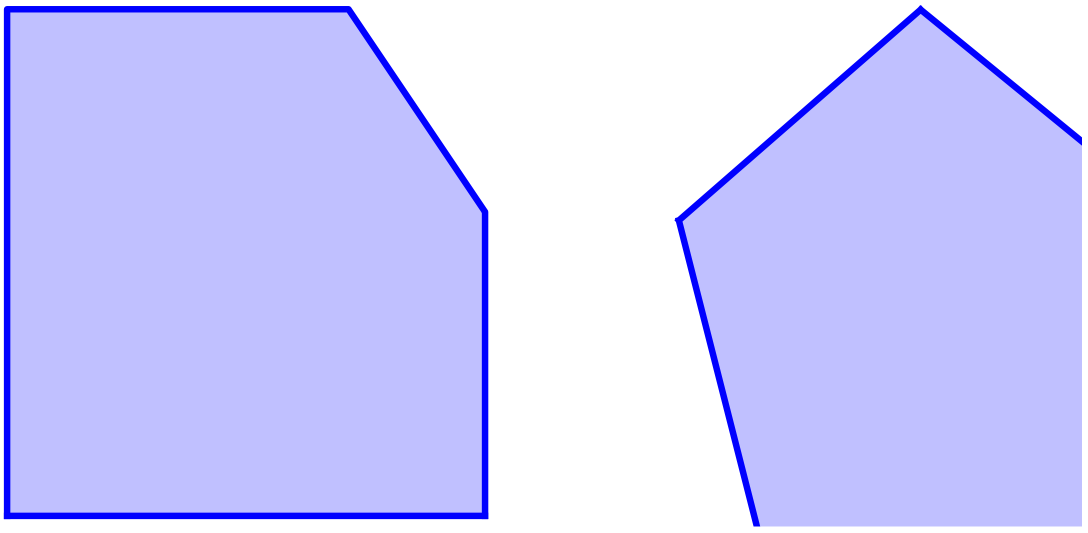
Figure 2.11: The shaded region on the left is bounded by the solid blue lines, and is therefore a polytope. The shaded region on the right is not bounded at the bottom and right and assumed to extend indefinitely. It is therefore not a polytope.

2.4.5 Proof of the Fundamental Theorem

Theorem 2.4.5 (Fundamental theorem)**.** *Let $P = \{x \in \mathbb{R}^n : Ax \leq b\}$ and consider the linear program $\max\{cx : x \in P\}$. If P is non-empty, the linear program either admits an optimal solution x^* corresponding to one of its vertices, or it is unbounded.*

Proof. Let $x \in P$ be a point in the feasible region. Write

$$x = \sum_{i=1}^{k} \lambda_i v_i + \sum_{j=1}^{h} \mu_j r_j \tag{2.26}$$

using the Weyl-Minkowski theorem. The values of λ, μ are constrained to be non-negative, and additionally the values of λ must sum to one.

We now want to *change variables* from x to λ, μ. That is, we want to re-write our linear program in terms of the vector (λ, μ). First re-write the objective function as

$$cx = \sum_{i=1}^{k} \lambda_i (cv_i) + \sum_{j=1}^{h} \mu_j (cr_j). \tag{2.27}$$

This sums over every entry of the vector (λ, μ); the coefficients are the inner product of c with either a vertex v_i or a ray r_j.

Next, write the constraints applied to (λ, μ). These are

$$\sum_{i=1}^{k} \lambda_i = 1, \tag{2.28}$$

$$\lambda_1, \ldots, \lambda_k \geq 0, \tag{2.29}$$

$$\mu_1, \ldots, \mu_h \geq 0. \tag{2.30}$$

We are therefore solving the linear program given by maximsing over all points in the feasible region within (λ, μ)-space.

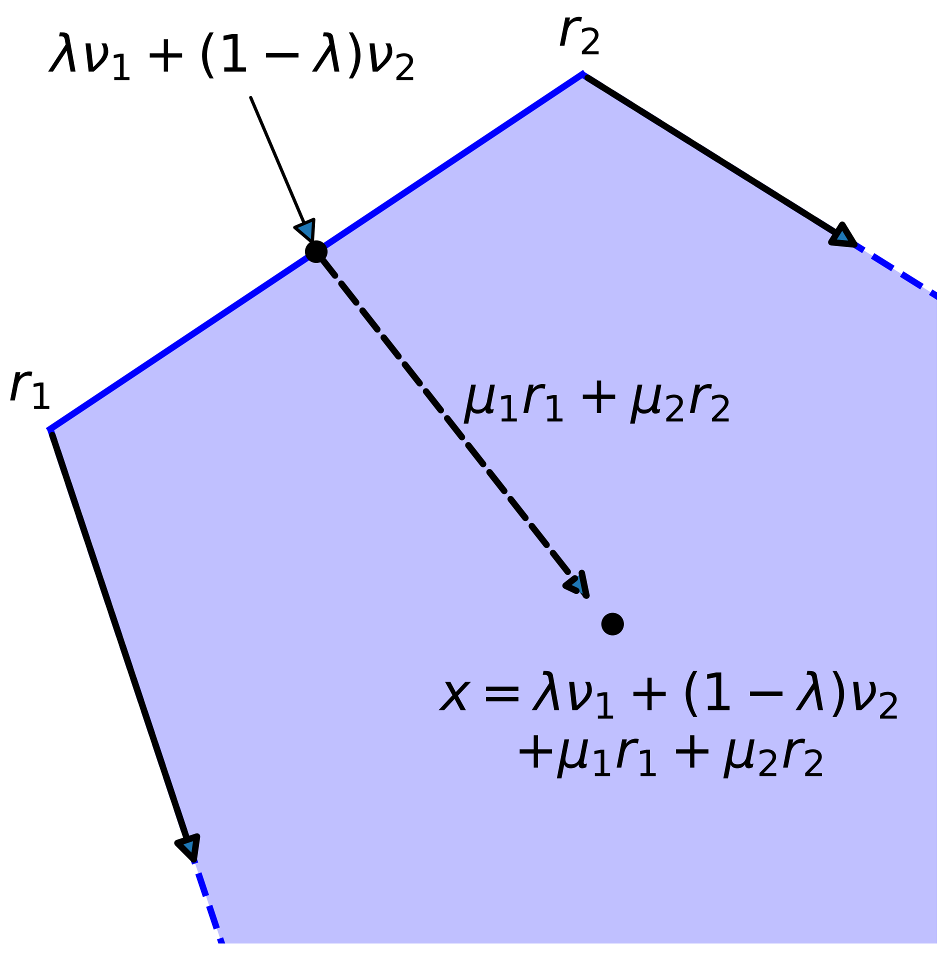
$\lambda\nu_1 + (1-\lambda)\nu_2$
r_2
r_1
$\mu_1 r_1 + \mu_2 r_2$
$x = \lambda\nu_1 + (1-\lambda)\nu_2$
$+\mu_1 r_1 + \mu_2 r_2$

Figure 2.12: The Weyl-Minkowski theorem says that we can express every point within a polyhedron in terms of some set of vertices (here ν_1, ν_2) and extreme rays (here r_1, r_2).

Now look at the objective function. If $cr_j > 0$ for some ray r_j we can increase μ_j, and hence the objective function, arbitrarily. This means the linear program is unbounded.

The alternative is that $cr_j \leq 0$ for all rays. In this case the optimal solution requires $\mu_j = 0$ for all j, as any non-zero coefficient μ_j would only reduce the objective function. Therefore

$$cx^* = \sum_{i=1}^{k} \lambda_i (cv_i) \tag{2.31}$$

and the optimal value has to be a linear combination of the values taken by the objective function at vertices. This gives us a finite number of values to check, and we conclude that

$$cx^* = \max_{i=1,\ldots,k} \{cv_i\}. \tag{2.32}$$

There is, then, an optimal solution corresponding to a vertex v_i. □

2.4.6 Sanity checks

Interior points cannot be optimal

An alternative negative proof can show that interior points cannot be optimal. This does not show the link between vertices and optimal points, but gives more confidence. An illustration is in figure 2.13.

Theorem 2.4.6. *Any point x in the interior of P cannot be optimal.*

Proof. Work by contradiction. Assume there is a point x in the interior of P that is optimal. Construct a ball B of radius $\delta > 0$ around x, $B = \{y \colon \|y - x\| \leq \delta\}$. We need $B \subseteq P$ so that all points are within the feasible region.

Now consider a point $w = x + \epsilon c$ where ϵ is small enough that $w \in B \subseteq P$. As c is the gradient of the objective function it must be the case that (for maximisation problems) the objective function at w is greater than at x. Therefore x is not optimal and we have a contradiction. □

Non-vertices may be optimal

Note that it is possible for *multiple* vertices to have the same, optimal, value of the objective function. In this case points within the facet (in its *relative interior*) linking these vertices will have the same optimal value. This may occur when the facet is orthogonal to the gradient of the objective function, c.

However, this does not matter. The theorem guarantees that *an* optimal solution can always be found at *some* vertex. In this case there are infinite optimal solutions obtained by convex combinations of the vertices of the facet, and all of the vertices are optimal. This is illustrated in figure 2.14.

2.4.7 Brute force solutions

The fundamental theorem therefore gives us a method to find the optimal solution of a linear program. That is, we first identify all vertices of the feasible region. We then evaluate the objective region at each vertex. By enumeration we then find which is optimal.

For example, consider the problem from the graphical solution section,

$$\begin{array}{lrcrcl} \max & 0.02x_A & + & 0.03x_B & & \\ \text{subject to} & x_A & + & x_B & \leq & 10 \\ & x_A & & & \leq & 7 \\ & & & x_B & \leq & 5 \\ & x_A, & & x_B & \geq & 0. \end{array} \tag{2.33}$$

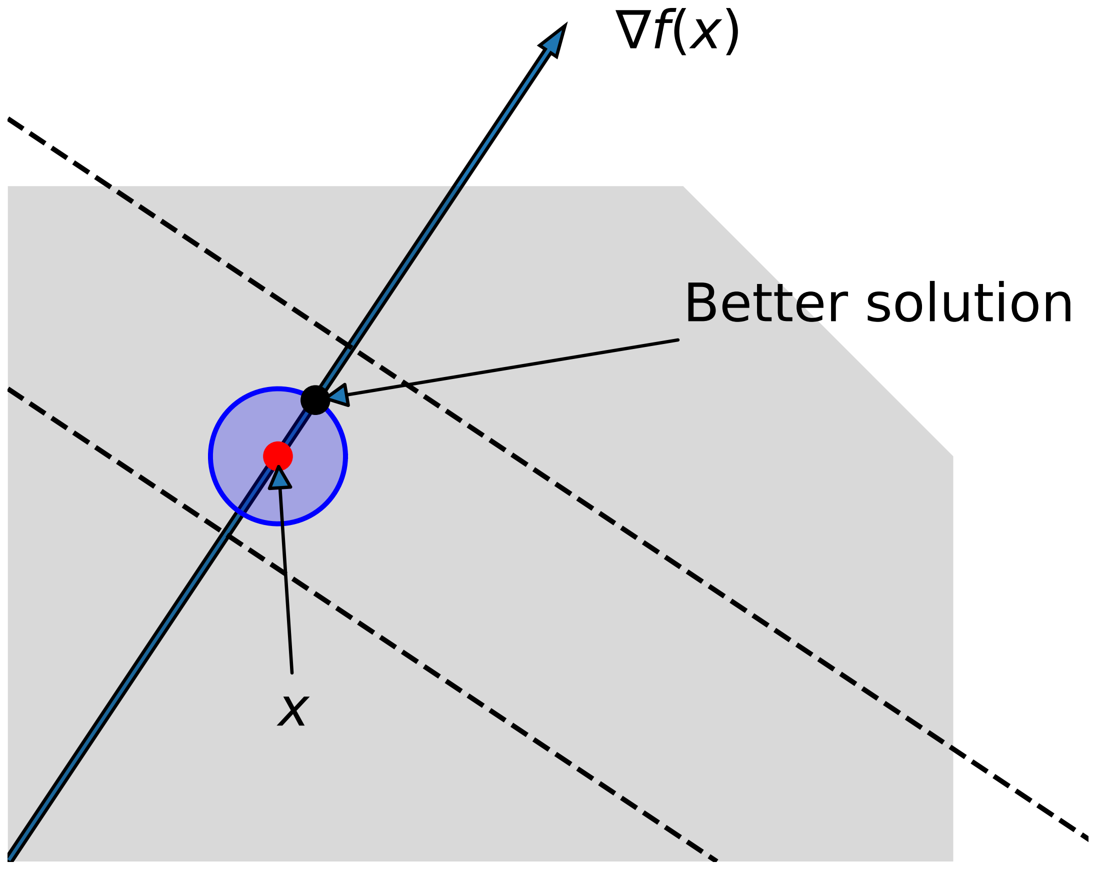

Figure 2.13: An interior point (red dot) of the feasible region (grey shaded region) cannot be optimal, as we can always construct a ball (blue shaded region) around it and improve the value of the objective function (whose level curves are the dashed lines) by moving along ∇f (the arrow).

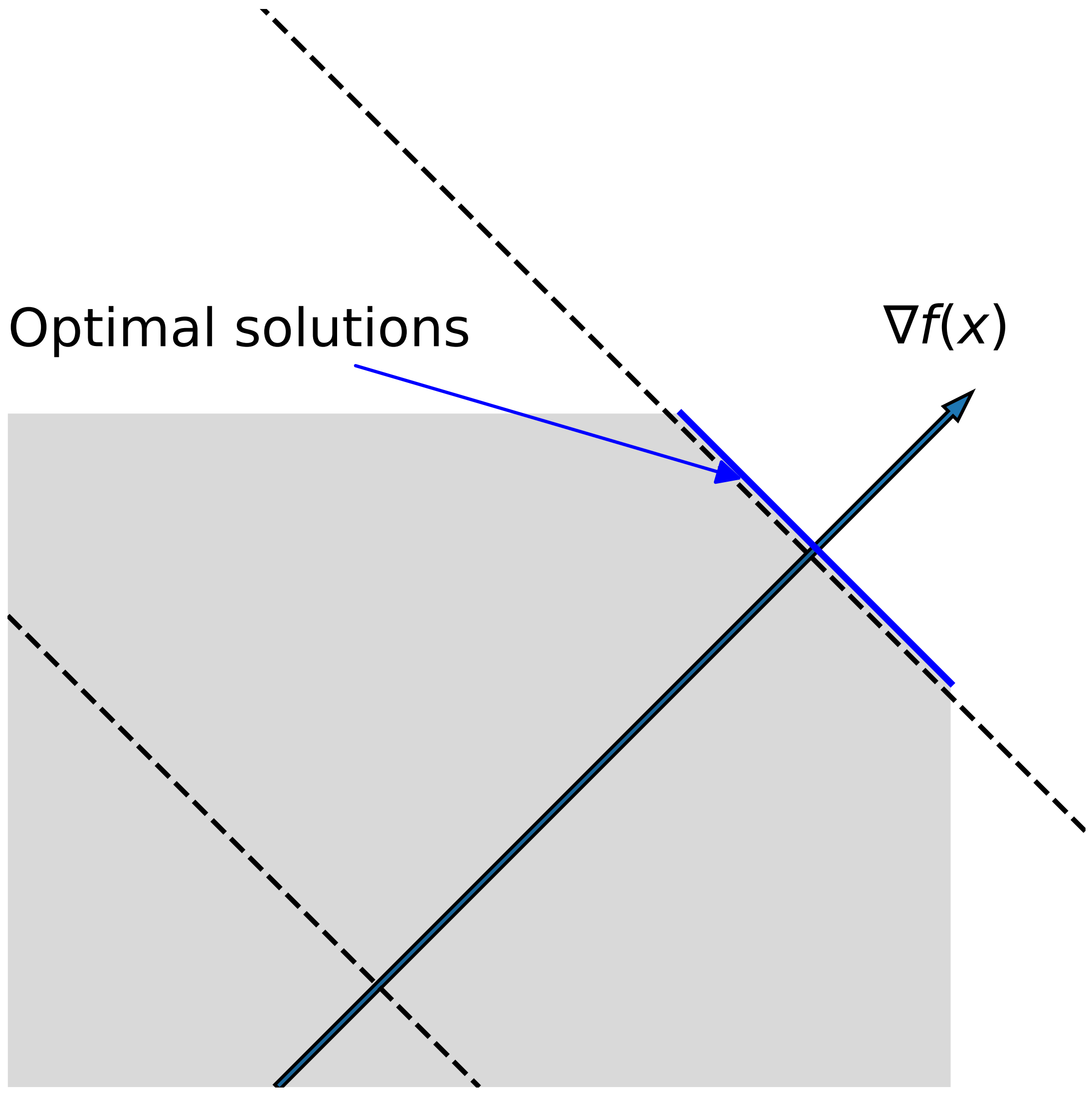

Figure 2.14: Points in the relative interior of a facet (the blue line) can be optimal if the level curve (dashed line) is constant within the facet so that ∇f is orthogonal to it. The value of the objective function is then the same at all points in the facet. If it is optimal, the vertices of the facet are then optimal (as is every point in the facet).

This has five vertices from the intersections of the four constraints.

In general we expect a feasible region with n variables and m constraints to be represented by a polyhedron with approximately

$$\binom{m}{n} = \frac{m!}{n!(m-n)!} \simeq \frac{m^m}{n^n(m-n)^{m-n}} \tag{2.34}$$

vertices. The geometric growth of the number of vertices make enumeration impractical, even with modern computers.

Chapter 3

Sorting

3.1 The motivation

In the previous chapter we saw how a linear program could be solved. We can find every vertex of the feasible region and then evaluate the objective function at all vertices. By sorting this values we then have the optimal solution.

However, we claimed that this method was impractically slow, as the computational cost can grow as m^m (sometimes called exponentially, or geometrically). We should make this explicit. Assume that one operation (finding one vertex and computing the objective function there) takes one microsecond, 1μs. Then compare how long it would take if the computational cost were linear (grows as m), quadratic (grows as m^2), or geometric (grows as 2^m or as m^m) as the number of vertices m increases:

Vertices (m)	Linear (m)	Quadratic (m^2)	2^m	m^m
1	1μs	1μs	1μs	1μs
10	10μs	$10^2\mu$s	$\sim 10^3\mu$s	~ 3 hours
100	100μs	10^{-2}s	$\sim 10^{22}$ years	$\sim 10^{192}$ years

We see that the power law cases (linear and quadratic) both complete in fractions of a second for a hundred vertices, whilst neither geometric case will complete within the lifetime of the universe. Note that Chvátal, writing around 1980, comments that problems with $m \geq 10^3$ were being solved routinely. In modern usage linear programs with $m \geq 10^6$ are being solved in under a minute by the best solvers.

3.2 The problem

The problem, for this chapter, is to understand how to analyse an algorithm. Once we can measure how fast, in principle, an algorithm completes its task, we can use this to compare different techniques.

We will not apply this analysis to linear programming algorithms at first, as there is a simpler case at hand. That is the final step in our impractical algorithm for linear programs: sorting a list of numbers.

3.2.1 Sorting

The *sorting problem* is to take a list a of objects with a *partial order* $\leq$, sort the list into non-decreasing order.

The type of the objects is left unspecified. They could be real numbers or integers, which will be our standard examples. They could be letters or words. The key point is that it must be possible to compare any two objects in the list to give an order. That is, $\forall x, y \in a$, the comparison

$x \leq y$ must be either true or false. A list containing all integers is fine. A list containing a mix of letters and numbers is not (without further assumptions). A list containing complex numbers is not fine either.

We assume the list has length n and refer to the individual entries in the list as a_i, where $i \in \{1, \ldots, n\}$ is the *index*. We are looking to construct a *permutation operator* $\pi \colon \{1, \ldots, n\} \to \{1, \ldots, n\}$ which is a bijective function on the indexes. The permutation operator tells us where to go in the original list to find the object at a given position in the sorted list.

Formally, the problem is to find a permutation π such that

$$a_{\pi(i)} \leq a_{\pi(i+1)} \quad \forall i \in \{1, \ldots, n\}. \tag{3.1}$$

As a concrete example, consider the list $a = (7, 2, 2, 6, 1, 4)$. By eye we see the sorted list is $s = (1, 2, 2, 4, 6, 7)$. Looking at the indices we construct the permutation:

$$\begin{array}{ccllllll} a & : & 7 & 2 & 2 & 6 & 1 & 4 \\ \text{index} & : & 1 & 2 & 3 & 4 & 5 & 6 \end{array} \to \begin{array}{ccllllll} s & : & 1 & 2 & 2 & 4 & 6 & 7 \\ \pi(\text{index}) & : & 5 & 2 & 3 & 6 & 4 & 1 \end{array} \tag{3.2}$$

3.3 Algorithms

We look at three different sorting algorithms before introduing a detailed analysis.

3.3.1 Brute Force Sort

This is a two step process.

1. Enumerate all possible permutations π of length n.
2. Return the first permutation giving a sorted list.

This is closely linked to the impractical algorithm introduced to solve the linear programming problem.

It is possible that only one permutation sorts the list, and that (by bad luck) it is the last one constructed. Given a list with n items we can construct $n!$ permutations of that list (consider how many positions the first object can take, then, having fixed the first, how many the second can take, and so on). For large n we have that $n! \sim n^n$, and we have seen how rapidly that grows.

3.3.2 Selection sort

This is an iterative process.

1. Find the smallest object.
2. Swap it with the object in position one (if strictly smaller).
3. Find the next smallest object in the list; alternatively, consider the list *except* the first object.
4. Swap it with the object in position two (if strictly smaller).
5. Continue to the end of the list.

When implemented in code this algorithm requires two loops. The outer loop runs over the whole list and corresponds to the position we are going to swap the "smallest" object into. The inner loop is to find the smallest object, and runs over only part of the list.

The steps for $a = (7, 2, 2, 6, 1, 4)$ are as follows. When we find the smallest object it will be in the sublist denoted by square brackets:

1. (a) : Smallest object in $([7, 2, 2, 6, 1, 4])$ is 1 at index 5.
   (b) : Swap with object at index 1, giving $(1, 2, 2, 6, 7, 4)$.

2. (a) : Smallest object in $(1, [2, 2, 6, 7, 4])$ is 2 at index 2.
   (b) : Do not swap as already at index 2, giving $(1, 2, 2, 6, 7, 4)$.
3. (a) : Smallest object in $(1, 2, [2, 6, 7, 4])$ is 2 at index 3.
   (b) : Do not swap as already at index 3, giving $(1, 2, 2, 6, 7, 4)$.
4. (a) : Smallest object in $(1, 2, 2, [6, 7, 4])$ is 4 at index 6.
   (b) : Swap with object at index 4, giving $(1, 2, 2, 4, 7, 6)$.
5. (a) : Smallest object in $(1, 2, 2, 4, [7, 6])$ is 6 at index 6.
   (b) : Swap with object at index 5, giving $(1, 2, 2, 4, 6, 7)$.
6. We have reached the last object, so the list is sorted.

3.3.3 Insertion sort

Conceptually this process splits the list a into two pieces, or sublists. One is sorted and the other unsorted. We write $a = (s|u)$ where $|$ marks the separation between the sublists. At the start s is empty and $u = a$. Then, while u is not empty, we

1. Compare the first entry of the unsorted list u with each entry of the sorted list s in turn *from the right*.
2. If $u_1 < s_i$, swap the entry in the sorted list with the first entry in the unsorted list. Do this for each entry in the sorted list until u_1 is in order.
3. Move the separation marker one entry to the right (so that s_i, which was swapped with u_1, is still in the sorted list).

As with selection sort, when implemented in code we have two loops, one running over the whole list and one over some part of the list. We may expect the computational cost of the two to be similar.

The steps for $a = (7, 2, 2, 6, 1, 4)$ are as follows.

1. (a) $(|7, 2, 2, 6, 1, 4)$ has nothing in the unsorted list.
   (b) Move the separation marker to the right, giving $(7|2, 2, 6, 1, 4)$.
2. (a) As $2 < 7$, swap s_1 and u_1, giving $(2|7, 2, 6, 1, 4)$.
   (b) Move the separation marker to the right, giving $(2, 7|2, 6, 1, 4)$.
3. (a) As $2 < 7$, swap s_2 and u_1, giving $(2, 2|7, 6, 1, 4)$.
   (b) As 2 is not less than 2 no further sorting is needed.
   (c) Move the separation marker to the right, giving $(2, 2, 7|6, 1, 4)$.
4. (a) As $6 < 7$, swap s_3 and u_1, giving $(2, 2, 6|7, 1, 4)$.
   (b) As 6 is not less than 2 no further sorting is needed.
   (c) Move the separation marker to the right, giving $(2, 2, 6, 7|1, 4)$.
5. (a) As $1 < 7$, swap s_4 and u_1, giving $(2, 2, 6, 1|7, 4)$.
   (b) As $1 < 6$, swap s_3 and (what was) u_1, giving $(2, 2, 1, 6|7, 4)$.
   (c) As $1 < 2$, swap s_2 and (what was) u_1, giving $(2, 1, 2, 6|7, 4)$.
   (d) As $1 < 2$, swap s_1 and (what was) u_1, giving $(1, 2, 2, 6|7, 4)$.
   (e) Move the separation marker to the right, giving $(1, 2, 2, 6, 7|4)$.
6. (a) As $4 < 7$, swap s_5 and u_1, giving $(1, 2, 2, 6, 4|7)$.

(b) As $4 < 6$, swap s_4 and (what was) u_1, giving $(1, 2, 2, 4, 6|7)$.

(c) As 4 is not less than 2 no further sorting is needed.

(d) Move the separation marker to the right, giving $(1, 2, 2, 4, 6, 7|)$.

7. The unsorted list is empty, so the sorted list is complete.

3.4 Analysis of algorithms

We have three different algorithms to sort a list, and a rough idea of how they will behave. However, that rough idea could easily be wrong. If the first permutation generated by the brute-force method sorts the list then it is faster than either selection or insertion sort.

Equally, it is generally impractical to work out how long a sorting algorithm may take in all cases and describe the result statistically, as the time taken will diverge to infinity with the size of the list.

Instead, we will look at the *worst case*: what is the maximum length of time an algorithm may take?

3.4.1 Definitions

An *elementary operation* (EO) is any simple operation an algorithm may take with roughly the same cost. Examples would be comparisons, swaps, additions, multiplications, or assigning a value to a variable.

The *instance size* is a number n that characterises the size of the problem. When sorting it would be the length of the list a.

In the *worst case* analysis we reduce every choice that the algorithm makes to the one needing the most elementary operations.

Useful results

The *Gauss series* or arithmetic progression formula is

$$1 + 2 + \cdots + n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}. \tag{3.3}$$

The size of a set of consecutive integers, $\{i, i + 1, \ldots, j\}$ is

$$|\{i, i + 1, \ldots, j\}| = j - i + 1. \tag{3.4}$$

3.4.2 Selection Sort

```
1 for i in range(0, n-1):
2     j = i
3     for k in range(i+1, n):
4         if a[k] < a[j]:
5             j = k
6     if a[j] < a[i]:
7         a[i], a[j] = a[j], a[i]
```

This Python code performs selection sort.

- Line 1 loops $n - 1$ times over:
  - Line 2 assigns one value (1 EO);
  - Line 3 loops $n - (i + 1) + 1$ times over:

    * Line 4 does one comparison (1 EO);
    * Line 5 assigns one value (1 EO);

  The total cost of the loop is $2(n-i)$ EOs;

  - Line 6 does one comparison (1 EO);
  - Line 7 does one comparison (1 EO);

The total cost of the loop is

$$\sum_{i=1}^{n-1}(1+2(n-i)+2) = \sum_{i=1}^{n-1}(2n+3) - 2\sum_{i-1}^{n-1} i \tag{3.5}$$

$$= (n-1)(2n+3) - 2\frac{(n-1)n}{2} \tag{3.6}$$

$$= (n-1)(n+3) = n^2+2n-3. \tag{3.7}$$

We see that the analysis gives the worst case of selection sort as $\sim n^2$. As shown earlier this grows much more slowly that the worst case $\sim n^n$ for the brute-force algorithm. Selection sort may be practical.

3.4.3 Insertion Sort

```
1 for i in range(1, n):
2     for j in range(i, 0, -1):
3         if a[j] < a[j-1]:
4             a[i], a[j] = a[j], a[i]
5         else:
6             break
```

This Python code performs insertion sort.

- Line 1 loops $n-1$ times over:
  - Line 2 loops i times over:
    * Line 3 does one comparison (1 EO);
    * Line 4 does one swap (1 EO);
    * Lines 5 and 6 never occur in the worst case.

    The total cost of the loop is $2(i-i)$ EOs;

  The total cost of the loop is

$$\sum_{i=2}^{n} 2(i-1) = \sum_{i=1}^{n} 2(i-1) \tag{3.8}$$

$$= 2\sum_{i=1}^{n} i - 2\sum_{i=1}^{n} 1 \tag{3.9}$$

$$= \frac{n(n+1)}{2} - 2n \tag{3.10}$$

$$= n^2 - n. \tag{3.11}$$

We see that the analysis gives the worst case of insertion sort as $\sim n^2$. For large n we expect insertion sort to be slightly faster than selection sort.

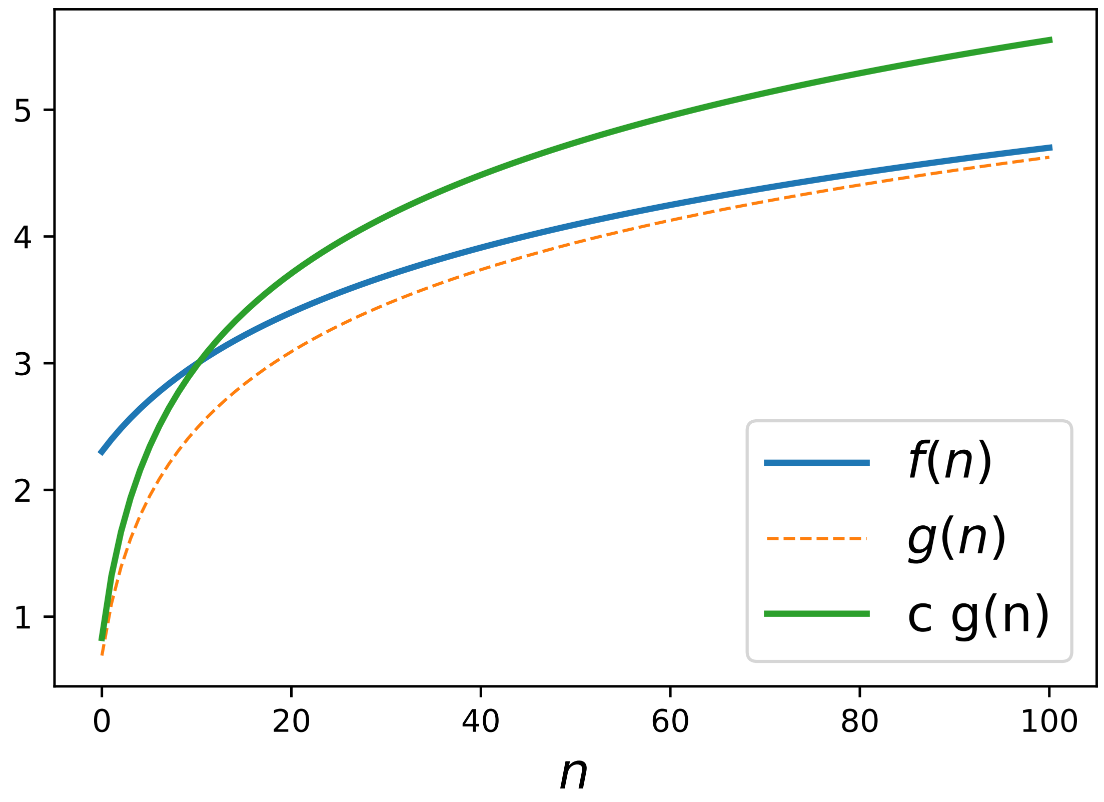

Figure 3.1: The function f is of the same order as the function g if it can be bounded by some constant multiple of g for all positive real numbers.

3.5 Computational complexity

In general, comparing algorithms by computing the number of elementary operations explicitly can be painfully hard to do. Also, it is often not useful. What we care about is how the cost (the number of elementary operations) grows as the instance size n grows. We therefore only care about the *largest* term in the *worst case* as $n \to \infty$.

3.5.1 Big $\mathcal{O}$ notation

Given two functions $f, g\colon \mathbb{R} \to \mathbb{R}$, we say that $f = \mathcal{O}(g)$, or f is of the same order as g, if $\exists n_0, c \in \mathbb{R}^+ \setminus \{0\}$ such that

$$f(n) \leq cg(n) \quad \forall n \geq n_0. \tag{3.12}$$

In other words, for large enough values f can be bounded by scaling g by a constant. This is illustrated in figure 3.1.

The purpose of computational complexity analysis is to bound any function f in terms of a small, simple set of functions g. We want the functions to be simple to make the analysis easier. Equally, we want the specific function used to be as close as possible to f, so the bound does not over-estimate too much.

3.5.2 Computational complexity

Consider two algorithms A and B that solve the same problem P. Let $f_A(n), f_B(n)$ be the number of elementary operations needed for each algorithm as a function of the instance size. Then $\mathcal{O}(f_A)$

is the *computational complexity* of algorithm A (similarly for algorithm B), and we consider A to be *more efficient* than B if

$$\mathcal{O}(f_A) < \mathcal{O}(f_B) \tag{3.13}$$

asymptotically, in the limit as $n \to \infty$.

Examples include

- An $\mathcal{O}(\log n)$ algorithm is more efficient than an $\mathcal{O}(n)$ algorithm.
- An $\mathcal{O}(n^2)$ algorithm is more efficient than an $\mathcal{O}(n^3)$ algorithm.
- An $\mathcal{O}(n^k)$ algorithm is more efficient than an $\mathcal{O}(2^n)$ algorithm, if $k \in \mathbb{N}$.

There are two implicit assumptions we have to make about the functions $f(n)$ to make progress. One is that the number of elementary operations is strictly non-negative so that $f\colon \mathbb{R}^+ \setminus \{0\}$. This seems natural. The other is that f is strictly increasing. This need not be true in general, but is usually a very good approximation.

3.5.3 Properties of the asymptotic model

When functions are the same order

Property 3.5.1. *Let $f, g\colon \mathbb{R}^+ \setminus \{0\} \to \mathbb{R}^+ \setminus \{0\}$. If*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = l \in (0, \infty) \tag{3.14}$$

then $f = \mathcal{O}(g)$ and $g = \mathcal{O}(f)$.

Proof. The definition of the limit implies $\exists \epsilon > 0, N > 0$ such that

$$\left| \frac{f(n)}{g(n)} - l \right| < \epsilon, \quad \forall n \geq N. \tag{3.15}$$

This is equivalent to

$$l - \epsilon < \frac{f(n)}{g(n)} < l + \epsilon, \quad \forall n \geq N. \tag{3.16}$$

We now have two cases.

1. From the upper bound we have $f(n) < (l+\epsilon)g(n)$ for all $n \geq N$. By setting $n_0 = N$ and $c = (l+\epsilon)$, the definition gives that $f = \mathcal{O}(g)$.
2. From the lower bound we have (assuming, without loss of generality, that $\epsilon < l$) $g(n) < (l-\epsilon)^{-1} f(n)$. By setting $n_0 = N$ and $c = (l-\epsilon)^{-1}$, the definition gives that $g = \mathcal{O}(f)$.

□

This shows that (non-zero!) constant factors are essentially irrelevant.

When one function is of lower order

Property 3.5.2. *Let $f, g\colon \mathbb{R}^+ \setminus \{0\} \to \mathbb{R}^+ \setminus \{0\}$. If*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \tag{3.17}$$

then $f = \mathcal{O}(g)$ but $g \neq \mathcal{O}(f)$.

Proof. This is very similar to the previous proof.

The definition of the limit implies $\exists \epsilon > 0, N > 0$ such that

$$\left| \frac{f(n)}{g(n)} - 0 \right| < \epsilon, \quad \forall n \geq N. \tag{3.18}$$

This is equivalent to

$$-\epsilon < \frac{f(n)}{g(n)} < \epsilon, \quad \forall n \geq N. \tag{3.19}$$

We now have two cases.

1. From the upper bound we have $f(n) < \epsilon g(n)$ for all $n \geq N$. By setting $n_0 = N$ and $c = \epsilon$, the definition gives that $f = \mathcal{O}(g)$.

2. By contradiction, assume $g = \mathcal{O}(f)$. By definition, there must exist $n_0, c \in \mathbb{R}^+ \setminus \{0\}$ such that $g(n) \leq cf(n)$ for all $n \geq n_0$. Thus

$$\frac{f(n)}{g(n)} \geq \frac{1}{c} > 0, \quad \forall n \geq n_0. \tag{3.20}$$

This contradicts the limit $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. Therefore $g \neq \mathcal{O}(f)$.

□

In this case we say that f is of *lower order* than g.

Constants are irrelevant

Property 3.5.3. *Assume $a, b \in \mathbb{R}^+ \setminus \{0\}$. Then*

$$af(n) + b = \mathcal{O}(f(n)) \tag{3.21}$$

Proof. This follows from theorem 3.5.1, as

$$\lim_{n\to\infty} \frac{af(n) + b}{f(n)} = a \in (0, \infty). \tag{3.22}$$

□

All logarithms are the same

Property 3.5.4. *Assume $a, b \in \mathbb{R}^+ \setminus \{0\}$. Then*

$$\log_a n = \mathcal{O}\left(\log_b n\right). \tag{3.23}$$

Proof. This follows from the previous case, as

$$\log_a n = (\frac{1}{\log_b a}) \log_b n \tag{3.24}$$

and the term in brackets is a constant. □

Lower order terms are irrelevant

Property 3.5.5. *If g is of lower order than f then*

$$f(n) + g(n) = \mathcal{O}(f). \tag{3.25}$$

Proof.

$$\lim_{n\to\infty} \frac{f(n) + g(n)}{f(n)} = \lim_{n\to\infty} \left(1 + \frac{g(n)}{f(n)}\right) \tag{3.26}$$

$$= 1. \tag{3.27}$$

Hence theorem 3.5.1 gives the result. □

Shifts are irrelevant

Property 3.5.6. *For a monotone increasing function f,*

$$f(n+a) = \mathcal{O}(f). \tag{3.28}$$

Rounding operators are irrelevant

Property 3.5.7. *The ceiling operator $\lceil n \rceil$ returns the smallest integer greater than n. We have*

$$\lceil f(n) \rceil = \mathcal{O}(f). \tag{3.29}$$

Proof. This follows from

$$\lceil f(n) \rceil \leq f(n) + 1 = \mathcal{O}(f). \tag{3.30}$$

□

3.6 Cobham-Edmonds thesis

A suggestion from 1965 by Alan Cobham and Jack Edmonds is that a problem P is *tractable* if

1. there is an algorithm A that solves P;
2. A has computational complexity at most polynomial in the size of instances of P.

Generally we think of tractable problems as solvable on a computer, whilst problems that are not tractable are not. In many specific cases this is untrue, as "average" instances of problems may be solved by the algorithm A much faster than the worst case suggests.

This chapter has shown that the sorting problem is tractable, by constructing two algorithms (selection and insertion sort) that are $\mathcal{O}(n^2)$. Neither are the fastest sorting algorithm available. It is important to note that the existence of a slower algorithm (such as brute-force sort, with complexity $\mathcal{O}(n!)$) does not mean that the *problem* in intractable.

Chapter 4

The Simplex Method

4.1 Recap

In the previous chapters we introduced the linear programming problem. We saw how this could be solved (graphically) for two variables, and also constructed an impractical, brute-force method that would work (very slowly) for the general case.

Problems of current interest can have thousands of constraints and millions of variables. To solve these we want to introduce and analyse the *simplex method*. Introduced by Dantzig in 1947, it builds on the fundamental theorem by giving a method of moving, step by step, from one vertex to an adjacent one until the optimal solution is found.

There are a number of steps we need to complete.

1. How do we characterise a vertex (algebraically)?
2. How do we find a vertex to start from?
3. How do we verify if the current vertex is optimal?
4. How do we move from one vertex to the next?

4.2 Setup and assumptions

In this chapter we will use the standard matrix form,

$$\begin{array}{lrcl} \min & cx & & \\ \text{subject to} & Ax & = & b \\ & x & \geq & 0. \end{array} \tag{4.1}$$

Here $A \in \mathbb{R}^{m\times n}, c, x \in \mathbb{R}^n, b \in \mathbb{R}^m$ are the given coefficients that define the problem. As usual m is the number of constraints and n the number of variables.

We make three assumptions:

1. $b \geq 0$. If this is not true for some row i so that $b_i < 0$ we can multiply that row (both b_i and A_{ij} for all columns j) by -1.
2. $m < n$. That is, the number of constraints is less than the number of variables. If this is not true then the linear system defined by $Ax = b$ is either square or overdetermined, implying one or no solutions (the feasible region is a single point or empty).
3. A has full row rank so its rows are linearly independent.

The second assumption may seem to contradict earlier examples where there were more constraints than variables. However, in all those cases the constraints were *inequalities*. To transform to standard form we need to increase the number of variables (using slack or surplus variables). This will change all those problems to ones with $m < n$.

The use of the third assumption can be checked by assuming A is not of full row rank. In that case we can use row operations to transform the system to $A'x = b'$, where the matrix A' contains two rows i and k such that

$$\begin{aligned} a'_i x &= b'_i, && (4.2) \\ a'_k x &= b'_k, && (4.3) \\ a'_i &= a'_k. && (4.4) \end{aligned}$$

There are then two cases.

1. $b'_i = b'_k$: the two constraints are identical and one can be dropped from the problem.
2. $b'_i \neq b'_k$: the constraints are inconsistent and so the linear problem is infeasible.

So the third assumption stops us from considering infeasible problems and ensures constraints are not repeating information.

4.3 Basic matrices

There is a general mathematical strategy of solving a complex problem by first solving a simple case, then transforming the general case to the simple one. This can seem indirect. The next few sections will build up a simple case for linear programs.

Let $B \subseteq \{1, \ldots, n\}$ be a subset of the column indexes of A, with $|B| = m$. That is, B will identify a number of columns of A, corresponding to variables in the linear program, and the number of columns identified will match the number of constraints.

A *basic matrix* A_B is the submatrix of A with *linearly independent columns* that is obtained by dropping every column with index not in B. If A_B is a basic matrix then

- A_B is square;
- A_B has full row and column rank (by the assumptions);
- A_B is invertible, as it has full row and column rank.

We can use this construction to split, of *partition*, the linear program. Let $N = \{1, \ldots, n\} \setminus B$ be the complement of B. Let c_B, x_B, A_B be the components (for the vectors c, x) or columns (for the matrix A) that are indexed by B, and c_N, x_N, A_N be the components (for the vectors c, x) or columns (for the matrix A) that are indexed by N. Then we can use column operations on the linear program to re-order the vectors and matrices, writing

$$\begin{aligned} c &= (c_B, c_N), && (4.5) \\ A &= (A_B, A_N), && (4.6) \\ x &= (x_B, x_N). && (4.7) \end{aligned}$$

This means that our standard matrix form description of the linear program becomes

$$\begin{array}{lrcl} \text{min} & \begin{pmatrix} c_B & c_N \end{pmatrix} \begin{pmatrix} x_B \\ x_N \end{pmatrix} & & \\ \text{subject to} & \begin{pmatrix} A_B & A_N \end{pmatrix} \begin{pmatrix} x_B \\ x_N \end{pmatrix} & = & b \\ & \begin{pmatrix} x_B \\ x_N \end{pmatrix} & \geq & \begin{pmatrix} 0 \\ 0 \end{pmatrix}. \end{array} \tag{4.8}$$

Rewriting this by expanding out the matrix multiplications gives

$$\begin{array}{lrcl} \min & c_B x_B + c_N x_N & & \\ \text{subject to} & A_B x_B + A_N x_N & = & b \\ & x_B & \geq & 0 \\ & x_N & \geq & 0. \end{array} \tag{4.9}$$

Why have we made the problem more complicated? Because the basic matrix is invertible, we can *directly solve* for the optimal basic solution. We have

$$Ax = b \tag{4.10}$$

$$\iff \quad A_B x_B + A_N x_N = b \tag{4.11}$$

$$\iff \quad x_B = A_B^{-1} b - A_B^{-1} A_N x_N. \tag{4.12}$$

This allows us to write the objective function as

$$cx = c_B x_B + c_N x_N \tag{4.13}$$

$$= c_B A_B^{-1} b + \left(c_N - c_B A_B^{-1} A_N\right) x_N. \tag{4.14}$$

We can also add the term $(c_B - c_B A_B^{-1} A_B) x_B$ to the objective function as it vanishes. This is useful later.

This defines the *basic form* of a standard form linear program: if A_B is a basic matrix then the basic form is

$$\begin{array}{lrcl} \min & c_B A_B^{-1} b + \left(c_N - c_B A_B^{-1} A_N\right) x_N & + & \left(c_B - c_B A_B^{-1} A_B\right) x_B \\ \text{subject to} & x_B + A_B^{-1} A_N x_N & = & A_B^{-1} b \\ & x_B & \geq & 0 \\ & x_N & \geq & 0. \end{array} \tag{4.15}$$

4.4 Basic solutions

When the linear program is in basic form we call x_B the *basic variables* and x_N the *non-basic variables.* If there is a solution with $x_N = 0$ we call it a *basic solution.* A basic solution need not satisfy $x_B \geq 0$ and so it may not fall in the feasible region. If there is a solution $x = (x_B, x_N)$ with $x_N = 0$ and also $x_B \geq 0$ then it is called *basic feasible.*

Note that the constraints automatically imply that if the solution is basic and so $x_N = 0$, then

$$x_B = A_B^{-1} b. \tag{4.16}$$

Basic solutions can be constructed by solving the linear system defined by the basic matrix.

4.4.1 Basic solutions and vertices

There is a direct link between basic solutions and vertices.

Theorem 4.4.1. *x is a basic feasible solution if, and only if, it is a vertex of the feasible region.*

Proof. First prove the implication using contradiction.

Let x be a basic feasible solution with $x = (x_B, x_N)$. Assume, by contradiction, that x is not a vertex. As the feasible region is convex, if x is not a vertex then there are two distinct points y, z, both distinct from x, such that

$$x = \lambda y + (1 - \lambda) z, \quad \lambda \in (0, 1). \tag{4.17}$$

We can split both points in the same way as the basic feasible solution x, as $y = (y_B, y_N)$ and $z = (z_B, z_N)$. This gives

$$x_N = \lambda y_N + (1 - \lambda) z_N, \quad \lambda \in (0, 1). \tag{4.18}$$

As x is a basic feasible solution we have $x_N = 0$. As $\lambda \in (0,1)$ we have $\lambda > 0$ and $1 - \lambda > 0$. As y and z are feasible we have $y_N, z_N \geq 0$. This requires that $y_N = 0 = z_N$.

Similarly, as y and z are feasible, we also have that $Ay = b$ and $Az = b$. As all the non-basic pieces must vanish, we have that $A_B x_B = b = A_B y_B = A_B z_B$. Since A_B is non-singular (as it is a basic matrix), any linear system defined by it has a unique solution. Therefore $x_B = y_B = z_B$. As all the non-basic pieces vanish, we have that $x = y = z$. This contradicts the assumption that they are all distinct. Therefore a basic feasible solution is a vertex.

Next prove that a vertex is a basic feasible solution. Define $\hat{B} \subseteq \{1, \dots, n\}$ as the set of indexes of the components of the vertex x which are positive, $\hat{B} = \{i \colon x_i > 0\}$. Let $\hat{N}$ be the complement, $\hat{N} = \{1, \dots, n\} \setminus \hat{B}$. Write $x = (x_{\hat{B}}, x_{\hat{N}})$ and re-write $Ax = b$ as

$$A_{\hat{B}} x_{\hat{B}} + A_{\hat{N}} x_{\hat{N}} = b. \tag{4.19}$$

By construction $x_{\hat{N}} = 0$ and so $A_{\hat{B}} x_{\hat{B}} = b$.

We need to show that $A_{\hat{B}}$ is a basic matrix. We do this by contradiction. Assume the columns of $A_{\hat{B}}$ are not linearly independent. Then there is some $y_{\hat{B}} \neq 0$ such that $A_{\hat{B}} y_{\hat{B}} = 0$. By construction $x_{\hat{B}} > 0$. Therefore there is some constant $\epsilon > 0$ such that

$$x_{\hat{B}} - \epsilon y_{\hat{B}} \geq 0 \text{ and } x_{\hat{B}} + \epsilon y_{\hat{B}} \geq 0. \tag{4.20}$$

Also by construction we have

$$A_{\hat{B}} \left(x_{\hat{B}} \pm \epsilon y_{\hat{B}} \right) = b. \tag{4.21}$$

Thus the two distinct points $(x_{\hat{B}} \pm \epsilon y_{\hat{B}}, 0)$ are feasible solutions to $Ax = b$. Finally, note that

$$x = (x_{\hat{B}}, 0) = \frac{1}{2} \left(x_{\hat{B}} - \epsilon y_{\hat{B}} \right) + \frac{1}{2} \left(x_{\hat{B}} + \epsilon y_{\hat{B}} \right) \tag{4.22}$$

This shows that x can be written as the convex combination of two distinct feasible points. This contradicts x being a vertex. Therefore the columns of $A_{\hat{B}}$ are linearly independent.

To show that $A_{\hat{B}}$ is basic we need to show that it is a square matrix. If $|\hat{B}| = m$ then the proof is complete. If not, we can select more indices from $\hat{N}$ so that the columns of the enlarged $A_{\hat{B}}$ are linearly independent. This is possible as A has full row rank. The solution will then be degenerate - it will have basic variables with zero value - but remain basic. $\square$

4.5 Optimality

The previous sections have shown how to algebraically characterise a vertex (and we remember that the optimal solution must lie on some vertex): a vertex corresponds to a basic feasible solution. We now want to understand how to check if a vertex (or equivalently a basic feasible solution) is optimal, without enumerating all cases.

4.5.1 Reduced costs

Remember that the linear program can be written in basic form, with some basic matrix A_B, as

$$\begin{array}{llcl} \text{min} & c_B A_B^{-1} b + \left(c_N - c_B A_B^{-1} A_N \right) x_N & + & \left(c_B - c_B A_B^{-1} A_B \right) x_B \\ \text{subject to} & x_B + A_B^{-1} A_N x_N & = & A_B^{-1} b \\ & x_B & \geq & 0 \\ & x_N & \geq & 0. \end{array} \tag{4.15}$$

Certain terms are sufficiently important to get names, which are all types of *reduced costs*:

$$\bar{c}_N = c_N - c_B A_B^{-1} A_N \qquad \text{reduced costs of non-basic variables,} \tag{4.23}$$

$$\bar{c}_B = c_B - c_B A_B^{-1} A_B \qquad \text{reduced costs of basic variables,} \tag{4.24}$$

$$\bar{c} = c - c_B A_B^{-1} A \qquad \text{reduced costs (of the variables).} \tag{4.25}$$

Note that by construction $\bar{c}_B = 0$ automatically. We also note that $\bar{c} = (\bar{c}_B, \bar{c}_N)$ in a similar fashion to the split of a solution into basic and non-basic pieces (this can be checked explicitly using the matrix form split).

In terms of the reduced costs the basic form linear program is

$$\begin{array}{llcl} \min & c_B A_B^{-1} b + \bar{c}_N x_N + \bar{c}_B x_B & & \\ \text{subject to} & x_B + A_B^{-1} A_N x_N & = & A_B^{-1} b \\ & x_B & \geq & 0 \\ & x_N & \geq & 0. \end{array} \tag{4.26}$$

This emphasises, as $\bar{c}_B = 0$, that x_B has no impact on the objective function. It also gives us an interpretation of the reduced costs of the non-basic variables: the component $(\bar{c}_N)_j$ is equal to the change in the objective function value when changing the non-basic variable $(x_N)_j$ by one unit.

Therefore, if we start from a basic feasible solution (where, by construction, $x_N = 0$), we can improve (make smaller) the objective function by increasing any component $(x_N)_j$ where the corresponding component of the reduced cost is negative, $(\bar{c}_N)_j < 0$.

4.5.2 Optimality test

Theorem 4.5.1. *If, given a basic feasible solution x with basis B, the reduced cost vector $\bar{c}$ is non-negative, then x is optimal.*

Proof. We show that, for any other feasible solution y, $cy \geq cx$. To do this define $d = y - x$. As

$$cy \geq cx \tag{4.27}$$
$$\iff \qquad c(y - x) \geq 0 \tag{4.28}$$
$$\iff \qquad cd \geq 0, \tag{4.29}$$

we are looking to prove $cd \geq 0$.

Since x and y are both feasable solutions we have $Ax = b$ and $Ay = b$, implying

$$Ad = A(x - y) \tag{4.30}$$
$$= Ax - Ay \tag{4.31}$$
$$= 0. \tag{4.32}$$

Writing $d = (d_B, d_N)$ as usual we have

$$Ad = 0 \tag{4.33}$$
$$\iff \qquad A_B d_B + A_N d_N = 0. \tag{4.34}$$

The basic matrix A_B is non-singular by construction, so

$$d_B = -A_B^{-1} A_N d_N. \tag{4.35}$$

We can then evaluate

$$cd = c_B d_B + c_N d_N \tag{4.36}$$
$$= c_B \left(-A_B^{-1} A_N d_N\right) + c_N d_N \tag{4.37}$$
$$= \left(c_N - c_B A_B^{-1} A_N\right) d_N \tag{4.38}$$
$$= \bar{c}_N d_N. \tag{4.39}$$

By assumption we have $\bar{c}_N \geq 0$. To show that $cd \geq 0$ we need to show that $d_N = y_N - x_N \geq 0$.

We note that y is a feasible solution which implies that $y_N \geq 0$. Similarly x is a *basic* feasible solution which requires that $x_N = 0$. Hence $d_N \geq 0$, implying that $cd \geq 0$ and hence $cy \geq cx$. Thus any feasible solution that is not x increases the value of the objective function, meaning x is optimal. □

4.6 Moving between vertices

We now have both a characterisation of vertices (as basic feasible solutions) and a test of their optimality (the associated reduced cost vector is non-negative). If we can easily move from one vertex to another then we have (most of) the ingredients of an algorithm.

We know intuitively that a vertex is connected to a neighbouring vertex by an edge; a one dimensional boundary line of the feasible region. We know a vertex corresponds to a basic feasible solution. The basis B is the set of indices defining which variables are basic at a particular vertex, and has complement N. If we move along an edge we might expect to only be changing two variables (in an appropriate coordinate system the one dimensional line defining the edge lies in a two dimensional plane spanned by two basis vectors). Therefore, moving along an edge from one vertex to another corresponds to *swapping one non-basic variable with one basic variable.* This is equivalent to swapping one entry of N with one entry of B.

4.6.1 Minimum ratio test

We go back to our linear program

$$\begin{array}{lrcl} \text{min} & c_B A_B^{-1} b + \bar{c}_N x_N + \bar{c}_B x_B & & \\ \text{subject to} & x_B + A_B^{-1} A_N x_N & = & A_B^{-1} b \\ & x_B & \geq & 0 \\ & x_N & \geq & 0. \end{array} \tag{4.26}$$

For compactness define $\bar{A} = A_B^{-1} A_N$ and $\bar{b} = A_B^{-1} b$, so the system of equations becomes

$$x_B + \bar{A} x_N = \bar{b}. \tag{4.40}$$

For each single row $i \in \{1, \ldots, m\}$ we can write this explicitly as

$$(x_B)_i + \sum_{j \in N} \bar{a}_{ij} (x_N)_j = \bar{b}_i. \tag{4.41}$$

Now, if x corresponds to a vertex then it is a basic feasible solution so that $x_N = 0$. We assume at this point that some component has negative reduced cost. That is, $\exists s \in N$ such that $\bar{c}_s < 0$. It must be the case that $s \in N$ (if one exists) as $\bar{c}_B = 0$ automatically. Therefore we can improve the value of the objective function by increasing the value of x_s (and hence removing s from N). The constraints then require that

$$(x_B)_i = \bar{b}_i - \bar{a}_{is} x_s. \tag{4.42}$$

There is no sum as all terms in N are still basic, so $(x_N)_j = 0$ for all $j \in N \setminus \{s\}$.

We also need to retain all the positivity constraints $(x_B)_i \geq 0$. This requires that

$$\bar{b}_i - \bar{a}_{is} x_s \geq 0. \tag{4.43}$$

If $\bar{a}_{is} \leq 0$ this cannot cause a problem: increasing x_s can never violate this constraint. However, if $\bar{a}_{is} > 0$ then this implies that we must impose

$$x_s \leq \frac{\bar{b}_i}{\bar{a}_{is}}. \tag{4.44}$$

As we need this condition to hold for all constraints, this means we must impose

$$x_s \leq \min_{i \in B \colon \bar{a}_{is} > 0} \frac{\bar{b}_i}{\bar{a}_{is}}. \tag{4.45}$$

We use this to define the intermediate variable

$$\theta^* = \min_{i \in B \colon \bar{a}_{is} > 0} \frac{\bar{b}_i}{\bar{a}_{is}}. \tag{4.46}$$

If $\theta^* > 0$ then setting $x_s = \theta^*$ ensures that

- x_s becomes basic (so that s moves from N to B);
- The value of every variable $(x_B)_r$ with

$$r \in \underset{i \in B\colon \bar{a}_{is} > 0}{\operatorname{argmin}} \frac{\bar{b}_i}{\bar{a}_{is}} \tag{4.47}$$

  is decreased to zero. We can select any one of these variables to leave the basis (so that r moves from B to N), whilst the others remain (with value zero).

This gives us a recipe for moving from vertex to vertex: find the value of θ^* for given index s and an index r that gives the minimum. Swap s and r. This is the *minimum ratio test.*

We restricted to the case where $\bar{a}_{is} > 0$ when computing the minimum ratio as the other case does not violate the positivity constraint. However, we also need to consider the objective function.

Theorem 4.6.1. *If there is an index s such that $\bar{c}{+}s < 0$ with $\bar{a}_{is} \leq 0$ for every row $i \in \{1, \dots, m\}$, the linear program is unbounded.*

Proof. Since $\bar{a}_{is} \leq 0$ for all i, x_s can be increased indefinitely without violating any constraint. As $\bar{c}_s < 0$ the objective function can be improved indefinitely. □

4.6.2 Degeneracy

We have seen with the minimum ratio test that it possible to have a basic feasible solution with at least one basic variable having zero value. These solutions are called *degenerate.*

Degenerate solutions can cause problems as different degenerate basic feasible solutions can correspond to the same vertex. To show this, consider a basic feasible solution which is degenerate for some row, index $\hat{\imath}$, so that

$$(x_B)_{\hat{\imath}} = \bar{b}_{\hat{\imath}} = 0. \tag{4.48}$$

Assume we want to increase the value of the non-basic variable x_s, $s \in N$, which starts from zero. As usual we need

$$\bar{b}_{\hat{\imath}} - \bar{a}_{\hat{\imath}s} x_s \geq 0 \tag{4.49}$$

to ensure that $(x_B)_{\hat{\imath}} \geq 0$. If the problem is not unbounded then the minimum ratio test gives

$$\theta^* = \min_{i \in B\colon \bar{a}_{is} > 0} \frac{\bar{b}_i}{\bar{a}_{is}} = 0, \tag{4.50}$$

as $\hat{\imath} \in B$ and $\bar{b}_{\hat{\imath}} = 0$.

Thus the non-basic variable x_s enters the basis but remains at value zero, whilst the basic variable $(x_B)_{\hat{\imath}}$ leaves the basis and also remains at value zero. So, in spite of changing the basis B the values of x are unchanged. Therefore we have not actually moved to a different vertex.

4.6.3 Cycling and Bland's rule

If there is a degenerate solution the simplex method (using the minimum ratio test to move from one vertex to another) my enter an endless loop in which the the same basic and non-basic variables are swapped in and out of the basis forever. B and N would be changing but x would remain the same.

The problem arises as there are multiple choices of indexes to move out of the basis. We need a rule to choose which indexes to use, so that this rule ensures a cycle cannot occur.

Bland's rule is the following method:

- Among all $s \in \{1, \dots, n\}$ with $\bar{c}_s < 0$, choose the smallest s.
- Among all $r \in \operatorname{argmin}_{i \in B\colon \bar{a}_{is} > 0} \frac{\bar{b}_i}{\bar{a}_{is}}$, choose the smallest r.

Theorem 4.6.2. *With Bland's rule, the simplex method converges in a finite number of iterations.*

We will not show the proof here - there is some discussion in Chvátal.

4.7 Summary

The simplex method with Bland's rule starts from the linear program

$$\begin{array}{lrcl} \min & c_B A_B^{-1} b + \bar{c}_N x_N + \bar{c}_B x_B & & \\ \text{subject to} & x_B + \bar{A} x_N & = & \bar{b} \\ & x_B & \geq & 0 \\ & x_N & \geq & 0. \end{array} \tag{4.26}$$

Start the algorithm with a basis $B \subseteq \{1, \ldots, n\}$ with a basic matrix A_B yielding a basic feasible solution $x_B = A_B^{-1} b \geq 0$ with $x_N = 0$.

At each step of the simplex method, then

1. Compute
   - A_B^{-1},
   - $\bar{A} = A_B^{-1} A_N$,
   - $\bar{b} = A_B^{-1} b$,
   - $\bar{c} = c - c_B A_B^{-1} A$.
2. If $|\{s \in \{1, \ldots, n\} : \bar{c}_s < 0\}| > 0$, then
   - Pick the smallest index $s \in \{1, \ldots, n\}$ with $\bar{c}_s < 0$;
   - Pick the smallest index r where

     $$r \in \underset{i \in B : \, \bar{a}_{is} > 0}{\operatorname{argmin}} \frac{\bar{b}_i}{\bar{a}_{is}}. \tag{4.51}$$

     If there is no such index then the problem is unbounded;
   - Let $B \to B \setminus \{r\} \cup \{s\}$.
3. Else we have reached the optimal solution. Stop, returning $x = (x_B, x_N)$ with $x_B = A_B^{-1}$ and $x_N = 0$.

4.8 Tableau form

The simplex method as summarised above is a working algorithm. However, it is not in the most efficient form to implement on a computer. In part this is because there are many different variables to compute and manipulate. In part it is because some of the calculations (such as the explicit matrix inversion of A_B) are expensive.

We can build on a number of concepts from linear algebra to make the algorithm more efficiently implemented.

4.8.1 The problem as a block matrix

We again start from the standard basic form linear program

$$\begin{array}{lrcl} \min & c_B A_B^{-1} b + \bar{c}_N x_N + \bar{c}_B x_B & & \\ \text{subject to} & x_B + \bar{A} x_N & = & \bar{b} \\ & x_B & \geq & 0 \\ & x_N & \geq & 0. \end{array} \tag{4.26}$$

We transfer the problem to the *tableau*

$$\begin{array}{|c|c|c|} \hline -c_B A_B^{-1} b & \bar{c}_B & \bar{c}_N \\ \hline \bar{b} & I & \bar{A} \\ \hline \end{array} \tag{4.52}$$

The objective function is contained in the first row, row 0. The value in position $(0, 0)$ is the (negative of the) objective function value of the current basic solution. Rows 1 to m contain the constraints. The right hand side $\bar{b}$ is in column 0. The m rows and columns corresponding to the basic variables are those of the identity matrix I, as the constraints have been written as $x_B + \bar{A}x_N = \bar{b}$ in this form of the linear program. Each row corresponds to a specific basic variable.

To complete the simplex method we need to apply Bland's rule and the minimum ratio test and then swap our chosen basic and non-basic variables.

1. Look at row zero, all columns except column zero. This contains the reduced costs. If any entry is negative we are not optimal and must continue.
2. Using Bland's rule, set s as the index of the first column containing a negative reduced cost.
3. Using Bland's rule and the minimum rate test, set r as the row index with the minimum ratio, only considering rows i where $\bar{a}_{is} > 0$, and taking the ratio with $\bar{b}_i$, which is in column zero of the tableau.
4. Perform a *pivot* operation so that the tableau entry $\bar{a}_{rs} = 1$ and every other entry in column s, corresponding to $\bar{a}_{is}$, is set to zero. This swaps the basis and non-basis entries.

4.8.2 Pivoting

Pivoting is the operation used in Gaussian Elimination to solve linear systems. It is highly efficient allowing us to move from one basic solution to another without explicitly constructing a lot of intermediate terms. Pivoting consists of two steps. Given a matrix A and a pair of row-column indexes (r, s) with $a_{rs} \neq 0$, the steps are

1. Divide the r^{th} row by a_{rs};
2. For each row with index $i \neq r$, subtract the r^{th} row multiplied by a_{is}

The first step ensures that $a_{rs} \to 1$. The second step ensures that $a_{is} \to 0$ for all $i \neq r$.

After the pivoting operation the column with index s becomes equal to a column of the identity matrix. That means that the variable s is now basic. Because the basic variables have columns that match those in the identity matrix, the pivoting operation will only change the contents of column r. Therefore the pivoting operation swaps $(x_B)_r$ and x_s as expected.

4.9 Two-phase simplex method

We now, via the simplex method in tableau form, have an efficient algorithm for solving a linear program. However, it is not a general algorithm: it requires that the linear program is in basic form,

$$\begin{array}{llcl} \min & c_B A_B^{-1} b + \bar{c}_N x_N + \bar{c}_B x_B & & \\ \text{subject to} & x_B + \bar{A}x_N & = & \bar{b} \\ & x_B & \geq & 0 \\ & x_N & \geq & 0, \end{array} \tag{4.26}$$

before it can start. We need to work out how to get a linear program in, for example, the standard form

$$\begin{array}{lrcl} \min & cx & & \\ \text{subject to} & Ax & = & b \\ & x & \geq & 0, \end{array} \tag{4.53}$$

into the basic form before we can use the simplex method.

The two-phase simplex method is the approach to use. This

1. iteratively constructs *some* basic feasible solution and from it the associated basic feasible tableau;

2. applies the tableau simplex method to find the optimal solution.

An important part of the first phase is that, by constructing *some* basic feasible solution it shows that the problem is feasible. If the first phase fails it means the problem is infeasible.

4.9.1 Auxiliary problem

Define the vector $e = (1, \dots, 1)$ with m components. Starting from the linear program in standard form,

$$\begin{array}{lrcl} \min & cx & & \\ \text{subject to} & Ax & = & b \\ & x & \geq & 0, \end{array} \tag{4.54}$$

define the *auxiliary problem*

$$\begin{array}{lrcl} \min & ey & & \\ \text{subject to} & Ax + Iy & = & b \\ & x & \geq & 0, \\ & y & \geq & 0. \end{array} \tag{4.55}$$

Note that $ey = \|y\|_1$ as $y \geq 0$. This problem is trying to find the "smallest" y (in the one norm) such that $Ax + Iy = b$.

Theorem 4.9.1. *The auxiliary problem admits an optimal solution (x^*, y^*) with $ey^* = 0$ if, and only if, the original problem is feasible.*

Proof. For the implication, work by contradiction. Assume there is an optimal solution (x^*, y^*) with $ey^* > 0$. Therefore there is no solution (x', y') with $x' \geq 0$ and $y' = 0$ satisfying $Ax' + Iy' = Ax' = b$. This is because any such solution would have objective function value $ey' = 0 < ey^*$, which is a contradiction.

Finally, if the original problem admits a feasible solution x' then the pair (x', y') with $y' = 0$ is feasible for the auxiliary problem. It is also optimal as the objective function $ey' = 0$, which is the minimum as $e, y' \geq 0$. □

Note that it immediately follows from the proof that if the auxiliary problem does not admit an optimal solution the linear program is infeasible.

4.9.2 Phase 1 tableau

The auxiliary problem as constructed automatically contains an identity matrix corresponding to the columns associated with y. It immediately follows that y is basic. Therefore we can immediately write down the tableau

$$\begin{array}{|c|c|c|} \hline 0 & 0 & 1, \dots, 1 \\ \hline b & A & I \\ \hline \end{array} \tag{4.56}$$

This is, however, not completely in the tableau form that we require for the simplex method. The issue is the reduced cost vector of the basic variables. To start the simplex method we need the reduced costs to be 0: here they are 1. This has been caused by expressing the objective function in terms of the basic variables. We instead want to express the objective function solely in terms of the non-basic variables. To do this, use

$$Ax + Iy = b \tag{4.57}$$

$$\implies \quad y = b - Ax \tag{4.58}$$

$$\implies \quad ey = eb - eAx. \tag{4.59}$$

By re-writing the objective function in this form we have the tableau

$$\begin{array}{|c|c|c|} \hline -eb & -eA & 0, \dots, 0 \\ \hline b & A & I \\ \hline \end{array} \tag{4.60}$$

We can now apply the simplex method to the auxiliary problem.

4.9.3 Outline of two-phase method

Assume we are starting from the standard form of the linear program,

$$\begin{array}{lrcl} \min & cx & & \\ \text{subject to} & Ax & = & b \\ & x & \geq & 0, \end{array} \tag{4.61}$$

construct the tableau for the auxiliary problem as

$$\begin{array}{|c|c|c|} \hline -eb & -eA & 0,\ldots,0 \\ \hline b & A & I \\ \hline \end{array} \tag{4.62}$$

Then

1. Solve the auxiliary problem. Let (x^*, y^*) be an optimal solution.
   (a) If the optimal objective function value $ey^* > 0$ the original problem is infeasible. Stop at this point.
   (b) If $ey^* = 0$ then
      i. If there is some basic variable y_i with zero value the auxiliary problem is degenerate. Each such basic y_i can be made non-basic by pivoting on any non-zero component of its row.
      ii. Delete all columns corresponding to the basic y_i variables.
      iii. Re-write the objective function in terms of the non-basic x_j variables.
2. Rewrite the objective function of the original problem in terms of only the non-basic variables.
3. Write the basic feasible tableau for the original problem using the basic solution found in phase 1. Apply the simplex method to it.

The issue of degeneracy in the auxiliary problem needs more explanation. If there is a degeneracy for y_i then the tableau is in the form

$$\begin{array}{c|c|c|c|} & & x_1 \ldots x_j \ldots x_n & y_1 \ldots y_i \ldots y_m \\ \hline & 0 & \ldots & \ldots \\ \hline & & & 0 \\ & & & \vdots \\ y_i & 0 & \bar{a}_{i1} \ldots \bar{a}_{ij} \ldots \bar{a}_{in} & 1 \\ & & & \vdots \\ & & & 0 \\ \hline \end{array} \tag{4.63}$$

Here we have added information about which rows and columns correspond to which variables at the top and left. This is the tableau once the simplex method has been run on the auxiliary problem, and the objective function is zero meaning a feasible solution exists.

We can pivot on any $\bar{a}_{ij}$ that is non-zero. The reason is that the value changes by $\bar{b}_i/\bar{a}_{ij}$, and $\bar{b}_i = 0$. This pivot operation ensures that x_j enters the basis with value zero, and y_i leaves the basis (also with value zero).

If, however, there is no entry $\bar{a}_{ij}$ that is non-zero, then the matrix A is not full rank. In this case the entire row (and hence variable y_i) can be removed from the tableau immediately.

4.10 Efficiency and complexity

Our aim in deriving the simplex method was to avoid enumerating all the vertexes of the feasible region. The reason is that enumerating all vertexes has computational complexity $\mathcal{O}(n!)$ which is not tractable.

The simplex method works by finding one vertex and then moving to neighbouring vertexes one at a time. If the simplex method has to visit every vertex to find the optimal solution then it also has complexity $\mathcal{O}(n!)$. Unfortunately there are "pathological" cases where this can occur.

Theorem 4.10.1 (Klee and Minty, 1972)**.** *The linear program*

$$\begin{array}{llcll} \max & \sum_{j=1}^{n} 10^{n-j} x_j & & & \\ \textit{subject to} & \left(2\sum_{j=1}^{i-1} 10^{i-j} x_j\right) + x_i & \leq & 100^{i-1}, & i \in \{1,\dots,n\} \\ & x_j & \geq & 0, & j \in \{1,\dots,n\}, \end{array} \tag{4.64}$$

requires $2^n - 1$ *iterations to solve with the simplex method.*

Therefore the worst case analysis of the simplex method shows that it is intractable. *However*, when applied to "generic" problems in practical use the simplex method is (nearly always) fast. Empirically the computational complexity of the average case is $\mathcal{O}(m\log(n))$.

Chapter 5

Integer linear programs

So far we have considered linear programs of the form

$$\begin{array}{lrcl} \min & cx & & \\ \text{subject to} & Ax & \geq & b \\ & x & \geq & 0 \\ & x & \in & \mathbb{R}^n. \end{array} \tag{5.1}$$

We have here made explicit that each component of x is a real number. In addition we have $A \in \mathbb{R}^{m\times n}, b \in \mathbb{R}^m, c, x \in \mathbb{R}^n$ being matrices and vectors in the appropriate real spaces.

There are a lot of applications where this does not make perfect sense. There are many situations where talking about half a person, or a third of a car, or similar, is meaningless. In these cases we instead need to consider the *integer linear program*

$$\begin{array}{lrcl} \min & cx & & \\ \text{subject to} & Ax & \geq & b \\ & x & \geq & 0 \\ & x & \in & \mathbb{Z}^n, \end{array} \tag{5.2}$$

where $A \in \mathbb{Q}^{m\times n}, b \in \mathbb{Q}^m, c, x \in \mathbb{Q}^n$ are now matrices and vectors in the appropriate *rational* spaces.

Another special case is where we have to make yes/no or true/false decisions. For this weconsider the *integer linear program*

$$\begin{array}{lrcl} \min & cx & & \\ \text{subject to} & Ax & \geq & b \\ & x & \geq & 0 \\ & x & \in & \{0,1\}^n, \end{array} \tag{5.3}$$

where again $A \in \mathbb{Q}^{m\times n}, b \in \mathbb{Q}^m, c \in \mathbb{Q}^n$ are now matrices and vectors in the appropriate *rational* spaces.

5.1 Optimal solutions

The feasible region of an integer linear program is not a polyhedron. It is instead the intersection of a polyhedron with a lattice, where the lattice is a grid of integer points. This immediately implies that, in general, an integer linear program cannot be solved by the simplex method (as the "vertices" need not be optimal). This is illustrated in figure 5.1.

We can, however, solve the linear program *relaxation* of an integer linear program. This is obtained by dropping the restriction $x \in \mathbb{Z}^n$, replacing it with $x \in \mathbb{R}^n$. Essentially, we pretend

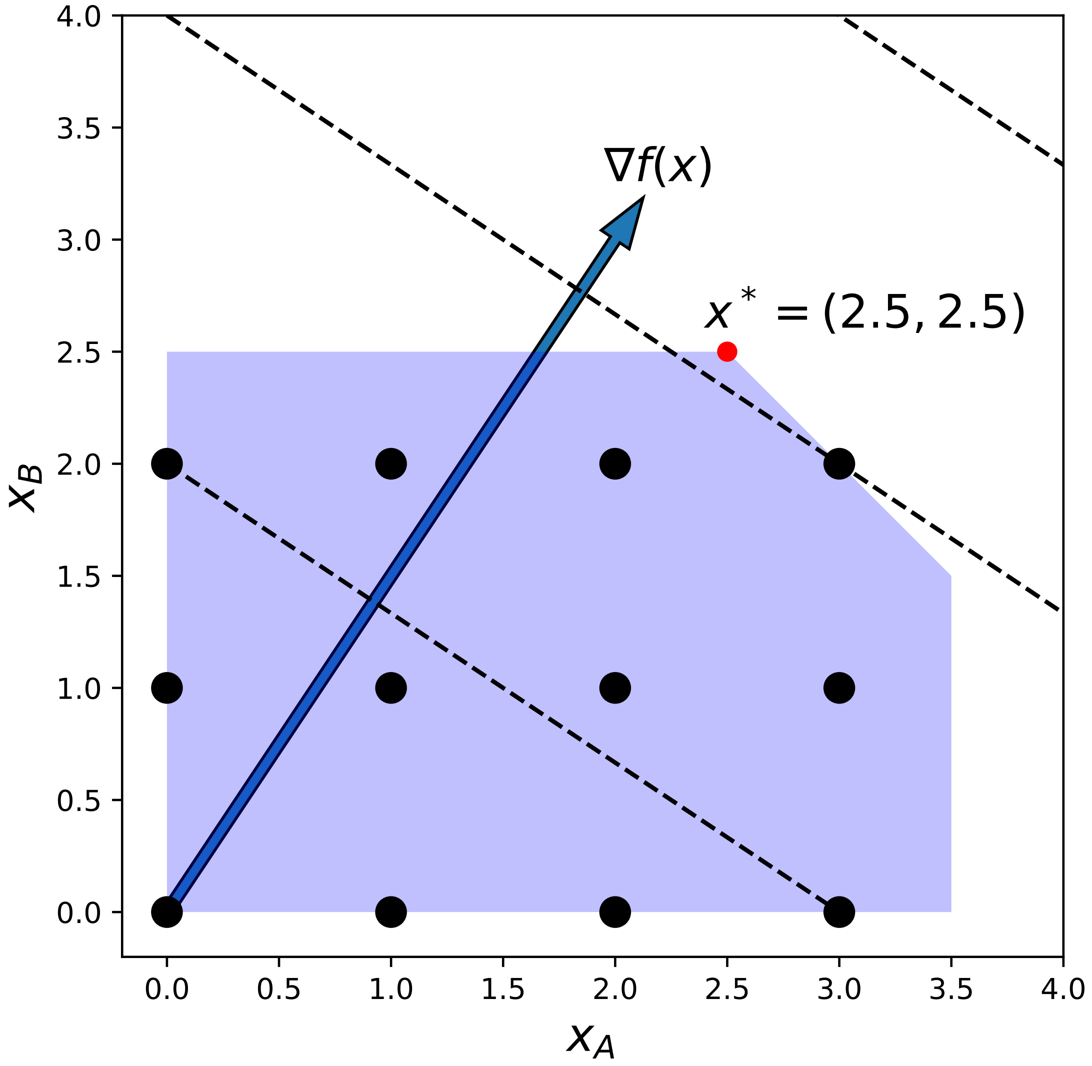

Figure 5.1: The feasible region for an integer linear program is given by the intersection of the feasible region for the standard linear program (the blue shaded region) and the integer lattice. The black dots here form that restricted feasible region. The optimal solution for the standard problem (shown by the red dot) does not lie in this feasible region, although it is close to the optimal integer solution (as seen by the level curves of the objective function, given by the dashed lines).

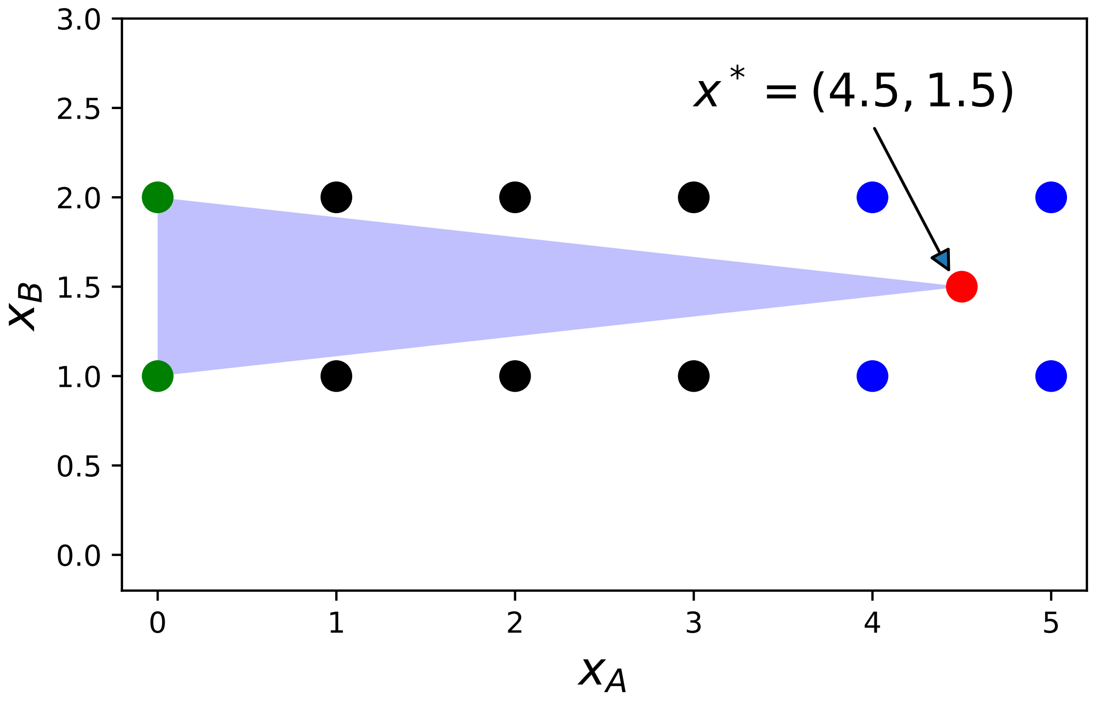

Figure 5.2: A pathological integer linear program. The feasible region of the standard problem (blue shaded region) is “long and thin”. The integer solutions (blue dots) that neighbour the solution to the standard problem (red dot) are not close to any feasible solution to the integer problem (green dots).

that an integer linear program is a standard linear program. This can be solved by the simplex method as before.

Unfortunately, solutions of the relaxation problem can be a long way from the optimal solution of the integer linear program. If the feasible region near the optimal solution x^* to the relaxation problem is “long and thin”, then points on the integer lattice surrounding x^* can easily be infeasible. It is possible to construct cases where the difference in the objective function value between the integer linear program and it relaxation problem are arbitrarily large. This is illustrated in figure 5.2.

Typical methods for solving integer problems do use the simplex method to solve linear programs, typically applied to many pieces of the feasible region individually (later courses will cover this). The simplex method remains a key building block.

5.2 Total unimodularity

5.2.1 When relaxation works

There are, however, problems where the relaxation problem works. Consider the problem

$$\begin{array}{lrcl} \min & cx & & \\ \text{subject to} & Ax & = & b \\ & x & \geq & 0 \\ & x & \in & \mathbb{Z}^n, \end{array} \tag{5.4}$$

where $b \in \mathbb{Z}^n$. This can be constructed by scaling each row of $Ax = b$ to make b integer.

Consider the relaxation problem where all restrictions are dropped. For any optimal (basic) solution of the relaxation problem we have

$$x_B = A_B^{-1} b, \quad x_N = 0. \tag{5.5}$$

We have that b is integer. If A_B^{-1} is also integer it follows that x_B is integer. Therefore the solution to the linear program is also a solution to the *integer* linear program, and the simplex method will work.

5.2.2 Total unimodularity

We want some criteria that checks when the inverse of a matrix is integer, without having to compute it and enumerate the entries. We say that a matrix $A \in \mathbb{R}^{m \times n}$ is called *total unimodular* if the determinant of all its square submatrices takes values in $\{-1, 0, 1\}$.

Theorem 5.2.1. *If A is total unimodular then A_B^{-1} is integer.*

Proof. The inverse of A_B can be written as

$$A_B^{-1} = \frac{1}{\det(A_B)} \begin{pmatrix} \alpha_{1B_1} & \cdots & \alpha_{1B_m} \\ \vdots & \ddots & \vdots \\ \alpha_{mB_1} & \cdots & \alpha_{mB_m} \end{pmatrix}, \tag{5.6}$$

where $\alpha_{ij} = (-1)^{i+j} \det(M_{ij})$ and M_{ij} is the submatrix obtained from A_B by deleting row i and column j.

Since A is total unimodular $\det(M_{ij}) \in \{-1, 0, 1\}$ and hence $\alpha_{ij} \in \{-1, 0, 1\}$, and also $\frac{1}{\det(A_B)} \in \{-1, 0, 1\}$. Therefore A_B^{-1} is integer. □

It is possible to frame the shortest path problem as an integer linear program, and to prove that the constraint matrix A is total unimodular so that it can be solved using the simplex method. However, more efficient methods are available as we will see next.

Chapter 6

Graphs

6.1 Introduction

The idea of a graph (not in the figure or plotting sense, but as a structure) was introduced in the 18th century to ask questions about paths. The idea now appears everywhere, but transport maps remain one of the key examples. A graph will abstract away all of the details and only show destinations (usually as points) and connecting paths between destinations (usually as lines), together with minimal extra information (such as the distance along a path, or the cost to travel along it).

The mathematical structure can be used to answer a number of detailed questions, but one key question is fundamental. Given a graph with paths linked to distances, what is the shortest distance between two given points?

6.2 Directed graphs

The mathematical definition of a *(directed) graph* is the ordered pair $G = (V, A)$, where the sets are

- the set of *vertices* (or *nodes*) V, with $|V| = n$;
- a set of ordered pairs $A \subseteq V \times V$, excluding self loops, called *arcs*, with $|A| = m$.

Given an arc $(i, j) \in A$, the node i is called the *tail* and node j the *head*. In a directed graph, travel is only possible from the tail to the head (so imagine drawing an arrow on the arc).

We have not explicitly said what space the nodes are drawn from. In general it does not matter. When giving mathematical examples it is typical to use the natural numbers, so $V = \{1, \ldots, n\}$, and the node number labels the destination. When thinking about concrete transport examples, it can make more sense to use a name or an alphanumeric code. A simple directed graph using integers is illustrated in figure 6.1.

6.2.1 Definitions

We call a node $j \in V$ *adjacent* to another node $i \in V$ if $(i, j) \in A$. As, by construction, there are no self-loops allowed, a node cannot be adjacent to itself.

We call an arc $a \in A$ *incident* to a node $j \in V$ if $a = (i, j)$ for some node $i \in V$. So an arc is incident on a node if there is some other node in the graph that uses the arc to move to the node.

We call a directed graph G *complete* if A contains an arc for each distinct pair of nodes,

$$A = \{(i, j) \colon \forall i \in V, j \in V, i \neq j\} . \tag{6.1}$$

If the graph is complete we can immediately travel from any node to any other node using just one arc.

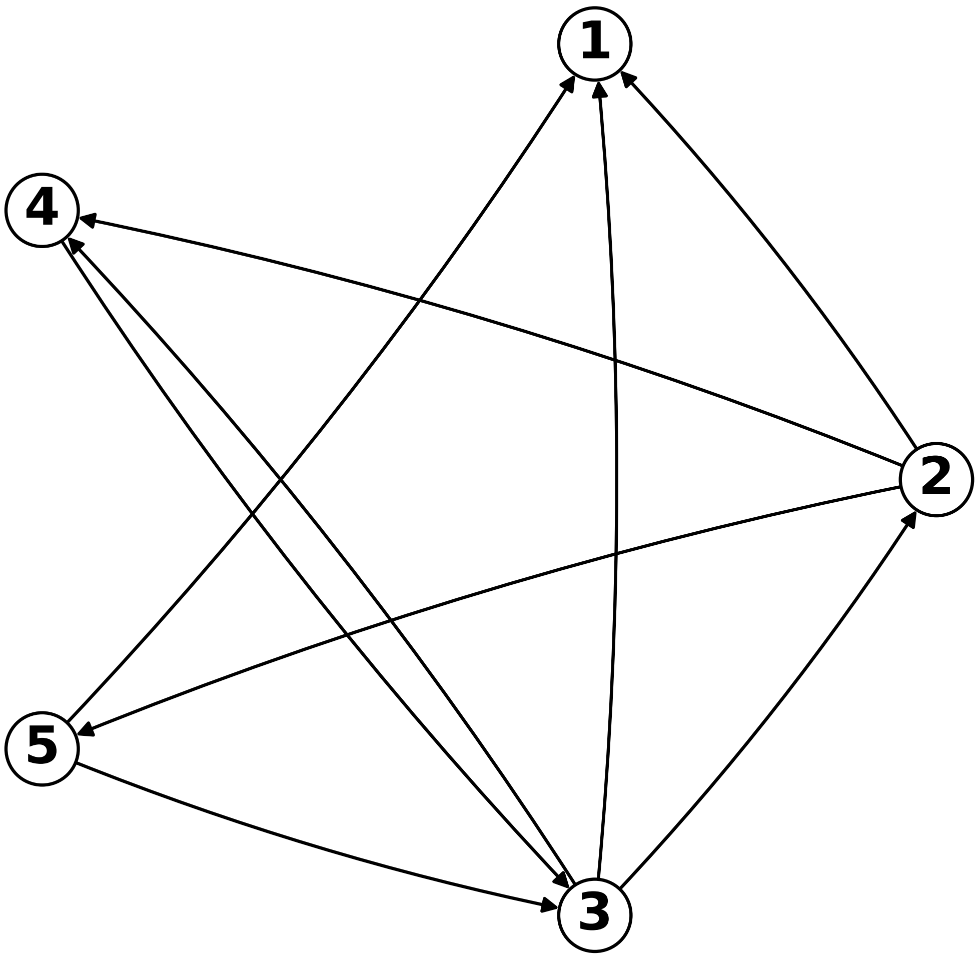

Figure 6.1: A simple directed graph. The vertices or nodes are the circles, here labelled by natural numbers. The edges are illustrated as arrows.

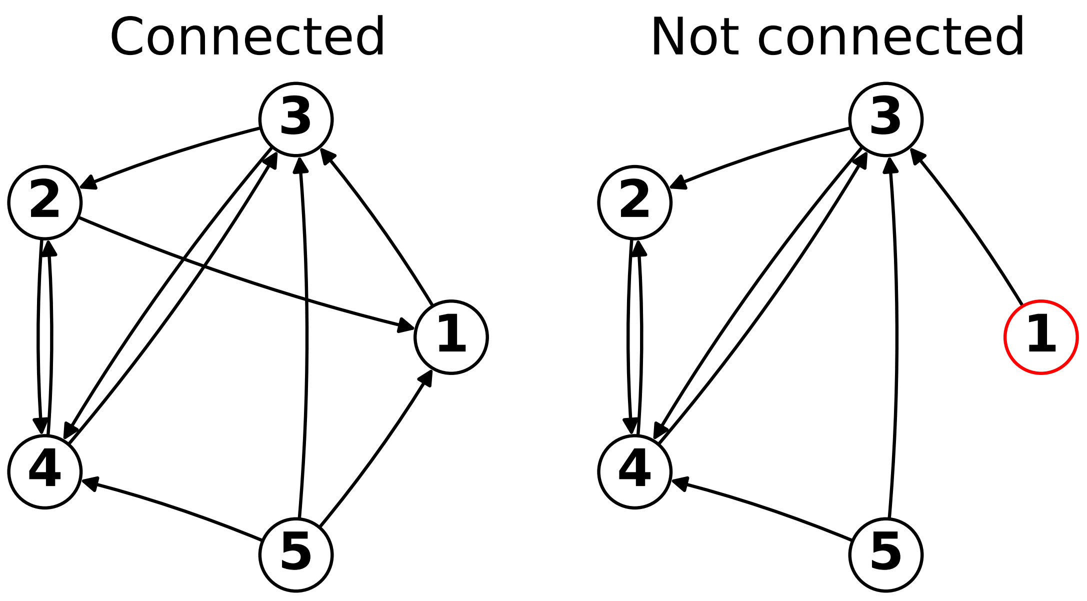

Figure 6.2: The graph on the left is connected but not complete. The graph on the right is not connected, as no edge goes in to node 1.

6.2.2 Number of arcs

Theorem 6.2.1. *In any directed graph, $m \leq n(n-1)$, with $m = n(n-1)$ in the complete case.*

Proof. In the complete case, each of the n nodes is adjacent to all of the other $(n-1)$ nodes: we have $m = n(n-1)$ arcs.

If the graph is not complete, some arcs are missing. Hence $m \leq n(n-1)$. □

We called a directed graph *sparse* if $m \ll n(n-1)$.

6.2.3 Paths

If the graph is not complete then traversing a single arc may not be enough to get us from our start point to the desired end point. Instead we may have to traverse multiple arcs.

We define a *path* as a sequence

$$P = (i_1, i_2), (i_2, i_3), \ldots, (i_{k-1}, i_k), (i_k, i_{k+1}) \tag{6.2}$$

of $k \geq 1$ consecutive and distinct arcs. Note that the head of each arc in the sequence matches the tail of the next arc, making the arcs consecutive.

Given a path, we say that a node $v \in V$ is *connected* to a node $w \in V$ if there is a path P with $i_1 = v$ and $i_{k+1} = w$.

Finally, we say that a graph is *connected* if every pair of its nodes is connected. This is illustrated in figure 6.2.

6.2.4 Cuts

So far everything has been set either at the level of individual nodes or at the level of the whole graph. For building algorithms that allow us to analyse paths as they move through the graph, we need to look at subsets of the graph, and how we might move into or out of those subsets.

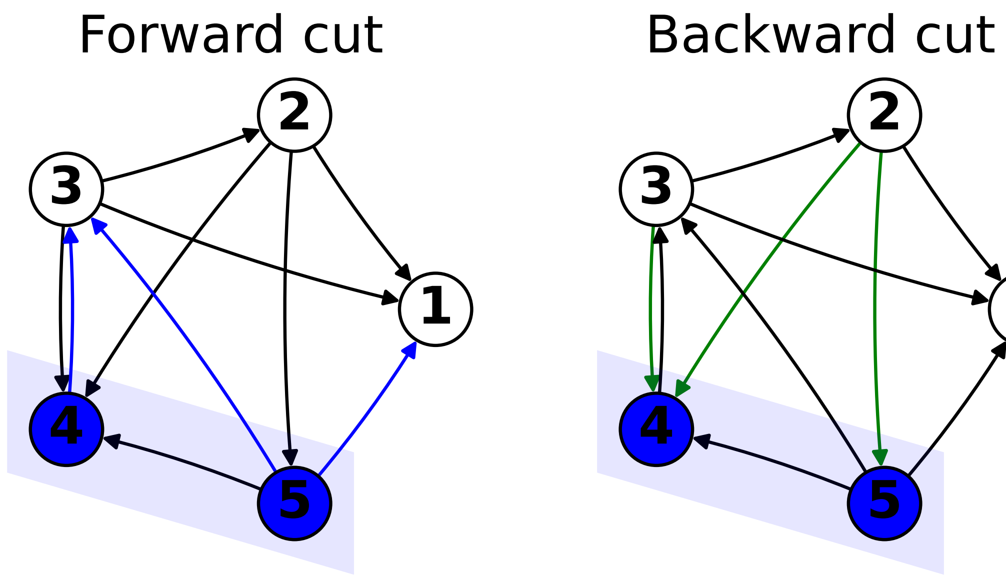

Figure 6.3: The set $S = \{4, 5\}$ is shaded in blue. Its forward cut $\delta^+(S)$ is every edge (highlighted in blue in the left plot) that leaves S. Its backward cut $\delta^-(S)$ is every edge (highlighted in green in the right plot) that enters S.

Let $S \subseteq V$ be some part of graph (by looking at some suset of the vertices). We define the *forward cut induced by* S to be the set of arcs "leaving S". That is

$$\delta^+(S) = \{(i, j) \in A \colon i \in S \text{ and } j \in V \setminus S\} . \tag{6.3}$$

Similarly, we define the *backward cut induced by* S to be the set of arcs "entering S". That is

$$\delta^+(S) = \{(i, j) \in A \colon i \in V \setminus S \text{ and } j \in S\} . \tag{6.4}$$

These are illustrated in figure 6.3.

Of course, we can apply these definitions to the case to single nodes, by setting $S = \{i\}$ for any node $i \in V$. These have special names: the *forward star* of the node i is $\delta^+(i)$, whilst the *backward star* of the node i is $\delta^-(i)$. The sizes of these sets also have special names, with the *out-degree* of node i being $|\delta^+(i)|$, and the *in-degree* of node i being $|\delta^-(i)|$.

6.2.5 Representations

So far we have represented the graph using the vertices V and the arcs A. However, as in the linear programming case, there are multiple ways of representing the problem, each of which has its own advantages.

For simplicity these additional representations will assume that V is given by consecutive integers labelling the nodes. Depending on choice these can either start from 1 (so $V = \{1, \ldots, n\}$) or 0 (so $V = \{0, \ldots, n - 1\}$). The latter is more natural for a Python implementation, but the former more natural in many mathematical texts. With this assumption we only need to consider different representations of the arcs, A.

Adjacency list

In the *adjacency list* approach we construct a list L of size n. Each component of the list L_i contains a list of size at most $n - 1$, containing the indices of the nodes adjacent to i. In terms of the stars, we have

$$L_i = \{j \colon (i,j) \in \delta^*(i)\}. \tag{6.5}$$

The advantage of the adjacency list approach is that it is easy to use to navigate the graph. The component L_i gives every node that is reachable from node i using a single arc.

Adjacency matrix

In the *adjacency matrix* approach we construct a single matrix of size $n \times n$ that contains only zeros and ones. The matrix entry is one if the associated arc exists and zero otherwise. Explicitly,

$$m_{ij} = \begin{cases} 1 & \text{if } (i,j) \in A \\ 0 & \text{otherwise} \end{cases}. \tag{6.6}$$

The adjacency matrix is less efficient for navigating the graph, but more efficient if we need to check if an arc exists.

6.3 Shortest paths

Now that we have the terminology to represent and discuss graphs, we want to turn the word problem "Find the shortest path connecting two points" into something precise.

6.3.1 The problem

The *shortest path problem* is as follows. Given a directed graph $G = (V, A)$ with a (non-negative) length function $l \colon A \to \mathbb{R}^+$ and two nodes $s, t \in V$, find an $s - t$ path of shortest total length.

The length function tells us the distance along any one single arc: equivalently, the distance between any two nodes or vertices when moving in a specific direction. This need not be the same in both directions (hence the notion of a *directed* graph): think about one-way streets, for example. As a notational shortcut we will often talk about lengths of paths in addition to lengths of arcs. As a path is a sequence of consecutive arcs without loops and the length function is non-negative, it follows that

$$l(P) = \sum_{a \in P} l(a). \tag{6.7}$$

If the arc $a = (i,j)$ then sometimes the notation $l_{ij} = l((i,j)) = l(a)$ is used, and so we can write

$$l(P) = \sum_{(i,j) \in P} l_{ij}. \tag{6.8}$$

The two nodes in the problem are the *source* node s from which we start and the *target* node t that we are trying to get to.

There is a generalisation of the problem which is, in fact, no harder to solve. The *single source shortest path problem* is as follows. Given a directed graph $G = (V, A)$ with a (non-negative) length function $l \colon A \to \mathbb{R}^+$ and one node $s \in V$, find a path between s and every other node in $V \setminus \{s\}$ of shortest total length.

We note that it only makes sense to solve the shortest path problem if the source and target nodes are connected. Therefore it only makes sense to solve the single source shortest path problem if the entire graph is connected. Throughout the rest of this chapter we will assume that the graph is connected.

6.3.2 Subpath optimality

The key result in finding the shortest path is intuitively stated as "shortest paths are composed of shortest paths". This seems either obvious or trivial. The precise result is

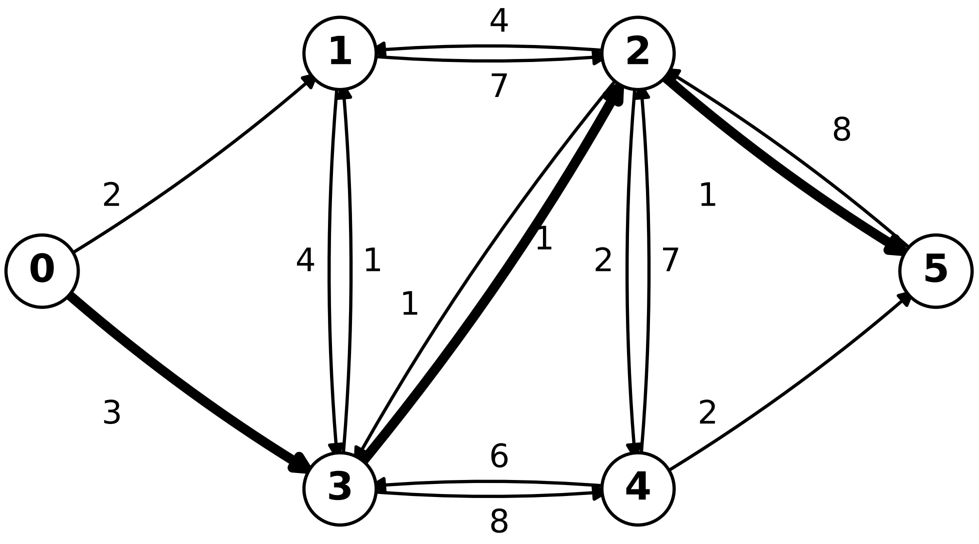

Figure 6.4: A graph with its shortest path. Each node is annotated with its label (a natural number here). Each edge (i, j) is annotated with its length l_{ij}. The bold lines show the shortest $0 - 5$ path, $P = (0, 3), (3, 2), (2, 5)$.

Theorem 6.3.1 (Subpath Optimality)**.** *Let $P = (i_1, i_2), (i_2, i_3), \dots, (i_{k-1}, i_k)$ be an $i_1 - i_k$ shortest path. For any pair of nodes i_u, i_v visited by P, with $u < v$, the subpath S from i_u to i_v is a shortest $i_u - i_v$ path.*

Proof. This follows by contradiction. if S is not a shortest $i_u - i_v$ path then there is another $i_u - i_v$ path S' that is strictly shorter than S, $l(S') < l(S)$. We can therefore define a new $i_1 - i_k$ path P' by using the "shortcut" S',

$$P' = P \setminus S \cup S'. \tag{6.9}$$

From the positivity of the length function it immediately follows that

$$l(P') = l(P) - l(S) + l(S') < l(P). \tag{6.10}$$

Therefore P was not the shortest path, and we have a contradiction. □

The original characterisation can now be more precisely stated as "every subpath of a shortest path is itself a shortest path".

6.4 Dijkstra's theorem

This is the central result that allow us to build an algorithm to solve the shortest path problem.

Theorem 6.4.1 (Dijkstra's theorem)**.** *Let $S \subseteq V$ be a subset of the vertices that contains the source s. Let $Y(i)$, for all $i \in V$, be the length of the corresponding shortest $s - i$ path. Let*

$$(v, w) \in \underset{(i,j)\in\delta^+(S)}{\operatorname{argmin}} \left[Y(i) + l_{ij} \right]. \tag{6.11}$$

Then $\varphi = P_{(s-v)} \cup \{(v, w)\}$ is a shortest $s - w$ path, where $P_{(s-v)}$ is a shortest $s - v$ path.

Proof. We show that any other path π to any other vertex $u \in V \setminus S$ has either the same length as φ or a strictly larger one.

First, decompose the path π as

$$\pi = \pi_1 \cup \{(i,j)\} \cup \pi_2, \tag{6.12}$$

where $i \in S$, $j \in V \setminus S$, π_1 is a shortest $s - i$ path, $(i,j) \in \delta^+(S)$, and π_2 is a shortest $j - u$ path. This is always possible due to the subpath optimality theorem. It allows us to concentrate on the arc (i,j) that leaves the set S. It follows that

$$l(\pi) = l(\pi_1) + l_{ij} + l(\pi_2). \tag{6.13}$$

Since we have chosen $(v,w) \in \delta^+(S)$ to minimise $Y(i) + l_{ij}$ it must be true that

$$l(\pi_1) + l_{ij} \geq Y(v) + l_{vw}. \tag{6.14}$$

Since $l(\pi_2) \geq 0$ it follows that

$$\begin{aligned} l(\pi) &= l(\pi_1) + l_{ij} + l(\pi_2) && (6.15)\\ &\geq l(\pi_1) + l_{ij} && (6.16)\\ &\geq Y(v) + l_{vw} && (6.17)\\ &= l(\varphi). && (6.18)\end{aligned}$$

Therefore $l(\pi) \geq l(\varphi)$ for all such paths, and therefore φ is a shortest $s - w$ path. □

6.4.1 Dijkstra's algorithm

We can now use subpath optimality and iteratively apply Dijkstra's theorem to give us an algorithm. Starting from the source node, we keep adding nodes to the set of nodes we have "seen" so that we have the shortest path in that set.

Being more precise, but still sketching in words. We want to find the shortest path from the source node s to every other node in the graph. We construct a set $S \subseteq V$, starting from $S = \{s\}$, of all the nodes we have so far "seen". We also need two *labels*. The first is $Y(i)$, the shortest path length from s to i. The second is $P(i)$, the predecessor of i in the $s - i$ path. At the start $Y(s) = 0$ and $P(s)$ is undefined (as it makes no sense).

We then iterate as long as $|S| < n$, so that $V \setminus S$ is not empty. For each iteration we

1. find $(v,w) \in \delta^+(s)$ that minimises $Y(v) + l_{vw}$;
2. set $Y(w) = Y(v) + l_{vw}$;
3. set $P(w) = v$;
4. set $S \to S \cup \{w\}$.

Note that, for every node i, the shortest path length from the source $Y(i)$ adn the predecessor node on the shortest path $P(i)$ are set only once, when the final shortest arc to i is found and i is added to S.

This solves the *single source* shortest path problem. To solve the shortest path problem when the target t is specified we can check to see if $w = t$ and, if so, stop at the end of that iteration. This makes no difference to the code complexity as, in the worst case that has to be considered, t will be the last node added to S.

6.4.2 Representing the length

We have seen multiple methods for representing the topology of the graph, each of which has their individual advantages. As Dijkstra's algorithm depends crucially on minimising a length and looking up l_{vw}, it is important to have an easy and efficient way of representing the arc lengths.

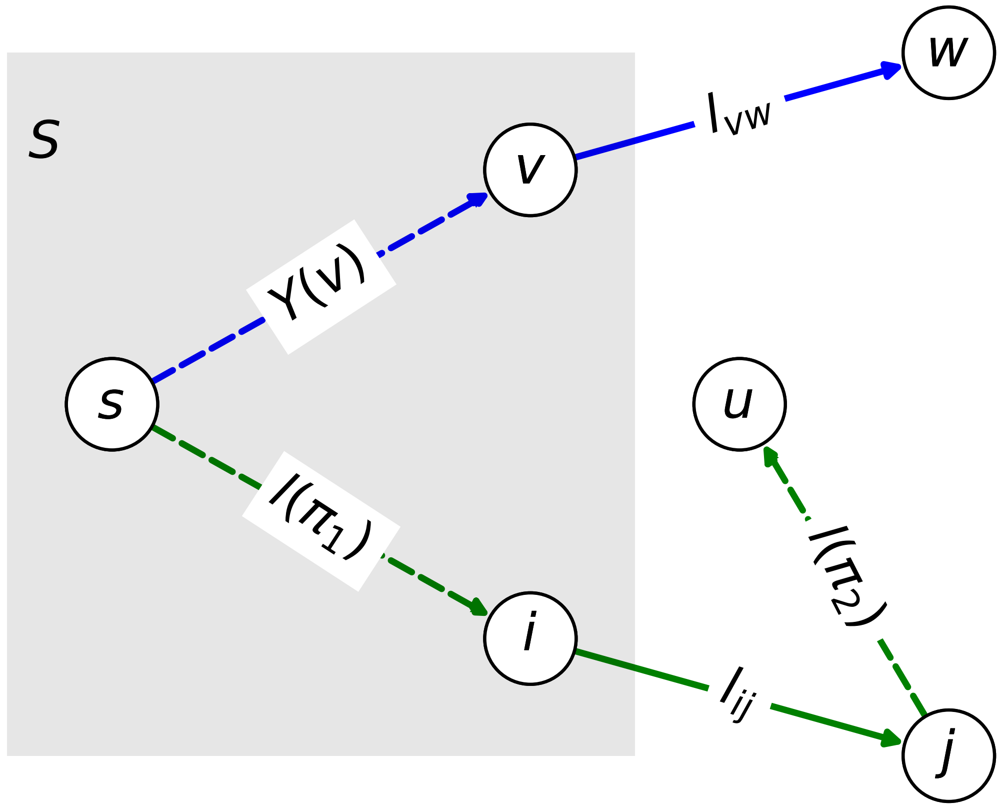

Figure 6.5: The key idea used in proving Dijkstra's theorem. We want to show that the shortest path from s to w is given by the shortest $s-v$ path (already found an in S, by assumption) linked to the $v-w$ edge. This must be true if the length of this path is no greater than the length of any other path connecting any point not in S to s.

Adjacency list

As in the graph case, we construct a list L_l of size n, where each component i contains another list of size at most $n-1$, containing the *lengths* of all arcs $(i,j) \in \delta^+(i)$, in order of index j.

This representation needs the adjacency list L as well in order to give the index j. Assuming that V is a set of consecutive integers of size n, the whole graph (including arc length information) is given once L, L_l are given. The size of the graph n is implicit in the length of the lists.

Adjacency matrix

Again this is similar to the graph case. We construct a matrix $M_l \in \{\mathbb{R} \cup \{\infty\}\}^{n \times n}$ where, for each $i, j \in V$, the component $(m_l)_{ij}$ is the length of the (i,j) arc, or is infinity if no such arc exists.

This representation implicitly stores all the information about the graph, including its topology, so the adjacency matrix M is not needed.

6.4.3 Implementing Dijkstra's algorithm

The adjacency list form is the most efficient for checking whether an arc exists. The adjacency matrix form is the most efficient for checking what the length of the arc is. We assume that we are not limited by the amount of memory that each form takes, so we will use both forms in our implementation. In the code snippet below, `L` is the adjacency list representing the topology of the graph and `Ml` is the adjacency matrix representation of the length function.

```
 1 S = [s]
 2 Y = np.empty(n)  # Value does not matter
 3 P = np.empty(n)  # Only size matters
 4 Y[s] = 0
 5 while len(S) < n:
 6     min_Y = np.inf
 7     v, w = -1, -1
 8     for i in S:
 9         if j in S:  # Only consider arcs leaving S
10             continue
11         if min_Y > Y[i] + Ml[i][j]:
12             min_Y = Y[i] + Ml[i][j]
13             v, w = i, j
14     Y[w] = Y[v] + Ml[v][w]
15     P[w] = v
16     S.append(w)
```

6.4.4 Complexity

To compute the complexity we consider the order of each line or block of the code above. We have

- Line 1 writes a single value, taking $\mathcal{O}(1)$;
- Lines 2-3 write n values, taking $\mathcal{O}(n)$ each;
- Line 4 writes a single value, taking $\mathcal{O}(1)$;
- Line 5 loops over the whole graph, so is executed n times. Each block takes:
  - Lines 6-7 write single values, taking $\mathcal{O}(1)$;
  - Lines 8-10 correspond to looping over $\delta^+(S)$ in its entirety. This is executed m times, where m is the out degree of S. Earlier results show $m \leq n(n-1)$. In the worst case $m = \mathcal{O}(n^2)$. Each block takes:

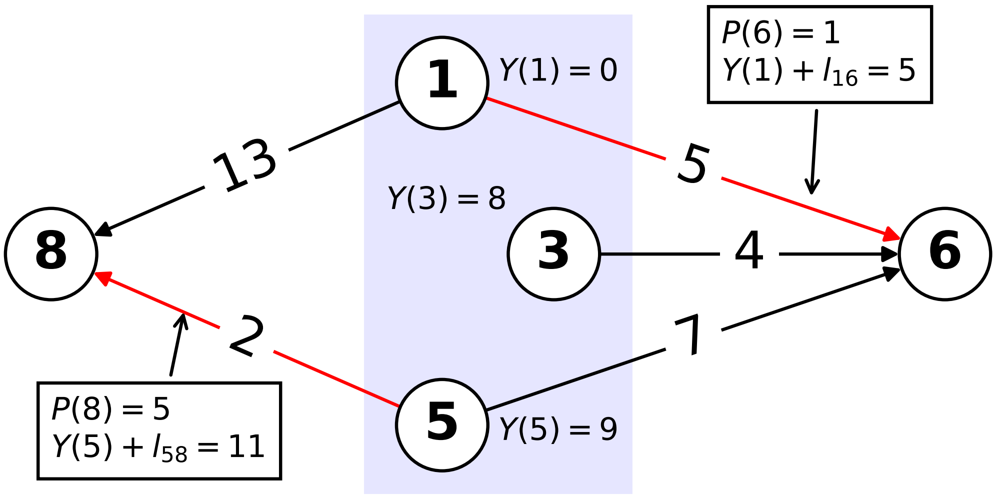

Figure 6.6: The key idea used in constructing the faster $\mathcal{O}(n^2)$ version of Dijkstra's algorithm. When trying to add to the set S (blue shaded region) we must compute the shortest path and predecessors to the points not in S, given the current S (red edges). By storing these values and updating them when needed, instead of re-computing every iteration, we save lots of computation.

  * Lines 11-13 are single comparisons or single writes, each taking $\mathcal{O}(1)$;

  Therefore this inner loop has complexity $\mathcal{O}(m) = \mathcal{O}(n^2)$.

- Lines 14-16 are single writes, each taking $\mathcal{O}(1)$;

Therefore this outer loop has complexity $\mathcal{O}(n)\mathcal{O}(n^2) = \mathcal{O}(n^3)$.

Therefore the whole algorithm has complexity $\mathcal{O}(n^3)$.

This is considerably faster than the complexity of the simplex algorithm (in its worst case).

6.4.5 A faster method

Finding the shortest path is a sufficiently important practical problem that we want to find an algorithm faster than $\mathcal{O}(n^3)$. This can be done. The key issue is that by repeatedly scanning $\delta^*(S)$ we are recomputing particular shortest path lengths multiple times. With additional structures we can reduce that computation.

We need the following observation:

$$(v,w) \in \underset{(i,j)\in\delta^+(S)}{\operatorname{argmin}} \{Y(i) + l_{ij}\} \tag{6.19}$$

$$\iff \quad (v,w) \in \underset{j\in V\setminus S}{\operatorname{argmin}} \left\{ \underset{(i,j)\in\delta^-(S)\colon i\in S}{\operatorname{argmin}} \{Y(i) + l_{ij}\} \right\}. \tag{6.20}$$

The point here is that if we already know, for given node $j \in V \setminus S$, which arc minimises $Y(i) + l_{ij}$, then we can find the arc (v,w) giving the shortest path by looking at the $\mathcal{O}(n)$ vertices $j \in V \setminus S$, rather than looking at the $\mathcal{O}(m) = \mathcal{O}(n^2)$ arcs in $\delta^+(S)$.

To use this observation to construct an algorithm, we change the way we think about the labels $Y(j), P(j)$, representing the length of the shortest $s - j$ path and the predecessor index in that shortest path respectively. In the original algorithm they were set *only when* j enters S. Until that point they had no value, or their value was meaningless.

In the new algorithm, the meaning of the labels will remain the same *as long as* $j \in S$. If $j \in V \setminus S$ then the value of $Y(j)$ and $P(j)$ will represent the length and predecessor index of the shortest $s - j$ path found *up to the current iteration.* At each iteration we will check *and update* the value of the labels when a new vertex enters S.

```
 1 S = [s]
 2 Y = np.copy(Ml[s][:])   # These values now matter
 3 P = np.zeros(n)         # These also now matter
 4 Y[s] = 0
 5 while len(S) < n:
 6     min_Y = np.inf
 7     w = -1              # Note: only w, not v
 8     for j in V:         # Note: change here
 9         if j in S:  # Only consider arcs leaving S
10             continue
11         if min_Y > Y[j]:
12             min_Y = Y[j]
13             w = j
14     S.append(w)
15     for h in L[w]:  # New step: update labels
16         if h in S:
17             continue
18         if Y[h] > Y[w] + Ml[w][h]:
19             Y[h] = Y[w] + Ml[w][h]
20             P[h] = w
```

The explicit complexity calculation is

- Line 1 writes a single value, taking $\mathcal{O}(1)$;
- Lines 2-3 write n values, taking $\mathcal{O}(n)$ each;
- Line 4 writes a single value, taking $\mathcal{O}(1)$;
- Line 5 loops over the whole graph, so is executed n times. Each block takes:
  - Lines 6-7 write single values, taking $\mathcal{O}(1)$;
  - Lines 8-10 correspond to looping over $V \setminus S$ in its entirety. In the worst case this is $\mathcal{O}(n)$. Each block takes:
    * Lines 11-13 are single comparisons or single writes, each taking $\mathcal{O}(1)$;

    Therefore this inner loop has complexity $\mathcal{O}(n)$.
  - Lines 14 is a single write taking $\mathcal{O}(1)$;
  - Lines 15-17 correspond to looping over $V \setminus S$ in its entirety again. In the worst case this is $\mathcal{O}(n)$. Each block takes:
    * Lines 18-20 are single comparisons or single writes, each taking $\mathcal{O}(1)$;

    Therefore this inner loop has complexity $\mathcal{O}(n)$.

  Therefore this outer loop has complexity $\mathcal{O}(n)\mathcal{O}(n) = \mathcal{O}(n^2)$.

Therefore this algorithm has complexity $\mathcal{O}(n^2)$. When the graph is large this can be a substantial improvement in efficiency.

Note that this is not the fastest algorithm that exists. It is possible to construct algorithms with complexities $\sim \mathcal{O}(n \log(n))$.