Optimization

Course Notes corresponding to parts of the Course 2MMD10 and 2DME20

Cor Hurkens, Frits Spieksma

Eindhoven University of Technology

October 8, 2024

Contents

Preface

These course notes correspond to the course "2MMD10 Optimization". Its subject is unconstrained optimization, constrained optimization, and discrete optimization. The latter two subjects are covered in these course notes. We expect that each of the five chapters corresponds to one week of teaching (where one week of teaching corresponds to two teaching sessions and one or two instruction sessions; a session is two times 45 minutes).

The course assumes that the reader has had a first introduction to linear algebra, linear programming, operations research and has some elementary knowledge of mathematical modeling, linear programming, and graph theory. In particular, the courses 2WF20 or 2DE20 (Linear Algebra) are a prerequisite, and the courses 2WO20 or 5EMA0 (Linear Optimization) as well as 2WF60 (Graph Theory) are recommended.

As a disclaimer, each of the subjects covered here constitute fundamental topics in themselves to which a large number of books have been devoted. Here, we will only scratch the surface, and cover the essentials. Moreover, we will do so in an informal way; there exist more specialized courses offering a more specialized treatment, for instance courses organized under auspicies of MasterMath or the LNMB.

The book of Boyd and Vandenberghe [3] has served as inspiration for Part I, as well as Woeginger's slides, adapted by Hurkens and Nederlof, and their exercises. A famous, classic book on convex optimization is Rockafellar [11].

For Part II, there are some excellent textbooks on combinatorial optimization. We mention Schrijver's trilogy: "Combinatorial Optimization. Polyhedra and Efficiency" [12]. Another example is: "Combinatorial optimization" by Cook, Cunningham, Pulleyblank, and Schrijver [5]. Other related textbooks are: Nemhauser and Wolsey [10], and Wolsey [14].

Part I

Constrained Optimization

Constrained optimization, as opposed to unconstrained optimization is defined as finding a 'best' solution x from a restricted set X. The quality of a solution is measured by evaluating an objective function $f_0 : X \mapsto \mathbb{R}$. We call a particular solution x^* best if x^* achieves the lowest possible value over all solutions in X, i.e., if $f_0(x^*) \leq f(x)$ for all $x \in X$. Notice that this is without loss of generality, since minimizing f_0 is equivalent to maximizing $-f_0$.

We usually assume that f_0 is a continuous function. In order to describe the feasible set of solutions X, we distinguish between continuous and discrete (or combinatorial) structures. We start from a description with continuous functions $f_i : \mathbb{R}^n \mapsto \mathbb{R}$ $(i = 1, \ldots, m)$, and $h_j : \mathbb{R}^n \mapsto \mathbb{R}$ $(j = 1, \ldots, r)$, assuming that $X \subseteq \mathbb{R}^n$. Then, a general way to describe the set of feasible solutions is:

$$X = \{x \in \mathbb{R}^n |\ f_i(x) \leq 0, \text{for } i = 1, \ldots, m; h_j(x) = 0, \text{for } j = 1, \ldots, r\}.$$

Even if we assume that the functions f_i and h_j are continuous, the resulting formulation is too general to come up with satisfying solution techniques. For example, the equations $h_j(x) = 0$ may contain constraints like $x_j(1 - x_j) = 0$, implying that, in each feasible solution, $x_j = 0$ or $x_j = 1$ $(1 \leq j \leq r)$. This means that even discrete optimization problems can be formulated using these constraints. In any case, without any additional assumptions on the f_i's and the h_j's, solving the resulting optimization problem is hard. For these reasons, we investigate properties of functions that make these optimization problems more tractable. One key property, especially relevant for constrained optimization, is convexity, the subject of Chapter 1. Chapter 2 is devoted to conditions for optimality.

Chapter 1

Convexity

In the world of mathematical optimization, there is probably no concept more fundamental than convexity. Rockafellar phrases this in SIAM Review (1993) as follows: "in fact, the great watershed in optimization isn't between linearity and nonlinearity, but convexity and nonconvexity".

Simply put, if the function to be optimized is a convex function, and if the feasible region to be optimized over, is a convex region, then, well then, efficient algorithms for finding a minimum-cost solution exist. And although the notions of efficiency, and convexity will be formally defined later in this chapter, this informal statement motivates our goal for this chapter: understanding, recognizing, and being able to "play" with convexity.

1.1 A motivating theorem

The fundamental role of convexity in optimization is illustrated in the following result.

Consider the following optimization problem.

$$\text{minimize}_x f(x) \text{ subject to } x \in C,$$

where x is an $n \times 1$ vector of decision variables in $\mathbb{R}^n$, and where the function f, as well as the set $C \subseteq \mathbb{R}^n$, are given and convex (we give formal definitions later). Here is a powerful statement.

Theorem 1.1.1 *In case f is a convex function, and C is a convex set: each local minimum is a global minimum.*

We state the proof of this theorem in spite of the fact that definitions of relevant concepts still have to come.

Proof: Let x^* be a local minimum, so there is some $\varepsilon > 0$ such that $f(y) \geq f(x^*)$ for all $y \in B_\varepsilon(x^*) \cap C$. We proceed by using contradiction. Thus, suppose that x^* is not a global minimum. Then, there is some point $z \in C$ with $f(z) < f(x^*)$. Then pick $\alpha > 0$ small enough so that $w \equiv \alpha z + (1-\alpha)x^*$ is in $B_\varepsilon(x^*)$. By convexity of C, we have $w \in C$. And by convexity of f, we have

$$f(w) \leq \alpha f(z) + (1-\alpha) f(x^*) < f(x^*),$$

which is a contradiction to the fact x^* is a local minimum. □

Notice that both convexity of f, as well as convexity of C are needed.

- Question: does the argument above also work when the word "minimum" in Theorem 1.1.1 is replaced by "maximum"?
- Question: consider an optimization problem of the form given above where each local optimum is a global optimum. Are f and C necessarily convex?

1.2 Basics

1.2.1 Combinations

Let us recall some fundamental concepts from linear algebra. We point out that the two conditions (i) if $x \in L$ and $y \in L$, then $x + y \in L$, and (ii) if $x \in L$, then $\alpha x \in L$ for all $\alpha \in \mathbb{R}$ are equivalent to the condition: if $x \in L$ and $y \in L$, then $\alpha x + \beta y \in L$ for $\alpha, \beta \in \mathbb{R}$.

When given two distinct points $x, y \in \mathbb{R}^n$ and two scalars $\alpha, \beta \in \mathbb{R}$, we can construct various types of combinations.

Definition 1.2.1 *$z := \alpha x + \beta y$ is called a* linear *combination of x and y.*

Geometrically, this means that the point z lies on the plane that goes thru the origin, x and y. By placing conditions on α and β various special linear combinations arise:

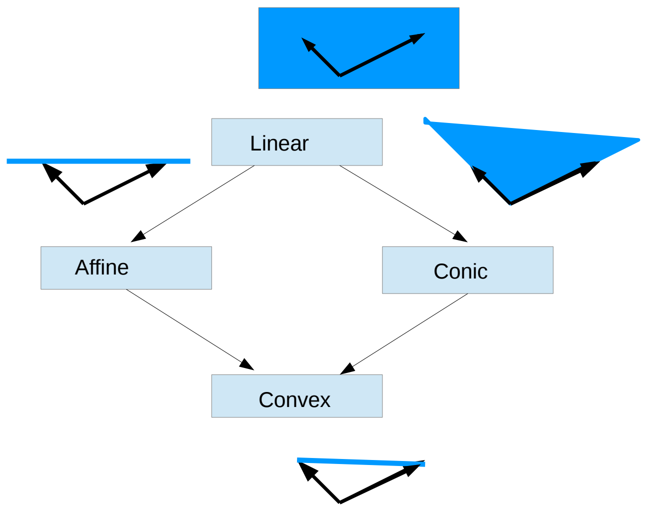
Definition 1.2.2 *If $\alpha + \beta = 1$, then $z := \alpha x + \beta y$ is called an* affine *combination of x and y.*

Geometrically, this means that the point z lies on the line that goes thru the two points x and y.

Definition 1.2.3 *If $\alpha, \beta \geq 0$, then $z := \alpha x + \beta y$ is called a* conic *combination of x and y.*

Definition 1.2.4 *If $\alpha + \beta = 1$ and $\alpha, \beta \geq 0$, then $z := \alpha x + \beta y$ is called a* convex *combination of x and y.*

Geometrically, a point z being a convex combination of points x and y, means that point z lies on the line segment between x and y. Clearly, when point z is a convex combination of two given points x, y, then it is also both an affine combination, as well as a conic combination of x and y.



1.2.2 Sets

The four definitions from the preceding section correspond to definitions of types of sets. Indeed, a set L is {linear, affine, conic, convex} if, for each pair of points in L, its corresponding {linear, affine, conic, convex} combination is also in L. More precisely:

Definition 1.2.5 *A set $L \subseteq \mathbb{R}^n$ is* linear *if, for all $x, y \in L$, we have: $\alpha x + \beta y \in L$ for all $\alpha, \beta \in \mathbb{R}$.*

Definition 1.2.6 *A set $L \subseteq \mathbb{R}^n$ is* affine *if, for all $x, y \in L$, we have: $\alpha x + \beta y \in L$ for all $\alpha, \beta \in \mathbb{R}$ with $\alpha + \beta = 1$.*

Clearly, linear sets are affine sets (since linearity of a set means, by definition, that for each $x, y \in L$ it is true that, for each α, β, we have $\alpha x + \beta y \in L$. Thus, in particular, for those α, β that satisfy $\alpha + \beta = 1$).

Theorem 1.2.1 *The following are equivalent. For nonempty $L \subseteq \mathbb{R}^m$:*

1. *L is an affine set,*
2. *$L = \{x \mid Ax = b\}$ for some $k \times m$ matrix A and $b \in \mathbb{R}^k$, and*
3. *$L = \{Cx + d \mid x \in \mathbb{R}^n\}$ for some $m \times n$ matrix C and $d \in \mathbb{R}^m$.*

Proof:

(1) $\Rightarrow$ (3): Pick $d \in L$. (If $L = \{d\}$, we are done: take for C the $m \times 1$ null-matrix, $n = 1$). Consider the set $L_d = \{x - d \mid x \in L\}$. Observe that L_d is a linear subspace. Indeed, let u,$v \in L_d$, and $\alpha, \beta \in \mathbb{R}$. Then $d, d+u, d+v$ are in L. As L is affine, we have that $d + 2\alpha u = (1 - 2\alpha)d + 2\alpha(d + u) \in L$; similarly $d + 2\beta v \in L$. Hence $d + \alpha u + \beta v = \frac{1}{2}(d + 2\alpha u) + \frac{1}{2}(d + 2\beta v) \in L$, and so $\alpha u + \beta v \in L_d$.
Consider any basis of L_d, of dimension n say, and let its vectors form the columns of an m by n matrix C. Then for each $y \in L_d$, there is an $x \in \mathbb{R}^n$ such that $y = Cx$, so $L = \{d + Cx \mid x \in \mathbb{R}^n\}$.

(3) $\Rightarrow$ (2): Consider $L_{C,d} = \{d + Cx \mid x \in \mathbb{R}^n\}$, for some $m \times n$ matrix C. If $k := m - rk(C) = 0$, then C has rank m, and $L_{C,d} = \mathbb{R}^m = \{x \mid [0\,0 \ldots 0]x = 0\}$. If $k > 0$, there is a $k \times m$ matrix M of full row rank, such that $MC = 0$. That is, the rows of M span the null-space of C^T. Take $A = M$, then, for each $y = Cx + d$ we have: $Ay = MCx + Md = b$, where A is a $k \times m$ matrix, and $b = Md \in \mathbb{R}^k$.

(2) $\Rightarrow$ (1): Consider two distinct solutions x_1, x_2, each satisfying $Ax = b$. Also consider $y = \alpha x_1 + \beta x_2$ for some α, β with $\alpha + \beta = 1$. Since:

$$Ay = A(\alpha x_1 + \beta x_2) = \alpha A x_1 + \beta A x_2 = \alpha b + \beta b = b,$$

it follows that $y \in L$. Hence L is an affine set. $\square$

In particular, Theorem 1.2.1 tells us that the solution space of a system of linear equalities is an affine set, and vice versa.

Definition 1.2.7 *A set $L \subseteq \mathbb{R}^n$ is a* cone *if, for all $x, y \in L$, we have: $\alpha x + \beta y \in L$ for all $\alpha, \beta \geq 0$.*

Clearly, linear sets are cones. Other examples of cones are:

- the Lorentz cone $L^{n+1} = \{(x,t) \mid x \in \mathbb{R}^n, t \in \mathbb{R}, ||x|| \leq t\}$,
- the class of positive semi-definite matrices $S_+^n = \{A \in S^n | \; A \succeq 0\}$.

As a reminder, we use the symbol S^n to denote the class of $(n \times n)$ symmetric matrices. A symmetric matrix A is called *positive semi-definite* if the scalar $x^T Ax \geq 0$ for every $x \in \mathbb{R}^n$. All eigenvalues of a positive semi-definite matrix are nonnegative, see Section 1.6 for more information on positive semi-definite matrices.

Definition 1.2.8 *A set $L \subseteq \mathbb{R}^n$ is* convex *if, for all $x, y \in L$, we have: $\alpha x + \beta y \in L$ for all $\alpha, \beta \in \mathbb{R}$ with $\alpha + \beta = 1$ and $\alpha, \beta \geq 0$.*

Observe that the most demanding of these last four definitions is Definition 1.2.5. Clearly, both a cone, as well as an affine set must be convex. Other examples of convex sets are:

- a hyperplane $H_{a,b} = \{x \in \mathbb{R}^n | \; a^T x = b\}$ (which is even an affine set),
- a halfspace $H_{a,b}^{\leq} = \{x \in \mathbb{R}^n | \; a^T x \leq b\}$,
- the unit ball $B^n = \{x \in \mathbb{R}^n | \; ||x|| \leq 1\}$.

The validity of each of these statements is easily illustrated by applying the definition of a convex set. Take for instance the unit ball, and consider $x, y \in B^n$. Thus, $||x|| \leq 1$ and $||y|| \leq 1$. Consider now $\alpha x + \beta y$ for some $\alpha, \beta \geq 0$ and $\alpha + \beta = 1$. Since $||\alpha x + \beta y|| \leq ||\alpha x|| + ||\beta y|| = \alpha ||x|| + \beta ||y|| \leq \alpha + \beta = 1$, it follows that $\alpha x + \beta y \in B^n$, and hence B^n is a convex set.
Question: where exactly does the argument fail to prove that B^n is an affine set?

1.2.3 Norms

Norms are functions that intend to represent *distance.* The presence of the following properties make a function a norm.

- $f(x) \geq 0$ for all $x \in \mathbb{R}^n$,
- $f(x) = 0 \Leftrightarrow x = 0$,

- $f(\lambda x) = \lambda f(x)$ for all $\lambda \in \mathbb{R}_+, x \in \mathbb{R}^n$, and

- $f(x + y) \le f(x) + f(y)$ for all $x, y \in \mathbb{R}^n$.

Definition 1.2.9 *If f is a norm, then*

- *the norm ball is $\{x \in \mathbb{R}^n \mid f(x) \le 1\}$, and*

- *the norm cone is $\{(x,t) \in \mathbb{R}^{n+1} \mid f(x) \le t\}$.*

For any norm, the norm ball is a convex set, and the norm cone is even a cone.

1.3 Convexity-preserving operations

There are various operations one can apply to a set. Here we describe examples of two different operations on sets that do not impact the convexity of the resulting set.

1.3.1 Intersection

Consider two convex sets $S_1, S_2 \in \mathbb{R}^n$. Their intersection $S_1 \cap S_2$ is also convex. This can be extended to the intersection of any number (finite or infinite) of sets, that is: the intersection of any number (finite or infinite) of convex sets is convex. In mathematical notation: if S_t is convex for all $t \in \mathcal{T}$, then $\cap_{t \in \mathcal{T}} S_t$ is convex. Intersection can provide an easy argument to show that a particular set is convex.

Example Consider the question whether a polyhedron $P = \{x \in \mathbb{R}^n \mid Ax \le b\}$ is convex. Since each inequality $a_j x \le b_j$, $j = 1, \ldots, m$ corresponds to a halfspace (which is convex), the set P corresponds to the intersection of m halfspaces, and must therefore be convex.

Example Consider the question whether the set of co-positive polynomials of degree n:

$$P_+^n \equiv \{(p_0, \ldots, p_n) \mid 0 \le p_0 + p_1 x + \ldots + p_n x^n \text{ for all } x \ge 0\},$$

is a convex set. The set P_+^n can be written as $\cap_{x \ge 0} P_x^n$, where

$$P_x^n \equiv \{(p_0, \ldots, p_n) \mid 0 \le p_0 + p_1 x + \ldots + p_n x^n\}.$$

Since, for each fixed x, the set P_x^n is a halfspace in $\mathbb{R}^{n+1}$, it follows that P_+^n is convex.

1.3.2 Affine functions

An affine function f has the form $f(x) = Ax + b$, where $A \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^m$. Given some set $S \in \mathbb{R}^n$, we denote by

$$f(S) = \{f(x) \mid x \in S\},$$

the *image* of S under f. And similarly, if $f(x) = Ax + b$, where $A \in \mathbb{R}^{n\times k}$ and $b \in \mathbb{R}^n$, we denote by

$$f^{-1}(S) = \{x \mid f(x) \in S\}$$

the *inverse image* of S under f.

Fact 1.3.1 *If f is affine, and if S is convex, then the image of S under f, as well as the inverse image of S under f, is convex.*

An example of this fact is *scaling*: if S is convex, and $\alpha \in \mathbb{R}$, then $\alpha S = \{\alpha x |\ x \in S\}$ is convex. Another example is translation: if S is convex, and $a \in \mathbb{R}^n$, then $S + a \equiv \{x + a \mid x \in S\}$ is convex.

We can use these facts to show that an ellipsoid is a convex set. Starting from the fact that the unit ball $B^n = \{x \in \mathbb{R}^n |\ ||x|| \le 1\}$ is convex, and recalling the definition of an ellipsoid, namely:

Definition 1.3.1 *Let Z be a non-singular $n \times n$ matrix; let $c \in \mathbb{R}^n$. Then $E(Z, c) := \{c + Zx \mid \|x\| \le 1\}$ is an* ellipsoid.

It follows that an ellipsoid is the image of an affine function applied to the convex ball, and hence, an ellipsoid is convex.

Lemma 1.3.1 *A set $E \subseteq \mathbb{R}^n$ is an ellipsoid if and only if*

$$E = \{y \in \mathbb{R}^n \mid (y - c)^T A^{-1}(y - c) \le 1\}$$

for some $c \in \mathbb{R}^n$ and some positive definite A. (Take $A = ZZ^T$)

Here is one way of defining the sum of two sets; this particular way is sometimes called the Minkowski-sum.

Definition 1.3.2 $S_1 + S_2 \equiv \{x + y|\ x \in S_1, y \in S_2\}$

Fact 1.3.2 *If S_1 and S_2 are convex, then $S_1 + S_2$ is convex.*

The Cartesian product of two sets is defined as follows.

Definition 1.3.3 $S_1 \times S_2 \equiv \{(x, y) \mid x \in S_1, y \in S_2\}$

Fact 1.3.3 *If S_1 and S_2 are convex, then $S_1 \times S_2$ is convex.*

Definition 1.3.4 *Let $a_1, \ldots, a_m \in \mathbb{R}^n$. The* convex hull *of $a_1, \ldots, a_m$ is*

$$conv\{a_1, \ldots, a_m\} \equiv \{\sum_{i=1}^{m} \lambda_i a_i \mid \sum_{i=1}^{m} \lambda_i = 1, \lambda_i \geq 0 \textit{ for all } i\}.$$

Clearly, for $m = 2$, the convex hull coincides with the set of convex combinations of a_1, a_2 as introduced in Definition 1.2.4. Let us now argue that $\text{conv}\{a_1, \ldots, a_m\}$ is a convex set. Choose, as affine function $f : \lambda \mapsto \sum_i \lambda_i a_i$, and choose as convex set: $C = \{\lambda \mid \sum_i \lambda_i = 1, \lambda_i \geq 0 \text{ for all } i\}$. Since the image of C under f is equal to $\text{conv}\{a_1, \ldots, a_m\}$, it follows that the convex hull of $a_1, \ldots, a_m$ is convex.

1.4 About extreme points of convex sets, and about separating sets

1.4.1 Systems having solutions ... or not

From linear algebra, we know that *either* the linear system $Ax = b$ has a solution *or* there exists a row vector y such that $yA = 0$ and $yb \neq 0$. And from Linear Programming we know that *either* $Ax = b, x \geq 0$ has a solution, *or* there exists a row vector y with $yA \leq 0$ and $yb > 0$ (Farkas' Lemma). Another version of Farkas' Lemma is: *either* $Ax \leq b$ has a solution, *or* there exists a row vector $y \geq 0$, such that $yA = 0$ and $yb < 0$. Evidently, the conditions $C1$ and $C2$ in *either* $C1$ *or* $C2$, cannot both be true, in the cases mentioned above. Such results can be phrased as so-called *separation* results, the subject of this section.

1.4.2 About extreme points

Let $S \in \mathbb{R}^n$ be a non-empty convex set.

Definition 1.4.1 *A point $x \in S$ is an* extreme point *of S when the equality $x = \lambda x_1 + (1 - \lambda)x_2$ with $x_1, x_2 \in S$ and $0 < \lambda < 1$ implies $x = x_1 = x_2$.*

Thus, an extreme point of a set cannot be written as a convex combination of two other points in the set.

Let us mention a number of examples. Consider the extreme points of a closed disk in $\mathbb{R}^2$. Each point on the boundary of the disk (and only each point on the boundary) satisfies the above definition. As another example, consider a convex polygon in $\mathbb{R}^2$. Observe that its corner points are its extreme points.

Lemma 1.4.1 *Let $P \subseteq \mathbb{R}^n$ be a polyhedron, i.e., P is the intersection of finitely many halfspaces. Then P has finitely many extreme points.*

Finally, consider the following *prism*: $\{x \in \mathbb{R}^3 : \; x_1 \geq 0, x_2 \geq 0, x_1 + x_2 \leq 1\}$. Question: Does the prism (a convex polytope) have extreme points?

Theorem 1.4.1 (Krein-Milman theorem) *A compact convex set S is the convex hull of its extreme points.*

1.4.3 About separating disjoint convex sets

It is a basic fact that when two convex sets in $\mathbb{R}^n$ are disjoint, that there exists a hyperplane separating the two sets. Various versions of this fact are described in the following theorems. We first consider the separation of a convex set C, and a point $x_0 \notin C$.

Theorem 1.4.2 *Let $C \subseteq \mathbb{R}^n$ be a closed, convex set. Let $x_0 \notin C$.*
Then there exists a nonzero vector $y \in \mathbb{R}^n$ and a scalar $z \in \mathbb{R}$ such that (i) $y^T x > z$ for all $x \in C$, and (ii) $y^T x_0 < z$.

Now we state the separation of two disjoint convex sets.

Theorem 1.4.3 *Let $C, D \subseteq \mathbb{R}^n$ be two closed, disjoint (i.e., $C \cap D = \emptyset$), convex sets with at least one of them bounded.*
Then there exist a nonzero vector $y \in \mathbb{R}^n$ and a scalar $z \in \mathbb{R}$ such that:

(i) $y^T x < z$ for all $x \in C$, and

(ii) $y^T x > z$ for all $x \in D$.

The hyperplane $H = \{x \mid y^T x = z\}$ is said to (strictly) *separate* C from D. We include a proof based on the argument given in Boyd and Vandenberghe [3].

Proof: We may assume that there exist points $c \in C$ and $d \in D$ that achieve the minimum distance between points in C and D. To see this, assume without loss of generality, that D is bounded. Then, for each x, $\text{dist}(x, D) = \min_{d\in D} ||x - d||$ is well-defined as D is closed. Pick any point $c' \in C$, and let $\rho = \text{dist}(c', D)$. Consider the set $D' = \{x \,|\text{dist}(x, D) \leq \rho\}$. This set is compact as well, and contains c'. Finally consider $C' = C \cap D'$, then this C' is closed, nonempty, bounded and convex. Therefore the set $\{(x, y) \mid x \in C', y \in D\}$ is closed, nonempty, and bounded as well, and hence contains a vector minimizing $||x - y||$. Let this be attained at $x = c, y = d$, with $||c - d|| \leq \rho$. As each vector in $C \setminus C'$ has distance more than ρ to D we have $c \in C$ and $d \in D$.

The idea is to construct a hyperplane that is perpendicular to the line segment between c and d, and that contains the midpoint of that line segment, namely $\frac{d+c}{2}$. This hyperplane is given by $\{x \mid (d - c)^T x - \frac{(d-c)^T(d+c)}{2} = 0\}$. We will show that this hyperplane is indeed a strictly separating hyperplane. To do so, we need to show that for each point $u \in D$, the expression $(d-c)^T u - \frac{(d-c)^T(d+c)}{2}$ is positive (a similar argument can be used to show that this expression is negative for each point $u \in C$). We argue by contradiction. Suppose there exists a point $u \in D$ such that

$$f(u) \equiv (d - c)^T u - \frac{(d - c)^T(d + c)}{2} \leq 0.$$

Using that $(d - c)^T(d - c) = ||d - c||_2^2$, we find that

$$f(u) = (d - c)^T(u - d + \frac{(d - c)}{2}) = (d - c)^T(u - d) + \frac{||d - c||_2^2}{2}.$$

Thus, as $||d-c|| > 0$, it must be the case that $(d-c)^T(u-d) < 0$. Now we can view the left-hand side of this expression as half the derivative of $||d + t(u - d) - c||_2^2$ at $t = 0$, which is apparently negative. Thus, "by taking a small step" from d in the direction of u, we get closer to c, i.e.:

$$||d + t(u - d) - c||_2 < ||d - c||_2.$$

Moreover, since $d \in D$, and $u \in D$, convexity implies that $d + t(u - d) \in D$. This however is impossible, since d is the point in D closest to C. □

Notice that separation of two convex, disjoint sets by a hyperplane, depends on the convexity of both sets, i.e., it should be obvious that there exist a pair of disjoint sets, one convex and one not convex that cannot be separated by a hyperplane.

Question: can you find two closed, convex disjoint sets that cannot be strictly separated? (Clearly, in the light of Theorem 1.4.3, both sets, if they exist, must be unbounded).

Theorem 1.4.4 *Let $C \subseteq \mathbb{R}^n$ be a convex set, and let x_0 lie on the boundary of C. Then there exist a nonzero vector $y \in \mathbb{R}^n$ and a scalar $z \in \mathbb{R}$ such that (i) $y^T x \leq z$ for all $x \in C$, and (ii) $y^T x_0 = z$.*

The hyperplane $H = \{x \mid y^T x = z\}$ is said to *support C at x_0*.

1.5 Convex functions

It is time to formally define what it means for a function to be convex. We start with (known) definitions of linear and affine functions.

Definition 1.5.1 *A function $f : \mathbb{R}^n \to \mathbb{R}^m$ is* linear *if $f(\alpha x + \beta y) = \alpha f(x) + \beta f(y)$ for all $x, y \in \mathbb{R}^n$ and $\alpha, \beta \in \mathbb{R}$. ($f(x) = Ax$)*

Definition 1.5.2 *A function $f : \mathbb{R}^n \to \mathbb{R}^m$ is* affine *if $f(\alpha x + \beta y) = \alpha f(x) + \beta f(y)$ for all $x, y \in \mathbb{R}^n$ and $\alpha, \beta \in \mathbb{R}$ with $\alpha + \beta = 1$. ($f(x) = d + Ax$)*

Here it is.

Definition 1.5.3 *A function $f : \mathbb{R}^n \to \mathbb{R} \cup \{\infty\}$ is* convex *if*

$$f(\alpha x + \beta y) \leq \alpha f(x) + \beta f(y)$$

for all $x, y \in \mathbb{R}^n$ and $\alpha, \beta \geq 0$ with $\alpha + \beta = 1$.

A function f is called *strictly convex* if the inequality in Definition 1.5.3 above is strict for $\alpha, \beta > 0$ and $x \neq y$. A function f is called *concave* if $-f$ is convex. Notice that a function that is convex as well as concave is affine. Indeed, if $f(\alpha x + \beta y) \leq \alpha f(x) + \beta f(y)$ and $-f(\alpha x + \beta y) \leq -\alpha f(x) - \beta f(y)$, it easily follows that $f(\alpha x + \beta y) = \alpha f(x) + \beta f(y)$ – at least for $\alpha, \beta \geq 0$; for a complete proof finish exercise 7. Conversely and more trivial, an affine function is both convex and concave. Here are some examples of convex functions.

- Any function that is a norm, is a convex function.
- If $C \subseteq \mathbb{R}^n$ is a convex set, then the function $f : \mathbb{R}^n \to \mathbb{R}$ defined by

  $$f(x) = \begin{cases} 0 & \text{if } x \in C \\ \infty & \text{otherwise} \end{cases}$$

  is convex.

Another definition of convexity of a function is based on the area (or set) "above" the function, called the epigraph. Notice that the word "epigraph" has also a meaning different from the one defined below, namely inscription.

Definition 1.5.4 *Let $f : \mathbb{R}^n \to \mathbb{R} \cup \{\infty\}$ be a function. Then the* epigraph *of f is*

$$epi(f) := \{(x,t) \mid x \in \mathbb{R}^n, t \in \mathbb{R}, f(x) \leq t\}.$$

Theorem 1.5.1 *f is a convex function $\Longleftrightarrow$ epi(f) is a convex set.*

Proof: $\Rightarrow$: Consider $(x_1,t_1) \in \text{epi}(f)$, $(x_2,t_2) \in \text{epi}(f)$, $\alpha \geq 0$, $\beta \geq 0$, $\alpha + \beta = 1$. Then $\alpha f(x_1) \leq \alpha t_1$, $\beta f(x_2) \leq \beta t_2$, as the $(x_i,t_i) \in \text{epi}(f)$, and $\alpha, \beta \geq 0$. Convexity of f yields $f(\alpha x_1 + \beta x_2) \leq \alpha f(x_1) + \beta f(x_2) \leq \alpha t_1 + \beta t_2$, hence $\alpha(x_1,t_1) + \beta(x_2,t_2) = (\alpha x_1 + \beta x_2, \alpha t_1 + \beta t_2) \in \text{epi}(f)$.

$\Leftarrow$: Consider the points $(x_1, f(x_1))$ and $(x_2, f(x_2))$. By definition, each of them is in epi(f). Since epi(f) is convex, it follows that $\alpha(x_1, f(x_1)) + \beta(x_2, f(x_2)) = (\alpha x_1 + \beta x_2, \alpha f(x_1) + \beta f(x_2))$ with $\alpha + \beta = 1, \alpha, \beta \geq 0$ is in epi(f). Thus, $f(\alpha x_1 + \beta x_2) \leq \alpha f(x_1) + \beta f(x_2)$, or in other words, f is convex. $\square$

Lemma 1.5.1 *Let $f : \mathbb{R}^n \to \mathbb{R} \cup \{\infty\}$ be* convex *and let $\gamma \in \mathbb{R}$.*
Then the sublevel set $\{x \in \mathbb{R}^n \mid f(x) \leq \gamma\}$ is a convex set.

1.5.1 The first order condition

Convex functions can also be characterized by considering their gradient. Indeed, when constructing a tangent line to any point on a convex function, the line will be "below" the function. We can use this to characterize convexity. Let us first recall the definition of a gradient. The *gradient* of a function $f : \mathbb{R}^n \to \mathbb{R}$ is the vector

$$\nabla f = (\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n})^T.$$

Theorem 1.5.2 *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a differentiable function. Then f is convex if and only if*

$$f(y) \geq f(x) + \nabla f(x)^T (y - x) \tag{1.1}$$

for all $x, y \in \mathbb{R}^n$.

Proof: $\Leftarrow$: Let $z = \alpha x + (1-\alpha)y$ with $0 \leq \alpha \leq 1$. Observe that $x = z + x - z = z + (1-\alpha)(x-y)$, and hence $x - z = (1-\alpha)(x-y)$. Let us now apply (1.1) to x and z:

$$f(x) \geq f(z) + \nabla f(z)^T (1-\alpha)(x-y), \tag{1.2}$$

and to y and z:

$$f(y) \geq f(z) + \nabla f(z)^T \alpha(y-x). \tag{1.3}$$

Multiplying (1.2) by α, multiplying (1.3) by $1-\alpha$, and summing the resulting expressions leads to:

$$\alpha f(x) + (1-\alpha)f(y) \geq f(z) = f(\alpha x + (1-\alpha)y),$$

showing that f is convex.

$\Rightarrow$: Now, we can assume convexity, that is:

$$f(\alpha y + (1-\alpha)x) \leq \alpha f(y) + (1-\alpha)f(x) \text{ for all } x, y, 0 \leq \alpha \leq 1.$$

Thus, we can derive:

$$\begin{aligned} f(\alpha y + (1-\alpha)x) \leq \alpha f(y) + (1-\alpha)f(x) \Rightarrow \\ f(x + \alpha(y-x)) - f(x) \leq \alpha(f(y) - f(x)) \Rightarrow \\ \frac{f(x+\alpha(y-x)) - f(x)}{\alpha} \leq f(y) - f(x) \Rightarrow \\ f(y) \geq f(x) + \frac{f(x+\alpha(y-x) - f(x)}{\alpha}. \end{aligned}$$

We set $g(\alpha) \equiv f(x + \alpha(y-x))$. Using this, we find:

$$f(y) \geq f(x) + \frac{g(\alpha) - g(0)}{\alpha}.$$

We now take the limit when $\alpha \downarrow 0$, to arrive at:

$$f(y) \geq f(x) + g'(0). \tag{1.4}$$

Observe that $g'(\alpha) = (\nabla f(x + \alpha(y-x))^T)(y-x)$. Substituting this in (1.4) leads to the desired result.

$\square$

Inequality (1.1) is referred to as the first-order condition - notice that the theorem only applies to functions that are differentiable.

It will come as no surprise that the gradient of a convex function f at a particular point x^* being zero is necessary and sufficient for that point being locally, and hence globally (recall Theorem 1.1.1), optimal. Indeed, for convex f:

$$f(x^*) = \min\{f(y) \mid y \in \mathbb{R}^n\} \iff \nabla f(x^*) = 0.$$

Observe that inequality (1.1) can be used to show that if the gradient of a convex function equals 0 in some point x^*, this point is a minimum. Indeed, assuming convexity of f, choose in (1.1) $x = x^*$, and it follows that, since $\nabla f(x^*) = 0$, $f(y) \geq f(x^*)$, implying that x^* is an optimum. The other direction (called Fermat's theorem) will be considered in Chapter 2.

The first-order condition (1.1) can be useful to prove convexity. Consider, as an example, the following statement.

Fact 1.5.1 *For a given $(n \times n)$ symmetric matrix A, a vector $b \in \mathbb{R}^n$, and scalar $c \in \mathbb{R}$, the quadratic function $f(x) = x^T Ax + bx + c$ is convex if and only if A is positive semi-definite.*

Proof: The first-order condition says: $f(y) \geq f(x) + \nabla f(x)^T(y - x)$. Since $\nabla f(x)^T = 2x^T A + b$, plugging this equality into the first-order condition leads to:

$$y^T Ay + by + c \geq x^T Ax + bx + c + (2x^T A + b)(y - x),$$

which, after simplifying, becomes $(y - x)^T A(y - x) \geq 0$. Since the latter inequality is true for all y, x when A is positive semi-definite, the result follows. □

Question: where (if at all) do we use symmetry of A?

A well-known special case arises for $n = 1$, the univariate quadratic function. For $a, b, c \in \mathbb{R}$, the univariate quadratic function $f(x) = ax^2 + bx + c$ is convex if and only if $a \geq 0$.

1.5.2 The second order condition

Informally speaking, a convex function "curves upwards" anywhere on its domain. This can be made formal by considering the derivative of the derivative, i.e., the second derivative, also known as the Hessian. Consider first the definition of the Hessian.

Definition 1.5.5 *The* Hessian *of a twice differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ is an $(n \times n)$ symmetric matrix $\nabla^2 f$ such that*

$$(\nabla^2 f)_{i,j} = \frac{\partial^2 f}{\partial x_i \partial x_j}.$$

Example 1.5.1 *Consider the function $f : \mathbb{R}^n \to \mathbb{R}$ defined by $f(x) = \frac{1}{2}x^T Q x + c^T x$, where Q is an $(n \times n)$ symmetric matrix, and c an $(n \times 1)$ vector. Then the Hessian of $f(x)$, denoted by $\nabla^2 f(x)$, equals Q.*

Recall: a function is *analytic* if it has a Taylor series for each point x in its domain that converges to the function in an open neighborhood of x.
For univariate analytic functions $f : \mathbb{R} \to \mathbb{R}$ we have:

$$f(x+h) \;=\; f(x) + f'(x)h + \frac{1}{2}f''(x+\alpha h)h^2,$$

for some $\alpha \in [0,1]$.
When generalizing this to the multivariate case $f : \mathbb{R}^n \to \mathbb{R}$, we get:

$$f(x+h) \;=\; f(x) + \nabla f(x)^T h + \frac{1}{2}h^T \nabla^2 f(x+\alpha h)h$$

for some $\alpha \in [0,1]$.

The Hessian can be used to provide yet another characterization of convex functions, provided they are twice differentiable.

Theorem 1.5.3 *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a twice differentiable function. Then f is convex if and only if the Hessian is positive semi-definite, i.e., if and only if $\nabla^2 f(x) \succeq 0$ for all $x \in \mathbb{R}^n$.*

The inequality $\nabla^2 f(x) \succeq 0$ is referred to as the second-order condition.

Lemma 1.5.2 *A function $f : \mathbb{R}^n \to \mathbb{R}$ is convex, if and only if $g(\lambda) := f(x + \lambda d)$ is convex for all $x, d \in \mathbb{R}^n$.*

Many well-known functions are convex (or concave). Here are some examples:

- $x \mapsto e^{ax}$ is convex on $\mathbb{R}$, for all $a \in \mathbb{R}$,
- $x \mapsto x^a$ is convex on $\mathbb{R}_+$, for all $a \leq 0$ and all $a \geq 1$ (concave otherwise),
- $x \mapsto \log(x)$ is concave on $\mathbb{R}_+$,
- $x \mapsto x\log(x)$ is convex on $\mathbb{R}_+$,

- $(x, y) \mapsto \frac{x^2}{y}$ is convex on $\{(x, y) \mid y > 0\}$,
- $x \mapsto \log(\sum_i e^{x_i})$ is convex on $\mathbb{R}^n$,
- $x \mapsto (\prod_i x_i)^{1/n}$ is concave on $\mathbb{R}^n_+$.

To prove the status of these examples, it suffices, for the univariate cases, to compute the second derivative. For the remaining three multivariate cases, we need to derive the Hessian and argue its positive semi-definiteness. More concrete:

- $f : (x, y) \mapsto \frac{x^2}{y}$ has gradient $\nabla f(x, y)^T = [\frac{2x}{y} \;\; \frac{-x^2}{y^2}]$, hence the Hessian $\nabla^2 f(x, y) = \begin{bmatrix} \frac{2}{y} & \frac{-2x}{y^2} \\ \frac{-2x}{y^2} & \frac{2x^2}{y^3} \end{bmatrix} = \frac{2}{y^3} \begin{bmatrix} y^2 & -xy \\ -xy & x^2 \end{bmatrix} = \frac{2}{y^3} \begin{bmatrix} y \\ -x \end{bmatrix} \begin{bmatrix} y & -x \end{bmatrix} \succeq 0$, for $y > 0$.
- $f : x \mapsto \log(\sum_i e^{x_i})$ has gradient ∇f with $\nabla f(x)_i = e^{x_i} / \sum_k e^{x_k}$, and so the Hessian $\nabla^2 f$ has $\nabla^2 f(x)_{i,j} = -e^{x_i} e^{x_j} / (\sum_k e^{x_k})^2$, for $i \neq j$ whereas $\nabla^2 f(x)_{i,i} = e^{x_i} \times (\sum_k e^{x_k} - e^{x_i}) / (\sum_k e^{x_k})^2$. Using the shorthand $z_i = e^{x_i}$ $(1 \leq i \leq n)$, we have that $\nabla^2 f(x) = (1^T z)^{-2}((1^T z)\mathrm{diag}(z) - zz^T)$. To show that $\nabla^2 f(x) \succeq 0$, we argue that $u^T \nabla^2 f(x) u \geq 0$, for all $u \in \mathbb{R}^n$. Indeed, we find that

  $$u^T \nabla^2 f(x) u = (1^T z)^{-2}((\sum_k z_k)(\sum_k z_k u_k^2) - (\sum_k z_k u_k)^2) \geq 0,$$

  which follows from the Cauchy-Schwarz inequality $(a^T b)^2 \leq a^T a \cdot b^T b$, (see Section 1.7.2), applied to $a_i = \sqrt{z}_i$ and $b_i = u_i \sqrt{z_i}$ $(1 \leq i \leq n)$.
- $f : x \mapsto (\prod_i x_i)^{1/n}$ has gradient ∇f with $\nabla f(x)_i = \frac{1}{n} f(x)/x_i$, so for the Hessian $\nabla^2 f$ we have $\nabla^2 f(x)_{i,j} = \frac{1}{n^2} f(x)/(x_i x_j)$, for $i \neq j$, while $\nabla^2 f(x)_{i,i} = \frac{1}{n^2} f(x)/(x_i x_i) - \frac{1}{n} f(x)/(x_i x_i)$. To show that f is concave we have to show that $-\nabla^2 f \succeq 0$. Using the shorthand $q_i = 1/x_i$, we have $\nabla^2 f(x) = -\frac{f(x)}{n^2}(n\,\mathrm{diag}(1/x_1^2, \ldots, 1/x_n^2) - qq^T)$ and so, for any $u \in \mathbb{R}^n$:

  $$-u^T \nabla^2 f(x) u = \frac{f(x)}{n^2}(n \sum_i (u_i/x_i)^2 - (\sum_i u_i/x_i)^2) \geq 0,$$

  by positiveness of f and application of Cauchy-Schwarz with $a_i = 1$ and $b_i = u_i/x_i$ $(1 \leq i \leq n)$.

Finally, let us consider some operations on functions that preserve convexity.

Lemma 1.5.3 *If $f, g : \mathbb{R}^n \to \mathbb{R}$ are convex functions, then so is*

$$\alpha f + \beta g$$

for all $\alpha, \beta \geq 0$

Lemma 1.5.4 *If $f, g : \mathbb{R}^n \to \mathbb{R}$ are convex functions, then so is* $\max\{f, g\}$.

Obviously, the above statement ceases to be true when "max " is replaced by "min".

Lemma 1.5.5 *If $f : \mathbb{R}^n \to \mathbb{R}$ is convex and $g : \mathbb{R}^m \to \mathbb{R}^n$ is affine, then $f \circ g$ is convex.*

1.6 Recognizing PSD matrices

From the previous paragraph, it has become clear that convexity of a function (that is twice differentiable), is intimately related to positive semi-definiteness of matrices. Let us recall some practical aspects of matrices that are positive semi-definite.

Definition 1.6.1 *A real matrix A is* symmetric *if $A^T = A$. The set of symmetric $n \times n$ matrices is denoted by S^n.*

Recall:

Theorem 1.6.1 *For any matrix $A \in S^n$, there exists an $n \times n$ matrix F and a diagonal matrix Λ so that $F^T F = I$ and $F^T A F = \Lambda$.*

Let $\lambda_1, \dots, \lambda_n$ be the diagonal entries of Λ, and let $f_1, \dots, f_n$ be the columns of F. Then

- $f_1, \dots, f_n$ is an orthonormal basis of $\mathbb{R}^n$.
- $Af_i = \lambda_i f_i$ for all i.
- $A = \lambda_1 f_1 f_1^T + \cdots + \lambda_n f_n f_n^T$.

A function $f : \mathbb{R}^n \to \mathbb{R}$ is *positive semi-definite* if $f(x) \geq 0$ for all $x \in \mathbb{R}^n$.

If $A \in S^n$, then the function $f(x) = x^T A x = \sum_i \sum_j A_{ij} x_i x_j$ is a *homogeneous quadratic function* ($f : \mathbb{R}^n \to \mathbb{R}$).

Definition 1.6.2 *Let $A \in S^n$. Then A is* positive semi-definite (PSD) *if*

$$x^T A x \geq 0 \text{ for all } x \in \mathbb{R}^n.$$

The set of positive semi-definite matrices is denoted by S^n_+.

An $A \in S^n$ is *positive definite (PD)* if A is PSD and non-singular. The set of positive definite matrices is denoted by S^n_{++}. We write $A \succeq 0$ to denote that A is PSD, and $A \succ 0$ if A is PD.

Most results on PSD matrices in this course are derived from this one:

Theorem 1.6.2 *Let $A \in S^n$. The following three statements are equivalent:*

1. *A is positive semi-definite.*
2. *each eigenvalue of A is ≥ 0.*
3. *there is some real matrix Z such that $A = Z^T Z$.*

In particular,

- A is PSD $\Longrightarrow \det(A) \geq 0$
- A is PSD $\Longrightarrow$ the diagonal entries of A are ≥ 0
- if A is diagonal, then: A is PSD $\Longleftrightarrow$ diagonal entries of A are ≥ 0
- if $A = \begin{bmatrix} B & 0 \\ 0 & C \end{bmatrix}$, then: A is PSD $\Longleftrightarrow$ both B and C are PSD.

Matrices $A, B \in S^n$ are *congruent*, if $B = U^T A U$ for some non-singular U.

Lemma 1.6.1 *Let $A, B \in S^n$ be congruent. Then $A \succeq 0 \Longleftrightarrow B \succeq 0$.*

Applying one or more of the following *symmetric matrix operations* to A yields a congruent matrix:

- scaling the i-th row and the i-the column by a $\lambda \neq 0$,
- interchanging the i-th row with the j-th row and the i-th column with the j-th column, and
- adding $\lambda \times$ the i-th row to the j-th row and adding $\lambda \times$ the i-th column to the j-th column.

By these operations a matrix A can be transformed to a congruent diagonal matrix D. Then, $A \succeq 0 \Leftrightarrow D \succeq 0 \Leftrightarrow D \geq 0$.

Example: is $\begin{pmatrix} 1 & 0 & -1 & 1 \\ 0 & 2 & 0 & 4 \\ -1 & 0 & 3 & 0 \\ 1 & 4 & 0 & 4 \end{pmatrix}$ PSD?

$$\begin{pmatrix} 1 & 0 & -1 & 1 \\ 0 & 2 & 0 & 4 \\ -1 & 0 & 3 & 0 \\ 1 & 4 & 0 & 4 \end{pmatrix} = \begin{pmatrix} \boxed{1} & 0 & -1 & 1 \\ 0 & 2 & 0 & 4 \\ -1 & 0 & 3 & 0 \\ 1 & 4 & 0 & 4 \end{pmatrix} \approx$$

$$\to \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 4 \\ -1 & 0 & 2 & 1 \\ 1 & 4 & 1 & 3 \end{pmatrix} \to \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \boxed{2} & 0 & 4 \\ 0 & 0 & 2 & 1 \\ 0 & 4 & 1 & 3 \end{pmatrix} \approx$$

$$\to \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 4 & 1 & -5 \end{pmatrix} \to \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & \boxed{2} & 1 \\ 0 & 0 & 1 & -5 \end{pmatrix} \approx$$

$$\to \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 1 & -5\frac{1}{2} \end{pmatrix} \to \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & -5\frac{1}{2} \end{pmatrix} \text{ NOT PSD!!}$$

And now:

$$\begin{pmatrix} 1 & 0 & -1 & 1 \\ 0 & 2 & 0 & 2 \\ -1 & 0 & 3 & 0 \\ 1 & 2 & 0 & 4 \end{pmatrix} = \begin{pmatrix} \boxed{1} & 0 & -1 & 1 \\ 0 & 2 & 0 & 2 \\ -1 & 0 & 3 & 0 \\ 1 & 2 & 0 & 4 \end{pmatrix} \approx$$

$$\to \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 2 \\ -1 & 0 & 2 & 1 \\ 1 & 2 & 1 & 3 \end{pmatrix} \to \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \boxed{2} & 0 & 2 \\ 0 & 0 & 2 & 1 \\ 0 & 2 & 1 & 3 \end{pmatrix} \approx$$

$$\rightarrow \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 2 & 1 & 1 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & \boxed{2} & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \approx$$

$$\rightarrow \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 1 & \frac{1}{2} \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & \frac{1}{2} \end{pmatrix} \text{YES, PSD!!}$$

1.7 Inequalities

The inequality defining convexity (Definition 1.5.3) is an inequality from which many other, well-known inequalities can be deduced. Consider for instance the well-known inequality:

$$e^z \geq 1 + z,$$

which is true for any $z \in \mathbb{R}$. Accepting the convexity of the function $f(z) = e^z$, and next applying the first-order condition $f(y) \geq f(x) + \nabla f(x)^T (y - x)$ gives: $e^y \geq e^x + e^x(y - x)$. Next by choosing $x = 0, y = z$, and plugging in, we arrive at the inequality.

1.7.1 Jensen's inequality

Another well-known inequality is Jensen's inequality, which can manifest itself in many different ways. (Johan Jensen (1859–1925): Danish mathematician and engineer)

Theorem 1.7.1 (Jensen, 1906) *For a convex function $f : \mathbb{R} \to \mathbb{R}$ and real numbers $x_1, \ldots, x_n$, we have*

$$\frac{1}{n} \cdot \sum_{i=1}^{n} f(x_i) \;\geq\; f\left(\frac{1}{n} \sum_{i=1}^{n} x_i\right). \tag{1.5}$$

If f is concave, then the inequality holds with $\leq$ instead of $\geq$. As a concrete outcome of the resulting inequality, observe that Jensen's inequality in this form can be used to show that, for a given set of numbers, the corresponding geometric mean is bounded from above by the corresponding arithmetic mean, as expressed in the following fact.

Fact 1.7.1 *For positive real numbers $x_1, \ldots, x_n$, we have*

$$(x_1 x_2 \cdots x_n)^{1/n} \;\leq\; \frac{x_1 + x_2 + \cdots + x_n}{n}.$$

Indeed, this fact follows when taking, in Theorem 1.7.1, $f(x_i) = \log\, x_i$ (recall that $\log\, x$ is a concave function). This leads to:

$$\frac{1}{n} \cdot \sum_{i=1}^{n} \log\, x_i \leq \log \frac{\sum_i x_i}{n}. \qquad (1.6)$$

Consider now the left-hand side of this inequality, and observe the following series of equalities:

$$\frac{1}{n} \cdot \sum_{i=1}^{n} \log\, x_i = \sum_{i=1}^{n} \frac{\log\, x_i}{n} = \sum_{i=1}^{n} \log\, x_i^{\frac{1}{n}} = \log\, \Pi_i x_i^{\frac{1}{n}}. \qquad (1.7)$$

Jointly, (1.6) and (1.7) imply that $\log\, \Pi_i x_i^{\frac{1}{n}} \leq \log \frac{\sum_i x_i}{n}$. From this, plus monotonic increase of e^x we conclude that the fact follows. Actually, Theorem 1.7.1 can further be generalized as follows.

Theorem 1.7.2 *For a convex function $f : \mathbb{R} \to \mathbb{R}$ and real numbers $x_1, \ldots, x_n$, and positive real numbers $a_1, \ldots, a_n$, we have*

$$\frac{\sum_{i=1}^{n} a_i f(x_i)}{\sum_{i=1}^{n} a_i} \;\geq\; f\left(\frac{\sum_{i=1}^{n} a_i x_i}{\sum_{i=1}^{n} a_i}\right).$$

Of course, this is indeed a generalization: by taking $a_i = 1$ for all i, Theorem 1.7.1 appears. As an application of this theorem, consider a discrete probability distribution defined by numbers p_i, where $p_1, p_2, \ldots, p_n \geq 0$ and $\sum_{i=1}^{n} p_i = 1$. The *entropy* of a probability distribution is defined as

$$H(p) \;=\; -\sum_{i=1}^{n} p_i \log p_i.$$

We now show how to derive an upper bound on the entropy by using Theorem 1.7.2. Indeed, by choosing $a_i = p_i$, $x_i = \frac{1}{p_i}$ and by letting $f(x) = \log\, x$, we can argue as follows:

$$H(p) = -\sum_{i=1}^{n} p_i \log p_i = \frac{\sum_{i=1}^{n} p_i \log\, \frac{1}{p_i}}{\sum_{i=1}^{n} p_i} \leq \log\, \frac{\sum_{i=1}^{n} p_i \times \frac{1}{p_i}}{\sum_{i=1}^{n} p_i} = \log\, n,$$

where the inequality follows from Theorem 1.7.2.

1.7.2 The master inequality

Let us consider a final, general inequality, that we refer to as the *master* inequality.

Let $g : \mathbb{R} \to \mathbb{R}$ be a strictly concave function, and let $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ be the function that is defined by

$$f(x, y) \;=\; y \cdot g\left(\frac{x}{y}\right).$$

Then all real numbers $x_1, \ldots, x_n$ and all positive real numbers $y_1, \ldots, y_n$ satisfy the inequality

$$\sum_{i=1}^{n} f(x_i, y_i) \;\le\; f\left(\sum_{i=1}^{n} x_i \,, \sum_{i=1}^{n} y_i\right)$$

Equality holds if and only if the two sequences x_i and y_i are proportional (that is, if there exists a real number t such that $x_i/y_i = t$ for all i).

One way of proving the master inequality is using induction on n. Clearly, the case $n = 1$ holds (with equality), and the case $n = 2$ holds with $\alpha = y_1/(y_1 + y_2)$ and $\beta = y_2/(y_1 + y_2)$:

$$\begin{aligned}
&f(x_1, y_1) + f(x_2, y_2) \;= \\
&\quad= \; y_1 \cdot g\left(\frac{x_1}{y_1}\right) + y_2 \cdot g\left(\frac{x_2}{y_2}\right) \\
&\quad= \; (y_1 + y_2)\left\{\frac{y_1}{y_1 + y_2} \cdot g\left(\frac{x_1}{y_1}\right) + \frac{y_2}{y_1 + y_2} \cdot g\left(\frac{x_2}{y_2}\right)\right\} \\
&\quad\le \; (y_1 + y_2) \cdot g\left(\frac{x_1 + x_2}{y_1 + y_2}\right) \;=\; f\left(x_1 + x_2,\, y_1 + y_2\right).
\end{aligned}$$

As g is strictly concave, equality holds if and only if $x_1/y_1 = x_2/y_2$. Finally, the inductive step for $n \ge 3$ also follows from the inequality.

From this master inequality, we can derive the validity of the well-known Cauchy-Schwarz inequality. For real numbers $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$, we have

$$\left(a_1^2 + a_2^2 + \cdots + a_n^2\right)\left(b_1^2 + b_2^2 + \cdots + b_n^2\right) \;\ge\; \left(a_1 b_1 + a_2 b_2 + \cdots + a_n b_n\right)^2.$$

The idea is to observe that $g(x) = \sqrt{x}$ is a strictly concave function, and next set $x_i = a_i^2$ and $y_i = b_i^2$.

1.8 Exercises

Exercise 1

Are the following sets convex?

(a) $\{(x, y) \in \mathbb{R}^2 \mid x + 2y \le 10,\; 5x + y \le 4\}$

(b) $\{(x,y) \in \mathbb{R}^2 \mid (x-1)^2 + y^2 \leq 4 \text{ or } (x+1)^2 + y^2 \leq 4\}$

(c) $\{x \in \mathbb{R} \mid \cos(x) \leq 0\}$

(d) $\{x \in \mathbb{R}^3 \mid \|x\| \geq 1\}$

(e) $\{\begin{bmatrix} 3x+y \\ -y-2 \end{bmatrix} \mid x \geq 0,\ y \leq 0,\ x^2+y^2 \leq 1\}$

(f) $\{x \in \mathbb{R} \mid x^3 - 2x^2 + x \geq 0\}$

(g) $\{x \in \mathbb{R}^3 \mid x_1 + x_2 e^t + x_3 e^{2t} \geq 2 \text{ for all } t \leq 3\}$

(h) The set of real 2×2 matrices A with $\det(A) \geq 1$

Exercise 2

A cake is to be divided among $n \geq 2$ children so that the ith child receives a fraction x_i of the cake. The vector $x = (x_1, \ldots, x_n)$ is called an *allocation*. An allocation is *feasible*, if it satisfies the following three properties:

(i) Every child must receive some non-zero share of the cake.

(ii) The entire cake must be allocated to the children.

(iii) The first child ($i = 1$) must be allocated a share that is at least twice as big as the share of any other child.

Show that the set S of all feasible allocations is convex.

Exercise 3

Are the following quadratic functions convex?

(a) $f(x) = x_1^2 + x_2^2 + 2x_1x_2 + 5x_1 - x_2 + 1$ on $\mathbb{R}^2$

(b) $f(x) = x_1^2 + x_2^2 + x_3^2 - 2x_1x_2 - 2x_1x_3 - 2x_2x_3$ on $\mathbb{R}^3$

Exercise 4

Are the following functions convex on the given domain? Are they concave?

(a) $f(x) = \arctan(x)$ on $\mathbb{R}^+$

(b) $f(x,y,z) = (y+2z)^2/(x-3y)$ on $\{(x,y,z) \in \mathbb{R}^3 \mid x - 3y > 0\}$

(c) $f(x,y,z) = x^2 + y^2 + 5z^2 - xy - xz - 3yz$ on $\mathbb{R}^3$

(d) $f(x) = \max\{\|x\|, \|x-a\|\}$ on $\mathbb{R}^n$, where $a \in \mathbb{R}^n$

(e) $f(x,y) = -\log(\exp(x) + \exp(y))$ on $\mathbb{R}^2$

(f) $f(u,x,y,z) = u\log u + x\log x + y\log y + z\log z$ on $\mathbb{R}^4_+$

Exercise 5

Let $f : \mathbb{R} \to \mathbb{R}$ be a convex function, and let function $g : \mathbb{R} \to \mathbb{R}$ be defined by $g(x) = (f(x) - 2)^2$.

(a) Give an example of f, so that g is not convex.

(b) Prove: If $f(x) \geq 2$ for all $x \in \mathbb{R}$, then g is convex.

Exercise 6

Prove or disprove:

(a) The square of a convex positive function is convex.

(b) The square of a concave positive function is concave.

(c) The reciprocal of a positive concave function is convex.

Exercise 7

Prove that a function $f : \mathbb{R}^n \to \mathbb{R}$ is affine (that is, $f : x \mapsto cx + d$ for some $c \in \mathbb{R}^n$ and $d \in \mathbb{R}$), if and only if f is both convex and concave.

Exercise 8

Prove that the function $f : \mathbb{R}^n \to \mathbb{R}$ with $f(x) = ||x||^2$ is strictly convex.

Exercise 9

Prove that a strictly convex function $f : \mathbb{R}^n \to \mathbb{R}$ has at most one minimizer.

Exercise 10

Consider the quadratic function $f(x) = \frac{1}{2}x^T Ax + cx$, where A is a symmetric $n \times n$ matrix. Assume that $f : \mathbb{R}^n \to \mathbb{R}$ is bounded from below.

(a) Show that A is positive semi-definite.

(b) Show that f attains its minimum.

Exercise 11

Determine the convex hull of the set $S_1 \cup S_2$ where $S_1 = \{(0,0,0)\}$ and $S_2 = \{x \in \mathbb{R}^3 \mid x_1^2 + x_2^2 \leq 1;\ x_3 = 1\}$.

Exercise 12

Let $f : \mathbb{R} \to \mathbb{R}$ be a convex function. Prove that for $x < y < z$, we have

$$\frac{f(y) - f(x)}{y - x} \leq \frac{f(z) - f(x)}{z - x} \leq \frac{f(z) - f(y)}{z - y}$$

Exercise 13

Prove for all positive real numbers x and y that

$$\frac{x}{3} + \frac{2y}{3} \leq \sqrt{\ln\left(\frac{1}{3}e^{x^2} + \frac{2}{3}e^{y^2}\right)}.$$

(Hint: Use the convexity of an appropriately chosen function.)

Exercise 14

Prove for all real numbers a, b, c, d that

$$\left(\frac{a}{2} + \frac{b}{3} + \frac{c}{12} + \frac{d}{12}\right)^4 \leq \frac{1}{2}a^4 + \frac{1}{3}b^4 + \frac{1}{12}c^4 + \frac{1}{12}d^4.$$

Exercise 15

Use Jensen's inequality to determine the maximum value of $\sin\alpha + \sin\beta + \sin\gamma$ where α, β, γ are the angles of a triangle.

Exercise 16

Use the convexity of $f(z) = z + \frac{1}{z}$ for positive real z to determine the minimum value of

$$g(x, y) = 2x^2 + 3y^2 + \frac{1}{2x^2 + 3y^2}$$

where x, y are positive real numbers.

Exercise 17

Prove Young's inequality, which states for real numbers $p, q > 1$ with $\frac{1}{p} + \frac{1}{q} = 1$ and for positive real numbers x and y that

$$xy \leq \frac{x^p}{p} + \frac{y^q}{q}.$$

(Hint: Use the arithmetic-geometric mean inequality.) When does equality hold?

Exercise 18

Show that the master inequality implies

(a) the concave version of Jensen's inequality,

(b) the Hölder inequality, which states for two real numbers $p, q > 1$ with $\frac{1}{p} + \frac{1}{q} = 1$ and for positive real numbers $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ that

$$\sum_{i=1}^{n} a_i b_i \leq \left(\sum_{i=1}^{n} {a_i}^p\right)^{1/p} \left(\sum_{i=1}^{n} {b_i}^q\right)^{1/q},$$

(c) the Minkowski inequality, which states for a real parameter $p > 1$ and for positive real numbers $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ that

$$\left(\sum_{i=1}^{n} (a_i + b_i)^p\right)^{1/p} \quad \leq \quad \left(\sum_{i=1}^{n} {a_i}^p\right)^{1/p} \quad + \quad \left(\sum_{i=1}^{n} {b_i}^p\right)^{1/p},$$

(d) the Milne inequality, which states for positive real numbers $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$

$$\left(\sum_{i=1}^{n} (a_i + b_i)\right) \left(\sum_{i=1}^{n} \frac{a_i b_i}{a_i + b_i}\right) \;\leq\; \left(\sum_{i=1}^{n} a_i\right) \left(\sum_{i=1}^{n} b_i\right).$$

Chapter 2

Optimality conditions

Finding necessary and sufficient conditions for optimality is the holy grail in optimization. In general, such conditions can be hard to find. However, the story in this chapter describes that, for convex optimization problems, the so-called Karush-Kuhn-Tucker conditions are necessary and (often) sufficient for optimality. And even for problems that are not convex, a local optimum will (usually) satisfy the KKT conditions. These facts have, of course, implications for our ability to actually solve (convex) optimization problems. Let's find out how this works!

2.1 Reviewing unconstrained optimization

Definition 2.1.1 *In a* continuous optimization problem*, we want to solve*

$$\begin{array}{ll} \textit{minimize} & f(x) \\ \textit{subject to} & x \in U \end{array}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ *is the objective function and where* $U \subseteq \mathbb{R}^n$ *is the feasible region.*

Of course, the choice for minimizing is an arbitrary one, since "minimize $-f(x)$" is equivalent to "maximize $f(x)$". Further, it is clear that by taking $U = \mathbb{R}^n$ an *unconstrained* optimization problem arises.

There are some nice properties arising from functions f that are continuous, and feasible regions U being compact.

Theorem 2.1.1 *(Weierstrass) If a function f is* continuous *and if the feasible region U is* compact*, then f attains its minimum (and its maximum) in U.*

Theorem 2.1.2 *(Weierstrass) If a* continuous *function $f : \mathbb{R}^n \to \mathbb{R}$ goes to $+\infty$ whenever $\|x\| \to \infty$, then f attains its minimum.*

Recall from the univariate case ($f : \mathbb{R} \to \mathbb{R}$) that:

$$x^* \text{ minimizes } f \implies f'(x^*) = 0.$$

This statement is often called "Fermat's theorem". Let us sketch an argument for this. It is a fact that when x^* locally minimizes a function f that is differentiable in x^*, there exists a $\delta > 0$ such that for all $x \in [x^* - \delta, x^* + \delta]$, we have $f(x) \geq f(x^*)$. Thus, for any h with $0 < h < \delta$, we have $f(x^* + h) - f(x^*) \geq 0$, or $\frac{f(x^*+h)-f(x^*)}{h} \geq 0$. By definition of the derivative, this means $f'(x^*) \geq 0$. It is also true that for any h with $-\delta < h < 0$, we have $f(x^* + h) - f(x^*) \geq 0$, or $\frac{f(x^*+h)-f(x^*)}{h} \leq 0$, leading to $f'(x^*) \leq 0$. It follows that $f'(x^*) = 0$.

Let us now consider a generalization of Fermat's theorem to higher dimensions.

Definition 2.1.2 *Let $f : \mathbb{R}^n \to \mathbb{R}$ be differentiable.*
A point x^* with $\nabla f(x^*) = 0$ is called a stationary *point.*

Fact 2.1.1 *Necessary optimality condition (first order)*
If $f : \mathbb{R}^n \to \mathbb{R}$ is differentiable and if $x^* \in \mathbb{R}$ is a local minimum, then x^* is a stationary point.

We give the following proof which is based on the argument above.

Proof: Let x^* be a local minimum. Consider the one-dimensional function $g(t) = f(x^* + te_i)$ for some $i \in \{1, \dots, n\}$ (where e_i denotes a unit vector). Clearly, at $t = 0$, our function g is differentiable, and in fact, we have $g'(0) = \lim_{h\to 0} \frac{g(h)-g(0)}{h} = \lim_{h\to 0} \frac{f(x^*+he_i)-f(x^*)}{h} = \frac{\partial f(x^*)}{\partial x_i}$. Now, since x^* is a local minimum of f it must be true that $t = 0$ is a local minimum of g, which, by Fermat's theorem implies $g'(0) = 0$. Hence $\frac{\partial f(x^*)}{\partial x_i} = 0$ for each $i \in \{1, \dots, n\}$. This means $\nabla f(x^*) = 0$.

□

It is important to realize that the stationarity of a point x^* is a necessary, but not a sufficient condition for optimality. Take for instance $f(x) = x^3$, where the point $x^* = 0$ is a stationary point not corresponding to a local minimum.

Theorem 2.1.3 *Necessary optimality condition (second order)*
If f is twice continuously differentiable and if x^* is a local minimum, then

(i) x^* is a stationary point, and

(ii) the Hessian $\nabla^2 f(x^*)$ is positive semi-definite.

Again, this condition is necessary, but not sufficient: the same example as above ($f(x) = x^3$) applies. In the following theorem, we formulate a sufficient condition:

Theorem 2.1.4 *Sufficient optimality condition (second order)*
If f is twice continuously differentiable and if

(i) x^* is a stationary point, and

(ii) the Hessian $\nabla^2 f(x^*)$ is positive definite,

then x^* is a local minimum

And although stationarity of a point x^*, together with the Hessian being positive definite is sufficient for optimality, it is not a necessary condition. Indeed, the point $x = 0$ is a local minimum of $f(x) = x^4$, yet its second derivative in $x = 0$ is not positive.

By now, it should be clear that stationary points are interesting. Each stationary point of a differentiable function is either a local minimum, or a local maximum, or a so-called *saddle point.* This can be made more precise by considering the second order Taylor approximation of a twice differentiable function f in a given point x^*:

$$f(x^*) + \nabla f(x^*)^T(x - x^*) + \frac{1}{2}(x - x^*)^T \nabla^2 f(x^*)(x - x^*).$$

Observe that in a stationary point x^* (so $\nabla f(x^*)^T = 0$), the properties of the Hessian determine the behavior of f. Indeed, if the Hessian is not positive semi-definite, the point x^* is not a local minimum; and similarly, if the Hessian is not negative semi-definite, the point x^* is not a local maximum.

Example 2.1.1 *Find the minimum, or minima, of the function $f(x, y) = x^2 + xy + y^2 - 2x - y + 3$.*

Apply the first-order necessary condition to find all stationary points. Since $\nabla f = \begin{pmatrix} 2x + y - 2 \\ x + 2y - 1 \end{pmatrix}$, solving $\nabla f = 0$ yields $x = 1, y = 0$; implying that $(x, y) = (1\ 0)$ is a (unique) stationary point. This point must in fact be optimal, since $\nabla^2 f = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$ is easily seen to be positive definite, thereby fulfilling the sufficient conditions.

In a much more general context, a typical class of algorithms used for solving unconstrained optimization problems are iterative algorithms. An *iterative algorithm* for the optimization problem $p^* = \min_x f(x)$ generates a sequence $x^{(0)}, x^{(1)}, \dots$ of points so that $f(x^{(k)}) \to p^*$ as $k \to \infty$.

Consider the k-th step in a typical iterative algorithm.

- the algorithm chooses a search direction $\Delta x^{(k)} \in \mathbb{R}^n$,
- the algorithm chooses a *step size* $t^{(k)} \in \mathbb{R}$
- the algorithm sets $x^{(k+1)} := x^{(k)} + t^{(k)} \Delta x^{(k)}$

The search direction Δx is called a *descent direction* when $(\nabla f(x))^T \Delta x < 0$. Indeed, then there exists a stepsize t so that the value of the newly generated point is an improvement over the previous point; in other words, there exists a t so that: $f(x + t\Delta x) < f(x)$.

There are many variations of this basic pattern displayed above. Typical names of such algorithms are gradient descent, steepest descent, Newton descent, and line search. Generally, these methods work well (and fast) for finding the minimum of a convex function.

2.2 Lagrangian Duality

In this chapter, however, we are interested in a situation where the variables must satisfy some given set constraints. Let us give a quote from Joseph-Louis Lagrange who already formulated the basic strategy to relate constrained optimization to unconstrained optimization.

In 1788, Joseph-Louis Lagrange wrote: "One can state the following general principle: If one is looking for the maximum or minimum of some function of many variables subject to the condition that these variables are related by a constraint given by one or more equations, *then one should add to the function whose extremum is sought the functions that yield the constraint equations each multiplied by undetermined multipliers and seek the maximum or minimum of the resulting sum as if the variables were independent*. The resulting equations, combined with the constraint equations, will serve to determine all unknowns."

2.2.1 The Lagrangian

Let us state, in a general form, a continuous optimization problem under a given set of constraints.

$$\begin{array}{llll} (P) \quad p^* \equiv & \text{minimize} & f_0(x) & \\ & \text{subject to} & f_i(x) \le 0 & i = 1, \ldots, r, \\ & & h_j(x) = 0 & j = 1, \ldots, s, \\ & & x = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n. & \end{array}$$

Observe that we use p^* to denote the value of an optimum solution to problem (P). We will assume that all f_i and all h_j are differentiable. Notice further that an equality constraint can be rewritten into two inequality constraints, and observe that convexity of $h_j(x)$ as well as $-h_j(x)$ boils down to the function h_j being an affine function. Further, in case all functions f_i are convex, for $i = 0, \ldots, r$ and all h_j are affine, for $j = 1, \ldots, s$, we call problem (P) a *convex* problem.

The main idea of Lagrangian duality is to "move" each of the constraints of this problem to the objective function, while associating a weight to each constraint. That is, we extend the current objective function with a weighted sum of the constraints; these weights are called Lagrangian multipliers. Here is a definition.

Definition 2.2.1 *The* Lagrangian *of problem (P) is:*

$$L(x, \lambda, \mu) \ := \ f_0(x) + \sum_{i=1}^{r} \lambda_i f_i(x) + \sum_{j=1}^{s} \mu_j h_j(x),$$

where $\lambda \ge 0$ *and* μ *arbitrary.*

The λ_i's (μ_j's) are called Lagrangian multipliers associated with the i-th inequality and he j-th equality constraint, respectively. Observe the impact of the added two terms on the value of the objective function: since, by definition, a feasible x has $f_i(x) \le 0$ and $h_j(x) = 0$, the nonnegativity of the λ_i's, and the existence of μ_j's, will ensure that the value of the Lagrangian rises (compared to the original objective) for infeasible solutions. Indeed, when given a specific $\lambda \ge 0$ and μ, the value of the Lagrangian is a lower bound to the objective function value for each feasible point x. We formulate this fundamental fact in a lemma.

Lemma 2.2.1 *If* $\lambda \ge 0$*, then for any* μ*, and any feasible* x*:*

$$L(x, \lambda, \mu) \le f_0(x).$$

2.2.2 The Lagrange dual function

From the Lagrangian, we can define the so-called Lagrangian dual function, which arises by taking the minimum value of the Lagrangian over x.

Definition 2.2.2 *The* Lagrange dual function *corresponding to problem (P) is:*

$$g(\lambda, \mu) = \min_x L(x, \lambda, \mu).$$

This dual function can provide us with lower bounds for the optimal value of our original problem (P). Indeed, it is an easy consequence of Lemma 2.2.1 that, for any $\lambda \geq 0$, and any μ, we have:

Lemma 2.2.2 *If $\lambda \geq 0$, then for any μ*

$$g(\lambda, \mu) \leq p^*.$$

Proof: Suppose x^* is a feasible solution to the original problem (P). Thus we have $f_i(x^*) \leq 0$ and $h_j(x^*) = 0$ for all i. Since, by definition, $\lambda_i \geq 0$, it follows that

$$\sum_{i=1}^{r} \lambda_i f_i(x^*) + \sum_{i=1}^{s} \mu_j h_j(x^*) \leq 0.$$

Thus, for this point x^*, we have $L(x^*, \lambda, \mu) = f_0(x^*) + \sum_{i=1}^{r} \lambda_i f_i(x^*) + \sum_{i=1}^{s} \mu_j h_j(x^*) \leq f_0(x^*)$. Finally, we observe that

$$g(\lambda, \mu) = \min_x L(x, \lambda, \mu) \leq L(x^*, \lambda, \mu) \leq f_0(x^*)$$

holds for *every* feasible x^*, including the one achieving the value p^*. □

2.2.3 The Lagrange dual problem

Thus $g(\lambda, \mu)$ provides us with a lower bound for each choice of $\lambda \geq 0$ and μ. Why not try to find the best one? Indeed, let us consider the problem of finding the $\lambda \geq 0$, and the μ that maximizes $g(\lambda, \mu)$.

First, let us restate our original problem, and call it the primal problem.

$$\begin{array}{llll} (P) & p^* \equiv & \text{minimize} & f_0(x) \\ & & \text{subject to} & f_i(x) \leq 0 \quad i = 1, \ldots, r, \\ & & & h_j(x) = 0 \quad j = 1, \ldots, s. \end{array}$$

Now, we call the problem of maximizing $g(\lambda, \mu)$ the Lagrange dual. More concrete: the corresponding *Lagrange dual* is the problem

$$\begin{aligned}(D)\quad d^* \equiv \quad & \text{maximize} \quad g(\lambda,\mu) \\ & \text{subject to} \quad \lambda_i \geq 0 \quad i=1,\dots,r.\end{aligned}$$

Due to our construction of the Lagrange dual, we have the following compact, yet important inequality:

Lemma 2.2.3 *$p^* \geq d^*$.*

This is called weak duality. Notice that this fact does not rely on convexity of the original problem. The difference between p^* and d^* is called the *optimal duality gap*. Let us consider two examples.

Example 2.2.1 *Determine the optimal primal and the optimal dual objective value for the problem*

$$\min\{x \mid x^3 - y \geq 0,\ y \geq 0\}.$$

The optimal solution to the primal problem can be found by inspection: $p^* = 0$ with $x^* = y^* = 0$. We can express the Lagrangian as follows:

$$L(x,y,\lambda_1,\lambda_2) \quad = \quad x + \lambda_1(y - x^3) - \lambda_2 y.$$

Let us now identify and solve the Lagrange dual of this problem. The Lagrange dual function is:

$$g(\lambda_1,\lambda_2) \ = \ \min_{x,y}\ (x - \lambda_1 x^3) + (\lambda_1 - \lambda_2)y.$$

From this, we observe that

- If $\lambda_1 > 0$, then $x - \lambda_1 x^3$ goes to $-\infty$ as x goes to ∞.
- If $\lambda_1 = 0$, then $x - \lambda_1 x^3 = x$ which also goes to $-\infty$ when x goes to -∞.

Thus, $d^* = -\infty$, and since $p^* = 0$, it follows that the duality gap is infinite.

Example 2.2.2 *Consider, for an $(n \times n)$ symmetric matrix A, the problem min$\{x^T Ax \mid x^T x = 1\}$. Give the Lagrangian, Lagrange dual function, and Lagrange dual of this problem.*

The Lagrangian is seen to be:

$$L(x,\mu) = x^T Ax + \mu(1 - x^T x) \ = \ \mu + x^T(A - \mu I)x.$$

From this, we observe that the Lagrange dual function equals:

$$g(\mu) = \min_x\ (\mu + x^T(A - \mu I)x) = \mu + \min_x x^T(A - \mu I)x.$$

Now, clearly, if $A-\mu I$ is positive semi-definite, then $\min_x\ x^T(A-\mu I)x = 0$, while if $A-\mu I$ is not positive semi-definite then $\min_x\ x^T(A-\mu I)x = -\infty$. Thus, $A-\mu I$ being positive semi-definite is equivalent to $\mu \leq \lambda_{\min}(A)$ (where $\lambda_{\min}(A)$ denotes the smallest eigenvalue of A). Hence $g(\mu) = \mu$ for $\mu \leq \lambda_{\min}$ and $g(\mu) = -\infty$ for $\mu > \lambda_{\min}$. And we conclude that we can write the Lagrange dual as follows:

$$\text{maximize } \mu \text{ subject to } \mu \leq \lambda_{\min}.$$

Hence in this case $p^* = d^*$ (and there is no optimal duality gap).

2.3 The weak Karush-Kuhn-Tucker Optimality conditions

Observe that in the absence of equality constraints, the general optimization problem $\min_x f_0(x)$, subject to $f_i(x) \leq 0$, for $i = 1, \ldots, r$ exhibits the following property. Assuming that respective gradients exist at a given feasible point x^*, application of Farkas' lemma (see Section 1.4.1) yields: *either*, there exist non-negative $\lambda_0, \lambda_1, \ldots, \lambda_r$, such that

$$\begin{bmatrix} \nabla f_0(x^*) & \nabla f_1(x^*) & \ldots & \nabla f_r(x^*) \\ & f_1(x^*) & & \\ & & \ddots & \\ & & & f_r(x^*) \\ 1 & 1 & \ldots & 1 \end{bmatrix} \begin{bmatrix} \lambda_0 \\ \lambda_1 \\ \vdots \\ \lambda_r \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

or there is a direction $d = [d_0^T, d_1, \ldots, d_r, d_\infty]^T$ with $d_\infty > 0$; $d_0^T \nabla f_0(x^*) + d_\infty \leq 0$; and $d_0^T \nabla f_i(x^*) + d_i f_i(x^*) + d_\infty \leq 0$, for $i = 1, \ldots, r$.
Combining the first two properties yields that direction vector d_0 is such that $d_0^T \nabla f_0(x^*) < 0$, and similarly, $d_0^T \nabla f_i(x^*) < 0$, for those i with $f_i(x^*) = 0$. It follows that then, for small enough positive $\varepsilon > 0$: $x^* + \varepsilon d_0$ is feasible (that is: $f_i(x^* + \varepsilon) \leq 0$, for ALL $i = 1, \ldots, r$) and has value $f_0(x^* + \varepsilon d_0) < f_0(x^*)$. In this case x^* cannot be a local minimum.

This gives a proper characterization of potential local and global optima. This can be extended to include equality constraints as well, and this is known as the Fritz-John conditions. If a feasible x^* is to be a local minimum, there must exist corresponding $\lambda_i \geq 0$, $i = 0, 1, \ldots, r$ (and μ_j, $j = 1, \ldots, s$), not ALL zero, that satisfy the matrix equation $\sum_{i=0}^r \lambda_i \nabla f_i(x^*) + \sum_j \mu_j \nabla h_j(x^*) = 0$, where $\lambda_i f_i(x^*) = 0$, for $i > 0$.

Observe that if gradients of the h_j and of the active f_i are linearly independent, then in the solution λ we have $\lambda_0 \neq 0$, and hence we can scale λ and μ so as to have $\lambda_0 = 1$. The set of candidate minima can be exhaustively enumerated by picking $\lambda_0 = 0$ or $\lambda_0 = 1$, and by selecting for each $i > 0$, either $\lambda_i = 0$ or $f_i(x) = 0$. For each of these 2^{r+1} possibilities, solve the resulting non-linear set of $n+r+1+s$ equations (in $n+r+1+s$ variables x, λ_i, μ_j), and test whether the signs of the λ_i and $f_i(x)$ agree.

Under certain extra conditions we can tell more, but this characterization already helps us to find the candidate optima for the general case!

2.4 The Karush-Kuhn-Tucker conditions

We now give the famous Karush-Kuhn-Tucker conditions. Informally, they arise when setting the derivative of the Lagrangian to 0.

Theorem 2.4.1 *(Karush-Kuhn-Tucker (KKT) conditions)*
If a feasible *point x^* is a local minimum, and the optimization problem satisfies a regularity condition, then there exist real numbers $\lambda_1, \ldots, \lambda_r$ and $\mu_1, \ldots, \mu_s$ such that*

- $\nabla f_0(x^*) + \sum_{i=1}^{r} \lambda_i \nabla f_i(x^*) + \sum_{j=1}^{s} \mu_j \nabla h_j(x^*) \ = \ 0,$
- $\lambda_i f_i(x^*) = 0$ *for* $i = 1, \ldots, r,$ *and*
- $\lambda_i \geq 0$ *for* $i = 1, \ldots, r.$

We mention two different regularity conditions:

(i) the gradients of the active inequality constraints, and the equality constraints are linearly independent,

(ii) the problem is convex, and there exists a feasible point x such that $f_i(x) < 0$ for all i (this is known as Slater's condition).

The relevance of the first regularity condition can be deduced from the following example:

Example 2.4.1 *Consider the following problem: minimize x, subject to $y \leq x^3, y \geq -x^3$.*

It is not difficult to verify that the point $(x, y) = (0, 0)$ is a local (even global) optimum solution. It is also not difficult to verify that $\nabla f_1^T(x, y) = (-3x^2\,, 1)$, and $\nabla f_2^T(x, y) = (3x^2\,, -1)$. Thus, when ignoring, in the above statement the phrase "and the optimization problem satisfies a regularity condition", something would be amiss since the equality $\nabla f_0^T + \lambda_1 \nabla f_1^T + \lambda_2 \nabla f_2^T = (0\,,0)$ boils down to $(1\,,0) + \lambda_1(-3x^2\,,1) + \lambda_2(-3x^2\,,-1) = (0\,,0)$, which does not have a solution for λ_1, λ_2 when $x = 0$.

And that is exactly where regularity condition (i) kicks in, since it is indeed the case that, for this problem, the two gradients ∇f_1 and ∇f_2 are linearly dependent in $(0, 0)$.

Another regularity condition is known as Slater's condition: it applies to a convex problem, and states that there must be a feasible point in the interior of the feasible region defined by the inequality constraints. In that case, even more is true. Indeed, if the optimization problem is convex, and satisfies, in addition, Slater's condition, the KKT conditions are necessary and sufficient for optimality. We state this fact as a separate theorem.

Theorem 2.4.2 *The KKT conditions characterize an optimal solution (and hence are both necessary and sufficient),*

- *if $f_0, \ldots, f_r$ are convex and $h_1, \ldots, h_s$ are affine; and*
- *if Slater's condition is satisfied: there exists a point that satisfies all inequality constraints strictly.*

Another equivalent formulation of strong duality for convex optimization:

Theorem 2.4.3 *Assume that a convex optimization problem satisfies Slater's condition (there exists a point that satisfies all inequality constraints strictly).*

Then vector x^* is an optimal solution,
if and only if there exists (λ^*, μ^*) with $\lambda^* \geq 0$ such that the inequality

$$L(x^*, \lambda, \mu) \;\leq\; L(x^*, \lambda^*, \mu^*) \;\leq\; L(x, \lambda^*, \mu^*)$$

is satisfied for all vectors x and for all vectors (λ, μ) with $\lambda \geq 0$.

Such a point (x^*, λ^*, μ^*) is called a *saddle point.* Notice that under the assumptions of the theorem, a saddle point is a KKT point.

The KKT conditions are conditions stated for a proposed point x^*. The conditions can be used however to actually compute an optimum solution, as witnessed by the following examples.

Example 2.4.2 *Let $a, c \in \mathbb{R}^n$ with $c \neq 0$.*

$$\begin{aligned} &\textit{minimize} && \textstyle\sum_{i=1}^n c_i x_i \\ &\textit{subject to} && \textstyle\sum_{i=1}^n (x_i - a_i)^2 \leq 1 \end{aligned}$$

Our goal is, by applying the KKT conditions formulated above, to express the optimum x^* in terms of the problem's parameters a and c. This can be done as follows; observe that KKT can be applied as the problem is convex and the point $x = a$ is an interior solution, Slater's condition holds. KKT yields:

- $c_i + 2\lambda(x_i^* - a_i) = 0$ for $i = 1, \ldots, n$,
- $\lambda \left(\sum_{i=1}^n (x_i^* - a_i)^2 - 1 \right) = 0$, and
- $\lambda \geq 0$

From the second equality it follows that $\sum_i (x_i^* - a_i)^2 = 1$ or $\lambda = 0$. If $\lambda = 0$, then the first set of equations gives that $c_i = 0$, for all i, which contradicts $c \neq 0$.
So $\lambda \neq 0$, and the first set of equalities now gives that $x_i - a_i = -\frac{c_i}{2\lambda}$ for all i. Combining this with the last equation gives: $4\lambda^2 = \sum_i c_i^2$, or in other words, $2\lambda = ||c||$, which is INDEED non-negative. This finally leads to $x_i = a_i - c_i/\|c\|$ and objective $c^T x = c^T a - \|c\|$.

Example 2.4.3 *For $A \in S^n$, show that $min\{x^T Ax \mid x^T x = 1\}$ yields the smallest Eigenvalue $\lambda_{min}(A)$ of matrix A.*

Note that here we do not necessarily have a convex problem. On the other hand, the single constraint is an equation $h(x) = 0$, with gradient $\nabla h(x) = 2x \neq 0$, for any feasible x. So the first regularity condition is satisfied. Setting the gradients of the Lagrangian to x and to μ equal to zero yields:

$$L(x, \mu) \;=\; x^T Ax + \mu(1 - x^T x) \;=\; \mu + x^T(A - \mu I)x$$

$$\nabla_x L(x, \mu) = \nabla x^T Ax + \mu \nabla(1 - x^T x) \;=\; 0$$

$$2x^T A - 2\mu x^T \;=\; 0$$

$$\nabla_\mu L(x, \mu) = 1 - x^T x = 0$$

If $x^T A = \mu x^T$ then either: $x = 0$ or: μ is an Eigenvalue and x is the corresponding Eigenvector. As x has norm 1 it cannot be zero. As A is symmetric, indeed (normalized) Eigenvectors exist; let λ_1 denote the lowest Eigenvalue. If $\mu > \lambda_1$ and $x = E_\mu$, then $f_0(x) = x^T Ax = \mu x^T x = \mu > \lambda_1 = \lambda_1 E_{\lambda_1}^T E_{\lambda_1} = E_{\lambda_1}^T A E_{\lambda_1} = f_0(E_{\lambda_1})$. Hence the minimum is attained at $x = E_\mu$ for $\mu = \lambda_1$.

2.5 Strong Duality

If we have equality for the optimal primal value and the optimal dual value, i.e., if we have $p^* = d^*$, we say that strong duality holds. As we saw earlier, strong duality does not hold in general. Convexity of the primal problem, together with Slater's condition does the trick. Informally said, Slater's condition states that there must exist a feasible solution in the interior of the inequality constraints, i.e., there must exist a point satisfying the inequality constraints strictly. We have the following result.

Theorem 2.5.1 *(strong duality for convex optimization)*
Suppose (convex program) and (Slater's condition is satisfied):

- $f_0, \dots, f_r$ *are convex and* $h_1, \dots, h_s$ *are affine*
- $\exists y : \; f_i(y) < 0$ *for* $i = 1, \dots, r$, *and* $\qquad h_j(y) = 0$ *for* $j = 1, \dots, s$

Then $p^* = d^*$.

Hereunder, we sketch a proof based on Boyd and Vandenberghe [3].
For convenience, we assume there are no equality constraints, i.e., $s = 0$. Moreover we assume $p^* > -\infty$.
Consider $\mathcal{A} := \{ \begin{bmatrix} u \\ t \end{bmatrix} \mid \exists x : f_i(x) \leq u_i, \; i = 1, \dots, r, \; f_0(x) \leq t \}$.

1. $\mathcal{A}$ is a convex set
2. $p^* = \min\{t \mid \begin{bmatrix} 0 \\ t \end{bmatrix} \in \mathcal{A}\}$
3. some hyperplane supports $\mathcal{A}$ at $(0, p^*)$; say
$$\begin{bmatrix} \lambda^* \\ \mu^* \end{bmatrix}^T \begin{bmatrix} u \\ t \end{bmatrix} \geq \alpha \text{ for } \begin{bmatrix} u \\ t \end{bmatrix} \in \mathcal{A}; \qquad \begin{bmatrix} \lambda^* \\ \mu^* \end{bmatrix}^T \begin{bmatrix} 0 \\ p^* \end{bmatrix} = \alpha$$
4. $\mu^* \geq 0, \lambda^* \geq 0$
5. If $\mu^* = 0$, then $0 > \sum_i \lambda_i^* f_i(y) \geq \alpha = 0$; contradiction.
6. If $\mu^* > 0$, then $g(\lambda^*/\mu^*) = p^*$.

The detailed arguments are as follows:

1. For $\alpha, \beta \geq 0$, $\alpha + \beta = 1$, and $(u^1, t_1), (u^2, t_2) \in \mathcal{A}$, we have that there are x_1, x_2 such that $\alpha f_i(x_1) \leq \alpha u_i^1$, $\beta f_i(x_2) \leq \beta u_i^2$, for all $i = 1, \ldots, r$ and $\alpha f_0(x_1) + \beta f_0(x_2) \leq \alpha t_1 + \beta t_2$. By convexity of the domains the point $\hat{x} = \alpha x_1 + \beta x_2$ exists; by convexity of the f_i we have that for all $i > 0$: $f_i(\hat{x}) \leq \alpha u_i^1 + \beta u_i^2$, and $f_0(\hat{x}) \leq \alpha t_1 + \beta t_2$; hence $\alpha(u^1, t_1) + \beta(u^2, t_2) \in \mathcal{A}$.

2. If $p^* = f_0(x^*)$ for some feasible x^*, then $f_i(x^*) \leq 0$, for $i > 0$, and so $((0, 0, \ldots, 0)^T, p^*) \in \mathcal{A}$. If there is a $t < p^*$ such that $(0, t) \in \mathcal{A}$, the corresponding existing vector x^t would satisfy $f_i(x^t) \leq 0$, for $i > 0$, and $f_0(x^t) \leq t < p^*$, contradicting the definition of p^* as minimum.

3. As $(0, p^* - \varepsilon) \notin \mathcal{A}$, for $\varepsilon \downarrow 0$, $(0, p^*)$ lies on the boundary of $\mathcal{A}$. Hence there is a hyperplane supporting $\mathcal{A}$ at point $(0, p^*)$, that is, a vector $(\lambda^*, \mu^*) \neq (0, 0)$ and an α exist such that for all $(u, t) \in \mathcal{A}$: $u^T \lambda^* + t\mu^* \geq \alpha = 0^T \lambda^* + p^* \mu^* = p^* \mu^*$.

4. Notice that by definition of $\mathcal{A}$, if $(u, t) \in \mathcal{A}$, then so is $(u, t) + \gamma e_i$, for any $\gamma > 0$, and any unit-vector e_i. If (λ^*, μ^*) would have some negative component at coordinate i, then $(\lambda^*, \mu^*)^T [(0, p^*) + \gamma e_i] < \mu^* p^*$ for $\gamma > 0$. So $\lambda^* \geq 0$, $\mu^* \geq 0$.

5. Slater's condition holds, so for some y: $f_i(y) < 0$, for all $i > 0$. Hence, if $\mu^* = 0$, then λ^* is not all-zero, and so $0 > \sum_i \lambda_i^* f_i(y) \geq \alpha = 0$; contradiction. So $\mu^* > 0$.

6. By definition $g(\lambda) = \min_x f_0(x) + \sum_{i>0} (\lambda)_i * f_i(x)$. We have $d^* = \max_{\lambda \geq 0} g(\lambda) \geq g(\lambda^*/\mu^*)$ $\geq \min_{(u,t) \in \mathcal{A}} t + \sum_{i>0} (\lambda^*/\mu^*)_i * u_i$, as for each x: $(u; t)^x := (f_1(x), \ldots, f_r(x); f_0(x)) \in \mathcal{A}$;
   Hence, we have $g(\lambda^*/\mu^*) \geq \frac{1}{\mu^*} \min_{(u,t) \in \mathcal{A}} \mu^* t + \sum_{i>0} (\lambda^*)_i * u_i \geq \frac{\alpha}{\mu^*} = p^*$.
   Actually we have equality throughout as $d^* \leq p^*$.

2.5.1 Complementary Slackness

The presence of strong duality has a number of interesting consequences for the possible values of the Lagrangian multipliers. Let us phrase the implications of strong duality, and let x^* denote an optimal primal solution, and (λ^*, μ^*) is an optimal dual solution. We have

$$\begin{aligned} f_0(x^*) &= g(\lambda^*, \mu^*) \\ &= \min_x (f_0(x) + \sum_{i=1}^r \lambda_i^* f_i(x) + \sum_{i=1}^s \mu_i^* h_j(x)) \\ &\leq f_0(x^*) + \sum_{i=1}^r \lambda_i^* f_i(x^*) + \sum_{i=1}^s \mu_i^* h_j(x^*) \\ &\leq f_0(x^*). \end{aligned}$$

When accepting this chain of (in)equalities, there is no choice but accepting that

$$\sum_{i=1}^r \lambda_i^* f_i(x^*) = 0,$$

which implies that

$$\lambda_i^* f_i(x^*) = 0 \text{ for each } i.$$

These conditions are known as *complementary slackness*; indeed they imply that for each constraint either it is satisfied by equality in an optimal primal solution, or the corresponding optimal Lagrangian multiplier equals zero (or both).

2.6 Duality Theorems

Duality theorems, or min-max relations, such as the one discussed above, can be found in many different fields, and are found under various names. A duality result reveals an understanding of the problem, and can be used algorithmically. We now mention a number of duality results.

Linear programming duality. Consider the linear program

$$\min\{c^T x \mid Ax \geq b\}.$$

The Lagrangian is

$$L(x, \lambda) := c^T x + \lambda^T (b - Ax).$$

The Lagrange dual function is

$$g(\lambda) = \min_x c^T x + \lambda^T (b - Ax) = \begin{cases} \lambda^T b & \text{if } c^T = \lambda^T A \\ -\infty & \text{otherwise} \end{cases}$$

So the dual is

$$\max\{g(\lambda) \mid \lambda \geq 0\} = \max\{\lambda^T b \mid c^T = \lambda^T A, \lambda \geq 0\}$$

Strong convex duality implies LP duality.

Theorem 2.6.1 *(Kőnig, 1931) In a bipartite graph $G = (X \cup Y, E)$, the number of edges in a maximum matching is equal to the number of vertices in a minimum vertex cover.*

Theorem 2.6.2 *(Hall's marriage theorem, 1935) A bipartite graph $G = (X \cup Y, E)$ contains a matching that covers X, if and only if $|N(S)| \geq |S|$ for all $S \subset X$.*

Theorem 2.6.3 *(Dilworth, 1950) For any (finite) partially ordered set, the maximum size of an antichain is equal to the minimum number of chains in a partition into chains.*

Theorem 2.6.4 *(Menger, 1927) For any undirected graph $G = (V, E)$ and $x, y \in V$, the maximum number of pairwise vertex-disjoint paths from x to y is equal to the minimum number of vertices in a vertex cut separating x from y.*

This result is actually a predecessor and close relative of the max-flow min-cut theorem.

2.7 Solving convex programs

In this section, we sketch an algorithm for solving the following convex optimization problem.

$$\min\{f_0(x) \mid f_i(x) \leq 0, i = 1\ldots, r,\ Ax = b\}.$$

We assume that there exists a feasible solution, even more we assume that Slater's condition holds (that is, there exists an x such that $Ax = b, f_i(x) < 0$ for all i). Thus, KKT applies, and there exist an optimal x^* and optimal (λ^*, μ^*) satisfying the KKT conditions.

Our problem is equivalent to the problem

$$\min\{f_0(x) + \sum_{i=1}^{r} I_-(f_i(x)) \mid Ax = b\}$$

where $I_- : \mathbb{R} \to \mathbb{R}$ is the *indicator function* of the set $\{u \mid u \leq 0\}$:

$$I_-(u) = \begin{cases} 0 & \text{if } u \leq 0 \\ \infty & \text{otherwise} \end{cases}$$

A minor point that spells potential trouble: this indicator function I_- is not differentiable! In order to circumvent this issue, let us define the following approximate indicator function:

Definition 2.7.1
$$\widehat{I}_-(u) = \left\{ \begin{array}{cc} -\frac{1}{t}\log(-u) & \textit{if } u < 0 \\ \infty & \textit{otherwise} \end{array} \right) ,$$

where t is a given parameter that determines the quality of the approximation.

Notice that

- The function $\widehat{I}_-$ is convex and non-decreasing (just as I_-),
- The function $\widehat{I}_-$ is differentiable (unlike I_-), and
- All sublevel sets of $\widehat{I}_-$ are closed.

Notice that, as $t \to \infty$, the function $\widehat{I}_-$ approximates I_- increasingly well. Instead of solving the original problem, this property allows us to solve an approximation of it. Indeed, we approximate $\min\{f_0(x) \mid f_i(x) \leq 0, i = 1 \ldots, r,\ Ax = b\}$ by

$$\min\{f_0(x) + \frac{1}{t}\phi(x) \mid Ax = b\}$$

where $t > 0$ and $\phi(x) := \sum_{i=1}^{r} - \log(-f_i(x))$.

The function $\phi(x)$ is called the *logarithmic barrier*. Let us now introduce the notion of a central path, which can be seen as a sequence of feasible solutions converging to the optimum solution of the original problem.

Definition 2.7.2 *The set* $\{x^*(t) \mid t > 0\}$ *is the* central path, *where*

$$x^*(t) := \textit{arg min}\{f_0(x) + \frac{1}{t}\phi(x) \mid Ax = b\}$$

Theorem 2.7.1 *The central path leads to the optimum.*

Proof:

The Lagrange dual function of $p^* = \min\{f_0(x) \mid f_i(x) \leq 0, Ax = b\}$ is

$$g(\lambda, \mu) \ := \ \inf_x \ f_0(x) + \sum_i \lambda_i f_i(x) + \mu^T(Ax - b)$$

For $x^* = \text{argmin}\{f_0(x) + \frac{1}{t}\phi(x) \mid Ax = b\}$, we have by KKT that

$$\nabla f_0(x^*) + \sum_i \frac{1}{-tf_i(x^*)} \nabla f_i(x^*) + A^T\mu^* = 0$$

for some μ^*. By setting $\lambda_i^* := 1/(-tf_i(x^*)) > 0$, we get

$$g(\lambda^*, \mu^*) = f_0(x^*) + \sum_{i=1}^{r} \lambda_i^* f_i(x^*) + (\mu^*)^T(Ax^* - b) = f_0(x^*) - \frac{r}{t}$$

Hence $p^* \ \leq \ f_0(x^*) \ = \ g(\lambda^*, \mu^*) + r/t \ \leq \ p^* + r/t.$

$\square$

From these last inequalities, it follows that, in order to approximate $p^* = \min\{f_0(x) \mid f_i(x) \leq 0, i = 1 \ldots, r, \ Ax = b\}$ within some given additive error $\epsilon > 0$, it suffices to solve

$$x^*(t) \ = \ \text{argmin}\{f_0(x) + \frac{1}{t}\phi(x) \mid Ax = b\}$$

with $t = r/\epsilon$ (so that $r/t = \epsilon$).

We proceed to argue that the presence of linear equality constraints does not complicate matters:

Consider the optimization problem $\min\{f(x) \mid Ax = b\}$.

If $\{x \mid Ax = b\} = \{Wy + v \mid y \in \mathbb{R}^k\}$, then this problem is equivalent to the unconstrained minimization problem $\min\{f(Wy + v) \mid y \in \mathbb{R}^k\}$, that is, the unconstrained minimization of $g : y \mapsto f(Wy + v)$.

In the same fashion, the equality constraints can be eliminated from

$$\min\{f_0(x) + \frac{1}{t}\phi(x) \mid Ax = b.\}$$

In order to approximate $p^* = \min\{f_0(x) \mid f_i(x) \leq 0, i = 1 \ldots, r, \ Ax = b\}$ within error $\epsilon > 0$, it suffices to solve the unconstrained problem

$$y^*(t) \ = \ \text{argmin}\{f_0(Wy + v) + \frac{1}{t}\phi(Wy + v)\}$$

with $t = r/\epsilon$.

As directly solving this problem for high t is usually inefficient, there is the barrier method: Given a strictly feasible $y = y^{(0)}$ and $t = t^{(0)} > 0$, do

1. compute $y^*(t)$, starting the solution algorithm with y (centering)
2. put $y \leftarrow y^*(t)$ (update)
3. if $r/t < \epsilon$ then quit; else put $t \leftarrow \mu t$ and repeat. (increase)

Centering step: by using the Newton method

Example 2.7.1 *Design a cylindrical tin can with volume at least v units, such that the total surface area is minimal.*

For height h and radius r, this problem becomes

$$\begin{aligned} &\text{minimize} && f(r,h) := 2\pi(r^2 + rh) \\ &\text{subject to} && \pi r^2 h \geq v \\ &&& r > 0 \text{ and } h > 0 \end{aligned}$$

This is not a convex optimization problem!

The substitution $r = e^x$ and $h = e^y$ yields the convex problem

$$\begin{aligned} &\text{minimize} && g(x,y) := 2\pi(e^{2x} + e^{x+y}) \\ &\text{subject to} && -2x - y - \ln\ v - \ln\ \pi \leq 0 \\ &&& x, y \in \mathbb{R} \end{aligned}$$

Let's write down the Lagrangian:

$$L(x,y,\lambda) = 2\pi(e^{2x} + e^{x+y}) + \lambda\,(\ln\ v - \ln\ \pi - 2x - y).$$

Next, applying KKT gives us

$$2\pi \begin{bmatrix} 2e^{2x} + e^{x+y} \\ e^{x+y} \end{bmatrix} = \lambda \begin{bmatrix} 2 \\ 1 \end{bmatrix}.$$

Hence $2e^{2x} + e^{x+y} = 2e^{x+y}$, whence $2e^x = e^y$ and $2r = h$. As the constraint holds with equality (why?), we can express r and h in terms of v.

2.8 Exercises

Exercise 1

Find all stationary points of the following functions. Determine all local minima, local maxima, and saddle points.

(a) $f(x,y) = \exp(x^2 + y^2) - x^2 - 2y^2$

(b) $f(x,y,z) = xy\exp(-x - y - z)$

(c) $f(x,y,z) = 2x^2 + y^2 + z^2 + xy + yz - 6x - 7y - 8z$

(d) $f(x,y) = (x^2 - 1)^2 + (xy - x - 1)^2$

(e) $f(x,y) = \log(1 + x^2 + y^2) - \sqrt{x^2 + y^2}$

(f) $f(x,y) = x^3y^3 - x^2 - y^2 + xy - 4$

(g) $f(x,y,z) = \frac{z^2}{x^2+y^2} + \frac{y^2}{x^2+z^2} + \frac{x^2}{y^2+z^2}$

(h) $f(x,y) = \frac{\log 1+x^2+y^2}{\exp(x^2+y^2)}$

Exercise 2

Let q be real parameter. Find all stationary points of the function $f(x,y) = x^3 - 3qxy + y^3$. Determine all local minima, local maxima, and saddle points.

Exercise 3

Find, for each real value of q, the stationary points of the function $f(x,y) = x^2 - qxy^2 + 2y^4$.

Exercise 4

Let $U \subset \mathbb{R}^n$ be a compact convex non-empty set, and let $v \in \mathbb{R}^n$.

(a) Prove that there exists a unique point in U that is closest to v.

(b) Give an example so that exactly two points in U are farthest from v.

(c) Give an example so that infinitely many points in U are farthest from v.

Exercise 5

Let $U \subseteq \mathbb{R}^n$ be a convex set, and let $f : U \to \mathbb{R}$ be a convex function.

(a) Prove that every local minimum of f is a global minimum.

(b) Prove that for strictly convex f, every local maximum of f is an extreme point of U.

Exercise 6

Show that for real $u, x, y, z > 0$, the following function has no minimum:

$$f(u,x,y,z) \;=\; \frac{1}{x^3} + \frac{2z^4}{y^6} + 3u^3z^2 + \frac{5x^2y^4z^2}{u}$$

Exercise 7

Show that every convex program (P) can be rewritten into an equivalent convex program (Q) with a linear objective function.

Exercise 8

Write down the KKT optimality conditions for the point $x^* = (1,1)$ in the following formulation. Is x^* a global optimum?

$$\begin{array}{lll} \min & -7x_1 - 5x_2 & \\ \text{s.t.} & 2x_1^2 + x_2^2 + x_1x_2 - 4 & \leq 0 \\ & x_1^2 + x_2^2 - 2 & \leq 0 \\ & -2x_1 + 1 & \leq 0 \end{array}$$

Exercise 9

Write down the KKT optimality conditions for the point $x^* = (1/2, -1/2, 1/2)$ in the following formulation. Is x^* a global optimum?

$$\begin{array}{lll} \min & x_3 & \\ \text{s.t.} & x_1 + x_2 & \leq 0 \\ & x_1^2 - 4 & \leq 0 \\ & x_1^2 - 2x_1 + x_2^2 - x_3 + 1 & \leq 0 \end{array}$$

Exercise 10

Write down the KKT optimality conditions for the point $x^* = (0, 1/2)$ in the following formulation. Is x^* a global optimum?

$$\begin{array}{lll} \min & 4x_1^2 + 2x_2^2 - 6x_1x_2 + x_1 & \\ \text{s.t.} & 2x_1 - x_2 & \leq 0 \\ & 2x_1 - 2x_2 & \leq -1/2 \\ & x_1 \leq 0,\ x_2 \geq 0 & \end{array}$$

Exercise 11

Write down the KKT optimality conditions for the point $x^* = (1, 0)$ in the following formulation. Is x^* a global optimum?

$$\begin{array}{lll} \min & x_1^2 + 3x_2^2 - x_1 & \\ \text{s.t.} & x_1^2 - x_2 & \leq 1 \\ & x_1 + x_2 & \geq 1 \end{array}$$

Exercise 12

Find the minimum value of the function $f(x, y) = (x - 2)^2 + (y - 1)^2$ subject to the conditions $y \geq x^2$ and $x + y \leq 2$.

Exercise 13

Let $q > 0$ be a real parameter. Write down the KKT conditions for the point $x^* = (0, 0)$ in the following formulation. For which values of q is x^* a global optimum?

$$\begin{array}{ll} \min & x^2 + (y - 1)^2 \\ \text{s.t.} & qx^2 - y \geq 0 \end{array}$$

Exercise 14

Prove that the following problem is convex. Use the KKT conditions to find an optimal solution.

$$\begin{aligned} \min \quad & 2x_1 \arctan(x_1) - \ln(x_1^2+1) + x_2^4 + (x_3-1)^2 \\ \text{s.t.} \quad & x_1^2 + x_2^2 + x_3^2 - 4 \le 0 \\ & -x_1 \le 0 \end{aligned}$$

Exercise 15

For a real parameter q, use the KKT conditions to find an optimal solution to the following formulation.

$$\begin{aligned} \max \quad & x + qy \\ \text{s.t.} \quad & x^2 + y^2 \le 1 \\ & x + y \ge 0 \end{aligned}$$

Exercise 16

Argue that the KKT conditions are necessary and sufficient conditions for optimality of a solution satisfying the KKT conditions in following formulation. Solve this problem.

$$\begin{aligned} \min \quad & x_1^2 + x_1 + x_2 + x_2^2 \\ \text{s.t.} \quad & x_1 - 2x_2 \le -1 \\ & 2x_1 + x_2 \le 2 \end{aligned}$$

Exercise 17

Is the following problem convex? Determine all stationary points. Determine an optimal solution.

$$\begin{aligned} \min \quad & x_1^2 - x_2^2 \\ \text{s.t.} \quad & x_2 - 1 \le 0 \\ & -x_2 \le 0 \end{aligned}$$

Exercise 18

Consider the following formulation.

$$\begin{aligned} \min \quad & x_3 \\ \text{s.t.} \quad & (x_1-1)^2 + x_2^2 + x_3^2 - 1 \ge 0 \\ & (x_1+1)^2 + x_2^2 + x_3^2 - 1 \ge 0 \\ & x_1^2 + x_2^2 + x_3^2 - 4 \le 0 \end{aligned}$$

(a) Prove that the feasible region U is not convex.

(b) Compute the convex hull of U.

(c) Find the optimal solutions minimizing x_3 over the convex hull of U. Which of these solutions are optimal solutions for the original problem?

Exercise 19

Determine all stationary points and all global maximizers for the following formulation.

$$\begin{aligned} \max \quad & \textstyle\sum_{i=1}^5 x_i^3 \\ \text{s.t.} \quad & \textstyle\sum_{i=1}^5 x_i^2 = 1 \end{aligned}$$

Exercise 20

For real numbers $c_1, \ldots, c_n$ with $c_1 < 0$ and $c_2, \ldots, c_n > 0$, consider the following formulation and determine an optimal solution.

$$\begin{aligned} \min \quad & \textstyle\sum_{i=1}^n c_i x_i \\ \text{s.t.} \quad & \textstyle\sum_{i=1}^n x_i^2 \le 1 \\ & x_1, \ldots, x_n \ge 0 \end{aligned}$$

Exercise 21

For real numbers $c_1, \ldots, c_n > 0$ and $b > 0$, find an optimal solution for the following formulation and show that it is unique.

$$\begin{aligned} \min \quad & \textstyle\sum_{i=1}^n c_i x_i^2 \\ \text{s.t.} \quad & \textstyle\sum_{i=1}^n x_i = b \end{aligned}$$

Exercise 22

Formulate the following problems, determine the KKT conditions, and find the optimal solutions.

(a) Find the area of the largest isosceles triangle ABC (with $|AC| = |BC|$) that is contained in the unit circle.

(b) Find the maximum surface area of a rectangular box whose twelve edges have total length 24.

(c) Find a point P in the plane that minimizes the sum of the squared distances from three given points A, B, C.

(d) Find a point P in the plane that minimizes the sum of the distances from three given points A, B, C. (You may assume that all angles of the triangle ABC are smaller than $2\pi/3$.)

Exercise 23

Determine the minimum of

$$f(x,y,z) = \frac{x^2}{(x-1)^2} + \frac{y^2}{(y-1)^2} + \frac{z^2}{(z-1)^2}$$

subject to the constraints $xyz = 1$ and $x \neq 1$, $y \neq 1$, $z \neq 1$.

Exercise 24

Give the Lagrangian, Lagrange dual function, and Lagrange dual of the problem $\min\{x^T x \mid Ax = b,\ x \geq 0\}$.

Exercise 25

Give the Lagrangian, Lagrange dual function, and Lagrange dual of the problem $\min\{\sum_{i=1}^{3} x_i \log(x_i)\} \mid x_1 + x_2 + x_3 = 1,\ x_1 + 2x_2 + 4x_3 \leq 2\}$.

Exercise 26

Give the Lagrangian, Lagrange dual function, and Lagrange dual of the problem $\min\{x^T Px \mid x^T Qx \geq 1,\ x^T x = 1\}$.

Part II

Discrete Optimization

Chapter 3

Combinatorial optimization problems and their formulations

In this introductory chapter on discrete optimization, we give an informal notion of combinatorial optimization problems. The defining characteristic of a combinatorial optimization problem is that it has a finite number of feasible solutions. We discuss two basic combinatorial optimization problems, namely the *matching* problem (Section 3.1), and the *independent set* problem (Section 3.2), and show how to formulate these problems as an integer program. We discuss in Section 3.3 the relation between a problem's difficulty and its formulation, and discuss some additional problems and their formulation in Section 3.4. We show in Section 3.5 that a problem may admit multiple, distinct, formulations, give a description of a general formulation in Section 3.6, describe some pitfalls in Section 3.7, and end with satisfiability discussed in Section 3.8.

3.1 The Matching Problem

Here is a story. A group of 10 friends, named Abigail, Bernard, Claudette, Dirk, Ethan, Franz, Geraldine, Hope, Isa and Joey decide to spend a night in a particular hotel. They have booked 5 two-person rooms, and the question they are now facing is: who sleeps with whom? To answer this delicate question, you decide to ask (confidentially) for each person's preferences. More precisely, you ask the following nine questions to each of the ten persons: is it OK for you to spend the night in the same room as person X? The answer is either yes or no, and the results are summarized in the table below.

Person	Abigail	Bernard	Claudette	Dirk	Ethan	Franz	Geraldine	Hope	Isa	Joey
A: are you willing ...?	-	yes	yes	yes	yes	yes	no	no	yes	no
B: are you willing ...?	yes	-	yes	no	yes	yes	yes	no	yes	no
C: are you willing ...?	yes	no	-	yes	no	no	yes	no	yes	no
D: are you willing ...?	yes	no	yes	-	yes	no	yes	no	yes	yes
E: are you willing ...?	no	no	yes	yes	-	yes	yes	yes	yes	no
F: are you willing ...?	yes	yes	yes	no	yes	-	yes	no	yes	yes
G: are you willing ...?	yes	no	yes	yes	no	no	-	yes	yes	no
H: are you willing ...?	yes	yes	yes	no	yes	no	yes	-	yes	no
I: are you willing ...?	no	no	no	no	yes	no	no	yes	-	yes
J: are you willing ...?	yes	no	yes	yes	no	yes	yes	no	yes	-

Table 3.1: Who is willing to sleep with whom?

The acquired information reveals some interesting social dynamics; for instance, while Isa seems to be pretty popular with everyone (everybody is willing to have her as a roommate), she herself is quite picky when it comes to selecting a roommate: only Hope and Joey qualify. Of course, preferences are not necessarily symmetric: for instance, while Abigail is willing to spend the night with Ethan, Ethan is not willing to spend the night with Abigail.

You decide to call a pair of persons X, Y *compatible* if person X is willing to sleep with person Y and person Y is willing to sleep with person X. Of course, your goal is to find 5 compatible pairs containing all 10 persons, or, if that turns out to be impossible, to find a maximum number of compatible pairs. In order to organize the information you have assembled in Table 3.1, you decide to set up a graph that has a node for each person, and two nodes are connected if and only if the corresponding persons each want to sleep with one another (thus, the nodes corresponding to Abigail and Ethan are not connected). The resulting graph has ten nodes called A, B, until J, and is represented in Figure 3.1.

Selecting an edge in this graph can be interpreted as selecting a pair of persons sharing the night. Let us say that two edges are *independent* when the two edges do not have a node in common; for instance edges $\{A, B\}$ and $\{C, D\}$ are independent, while the two edges $\{A, B\}$ and $\{A, C\}$ are not independent. So, it all boils down to the following question: can you select five edges in our graph G such that each pair of selected edges is independent?

Of course, some puzzling will give you the answer to the question for this particular instance. In general, however, puzzling is not a very satisfying strategy; larger instances may not be so easy to solve in this way, and moreover, in case you do not find a yes-answer to the question above, you will never

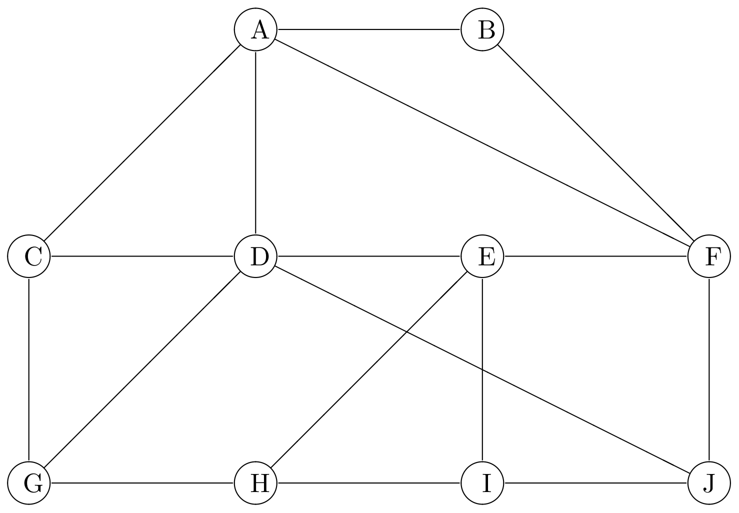


Figure 3.1: Our graph

know whether the true answer is “no”, or whether your puzzling was simply insufficient.

So, let us formulate this question as an integer program. By realizing that the crucial decision is to decide whether a particular pair of persons shares a room, i.e., to select an edge or not, we come up with the following formulation.

$$\begin{array}{llr}
\text{maximize} & x_{AB}+x_{AC}+x_{AD}+x_{AF}+x_{BF}+x_{CD}+x_{CG}+x_{DE}+x_{DG}+x_{DJ}+ & \\
& +x_{EF}+x_{EH}+x_{EI}+x_{FJ}+x_{GH}+x_{HI}+x_{IJ}, & \\
\text{subject to} & x_{AB}+x_{AC}+x_{AD}+x_{AF} & \leq 1, \\
& x_{AB}+x_{BF} & \leq 1, \\
& x_{AC}+x_{CD}+x_{CG} & \leq 1, \\
& x_{AD}+x_{CD}+x_{DE}+x_{DG}+x_{DJ} & \leq 1, \\
& x_{DE}+x_{EF}+x_{EH}+x_{EI} & \leq 1, \\
& x_{AF}+x_{BF}+x_{EF}+x_{FJ} & \leq 1, \\
& x_{CG}+x_{DG}+x_{GH} & \leq 1, \\
& x_{EH}+x_{GH}+x_{HI} & \leq 1, \\
& x_{EI}+x_{HI}+x_{IJ} & \leq 1, \\
& x_{DJ}+x_{FJ}+x_{IJ} & \leq 1, \\
& \text{all variables in} & \{0,1\}.
\end{array}$$

Clearly, this model features a binary variable for each edge. Of course, this formulation is tailored to the specific instance depicted in Figure 3.1. To be able to solve problems with a different number of persons and with different relationships, we want to formulate the problem in a more general way.

Thus, let us formulate this problem in a more general context. We are given an arbitrary graph $G = (V, E)$. A subset of the edges $M \subseteq E$ is called a *matching* if each vertex of V is incident to at most one edge of M. In other words, reverting to the terminology just introduced, M is a matching if each pair of edges of M is independent (see Figure 3.2 for an illustration).

Then, stated in other words: the problem is to find a matching in G consisting of as many edges as possible (a maximum cardinality matching). This problem is known as the Matching Problem, and is one of the fundamental problems in Combinatorial Optimization. A matching is called *maximum* when no matching of larger cardinality exists. A matching is called *maximal* when it cannot be enlarged. Clearly, a maximum matching is maximal; the reverse, however, is not necessarily true (see the exercises).

The matching problem can be formulated as an integer program as follows, where we use the symbol

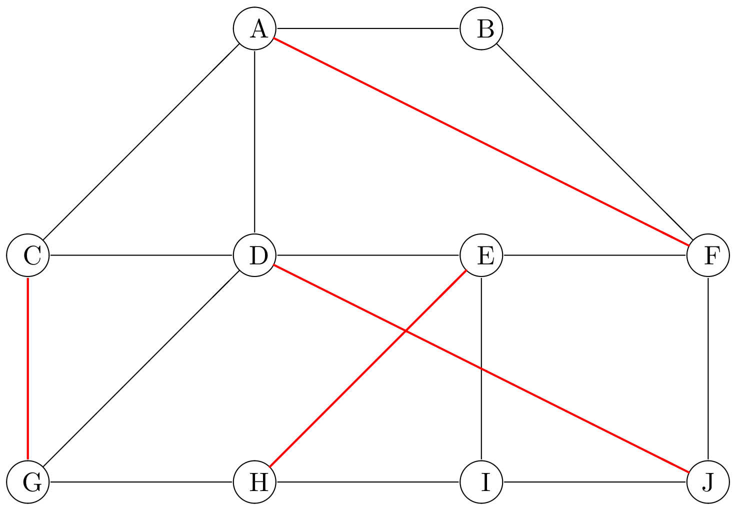


Figure 3.2: The set of red edges corresponds to a matching in our graph

$\delta(v)$ to denote the set of edges that are incident to vertex $v \in V$. For instance, in Figure 3.1, $\delta(J) = \{\{D, J\}, \{F, J\}, \{I, J\}\}$. We define a binary variable for each edge $e \in E$ as follows:

$$x_e = \begin{cases} 1 & \text{if edge } e \text{ is selected in the matching;} \\ 0 & \text{otherwise.} \end{cases}$$

And here is the integer program:

$$\begin{aligned} \text{maximize} \quad & \sum_{e \in E} x_e && && (3.1) \\ \text{subject to} \quad & \sum_{e \in \delta(v)} x_e \leq 1 && \text{for each } v \in V, && (3.2) \\ & x_e \in \{0, 1\} && \text{for each } e \in E. && (3.3) \end{aligned}$$

There are a number of popular variations of the matching problem:

- The *weighted* matching problem is a generalization of the cardinality matching problem above by assuming that there is a given weight function w defined on the edges. Then, the objective is to find a maximum weight matching, and the objective is changed accordingly to $\max \sum_{e \in E} w_e x_e$. (Obviously, it is a generalization, since the cardinality matching problem arises when $w_e = 1$ for each $e \in E$).

- In the *perfect* matching problem the set of feasible solutions is restricted to *perfect matchings*. These are matchings such that each vertex is incident to precisely one edge in the matching; then, constraints (3.2) become equalities. Notice that a perfect matching may not exist (consider e.g. a triangle), whereas there is always a feasible solution to (3.1)-(3.3). One can distinguish a weighted and an unweighted variant of the perfect matching problem.

- Instead of maximizing, one can also be interested in finding a minimum-weight (perfect) matching.

- The *bottleneck* weighted (perfect) matching problem. Again, there is a weight (or cost) w_e for each edge $e \in E$; however, now the objective function is to minimize the largest weight of an edge used in the (perfect) matching, i.e., to minimize $\max_e w_e x_e$.

- An important special case arises when the underlying graph is so-called *bipartite*. In such a setting, each node belongs to one of two classes (eg boys and girls), and edges exist only between a pair nodes of different classes. The weighted matching problem on a bipartite graph is known as the *assignment* problem. Its formulation as an integer programming problem is well-known; in the following formulation, we use a binary variable $x_{i,j}$ to denote whether the edge between nodes i and j ($i \neq j$) is selected:

$$\begin{aligned}
\text{max or min} \quad & \sum_i \sum_j c_{i,j} x_{i,j} & & (3.4)\\
\text{subject to} \quad & \sum_i x_{i,j} = 1 & \text{for each } j, & \quad (3.5)\\
\text{subject to} \quad & \sum_j x_{i,j} = 1 & \text{for each } i, & \quad (3.6)\\
& x_{i,j} \in \{0,1\} & \text{for each } i, j, i \neq j. & \quad (3.7)
\end{aligned}$$

We emphasize that we distinguish between, on the one hand, the problem itself, and, on the other hand, its formulation as an integer program. Indeed, *these two are not the same!*.

In fact, in some cases it is appropriate to show the correctness of a formulation. Such an argument is usually based on a correspondence between feasible solutions of the problem, and vectors of decision variables satisfying the constraints.

Let us illustrate this matter for the matching problem. Notice that there is a one-to-one correspondence between subsets of the set of edges E and the 0-1 vectors defined by the variables, which are indexed by the edges. To prove that this formulation is correct, we must show that there is a one-to-one correspondence between the subsets of the edges that define matchings and the feasible solutions of the above formulation. Moreover, we must show that the matching and its corresponding vector have the same value. This can be done as follows. First, consider an arbitrary matching M in a graph. By definition, this implies that each node $v \in V$ of the graph is incident with at most one edge of M. Let us now construct a solution vector x_M in a straightforward manner: we put a "1" in x_M when the corresponding edge is in M, and otherwise we put a "0" in the vector x_M. Clearly, x_M is a 0-1 vector, and obviously satisfies (3.3). Also, the solution vector x_M corresponding to M satisfies the constraints (3.2), since at most one of the variables in the left-hand side has value 1. Thus, a matching M corresponds to a feasible solution of the integer program. Second, consider a subset of the edges E' that is NOT a matching. Then there is a vertex, say v, that is incident to at least two edges in E'. But then, at least two of the variables in the left-hand side of (3.2) have value 1 for this particular vertex. Thus, the vector corresponding to E' is not feasible in the formulation. Finally, we observe that the value of each set $E' \subseteq E$ is equal to its number of edges, i.e., $|E'| = \sum_{e \in E'} 1 = \sum_{e \in E} x_e$.

In general, it may not be trivial to prove the one-to-one correspondence between feasible solutions of a combinatorial problem and solutions of its corresponding formulation, i.e., solutions satisfying the constraints. For many problems, however, a correctness proof is omitted because the problem is an (extension of) a well-known problem, and the correctness of a formulation is evident.

3.2 The Independent Set Problem

Here is another story. Suppose that Abigail is going to celebrate her birthday by organizing a (small) intimate dinner. In order to avoid boring conversations, she wants her guests to be as diverse as possible. Clearly, she wants to know each guest herself, and in fact, she considers anybody who is linked to her Facebook-page as somebody she knows, and hence as a candidate to invite to her dinner. However, she does not want to invite two persons who know each other. And by knowing each other, Abigail means: being directly linked to each other on Facebook. So, Abigail only wants to invite persons that (i) are linked to her on Facbook, and (ii) are not linked to each other on Facebook. Given these restrictions, she wants to maximize the number of persons invited.

To get a grip on this problem, Abigail decides to set up a network where there is a node for each person linked to her, and where two nodes are connected if and only if the two persons are directly linked to each other. So, the absence of an edge reflects the fact that the two corresponding persons do not know each other, and hence, could both be invited for Abigail's party. The resulting graph is depicted in Figure 3.3.

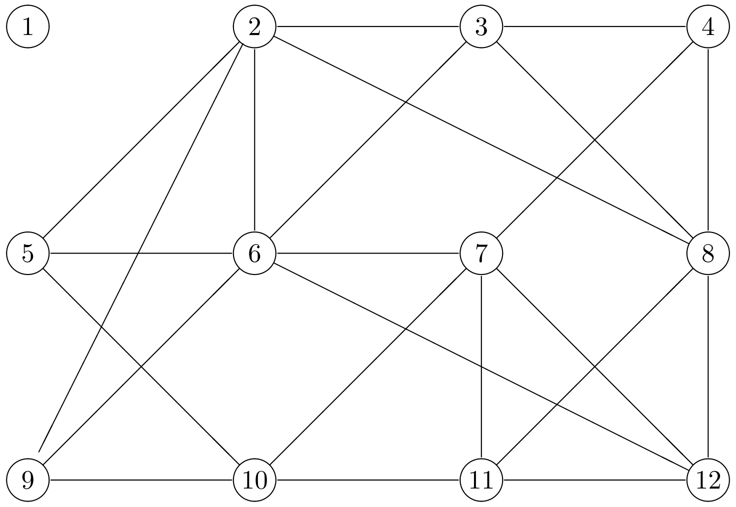


Figure 3.3: Abigail's network

(Admittedly, the names of Abigail's friends are not very revealing). From Figure 3.3 it is immediately clear that Friend No. 1 should always be invited by Abigail, since he/she is not linked to anyone of Abigail's other friends. Further, Friend No. 6 knows most of Abigail's friends so inviting her/him might not be a good idea. To avoid further puzzling, let us write down an integer programming formulation; there will be a binary variable for each friend, indicating whether (s)he is invited for dinner.

$$\begin{aligned}
\text{maximize} \quad & x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + x_9 + x_{10} + x_{11} + x_{12}, \\
\text{subject to} \quad & x_2 + x_3 \leq 1, x_2 + x_5 \leq 1, x_2 + x_6 \leq 1, x_2 + x_8 \leq 1, \\
& x_2 + x_9 \leq 1, x_3 + x_4 \leq 1, x_3 + x_6 \leq 1, x_3 + x_8 \leq 1, \\
& x_4 + x_7 \leq 1, x_4 + x_8 \leq 1, x_5 + x_6 \leq 1, x_5 + x_{10} \leq 1, \\
& x_6 + x_7 \leq 1, x_6 + x_9 \leq 1, x_6 + x_{12} \leq 1, x_7 + x_{10} \leq 1, \\
& x_7 + x_{11} \leq 1, x_7 + x_{12} \leq 1, x_8 + x_{11} \leq 1, x_8 + x_{12} \leq 1, \\
& x_9 + x_{10} \leq 1, x_{10} + x_{11} \leq 1, x_{11} + x_{12} \leq 1, \\
& x_i \in \{0, 1\} \qquad \text{for each } i = 1, \ldots, 12.
\end{aligned}$$

Again, we are not only interested in Abigail's problem; we want to be able to formulate this problem for an arbitrary person. To do so, consider an arbitrary graph $G = (V, E)$. A subset of the vertices $V' \subseteq V$ is called an *independent set* (or a *stable set*, or a *node packing*) if each edge of E is incident to at most one vertex of V'. In other words, V' is an independent set when no two vertices of an edge are both selected. The problem is to find an independent set in G containing as many vertices as possible (a maximum cardinality independent set). This is a basic problem within the field of combinatorial optimization, and called the Independent Set problem (also known as Stable Set, or as Node Packing).

In its general form, we can describe it as follows, where we define a binary variable for each node $v \in V$ as follows:

$$x_v = \begin{cases} 1 & \text{if vertex } v \text{ is selected in the independent set;} \\ 0 & \text{otherwise.} \end{cases}$$

And here is the integer program:

$$\begin{aligned} \text{maximize} \quad & \sum_{v \in V} x_v && && (3.8) \\ \text{subject to} \quad & x_v + x_w \leq 1 && \text{for all } \{v, w\} \in E, && (3.9) \\ & x_v \in \{0, 1\} && \text{for all } v \in V. && (3.10) \end{aligned}$$

3.3 Formulations and difficulty

Does the formulation of a problem tell us anything about the problem's difficulty? The short answer is no, it doesn't. Consider for instance the matching problem (Section 3.1) and the independent set problem (Section 3.2). These problems look similar, since the roles of the edges and vertices are interchanged. However, there is a striking difference between them. For the matching problem, fast and efficient algorithms that return an optimum solution exist, and have been designed and implemented. For instance, we can find an optimal solution to a matching problem on a graph with, say 10.000 nodes and 50.000 edges within seconds. In contrast, no such algorithms have been found for the independent set problem. In principle, this does not rule out the possibility that such fast algorithms exist for independent set; however, till this date, no one has ever found such an algorithm. In fact, it is widely suspected that such algorithms do not exist, but a proof of this hypothesis is lacking. All this boils down to the famous, 1 million-dollar worth, P = NP question. Thus, whereas the matching problem can be solved in *polynomial time*, the independent set problem is known to be *NP*-hard, see Chapter 4.

This crucial distinction between matching and independent set cannot be derived from viewing their IP-formulations. Therefore: *a formulation does not give an indication of the solvability of a problem.* In particular, the number of variables and/or constraints does not give a clue concerning the difficulty of a problem.

3.4 More optimization problems

3.4.1 Spanning forest

Consider an undirected graph $G = (V, E)$. A subset of the edges $E' \subseteq E$ is called a *forest* if the subgraph of G induced by E' is acyclic, see Figure 3.4 for an example. In case a forest consists of $|V| - 1$ edges it is called a *tree*. The problem is to find a maximum weight forest in G.

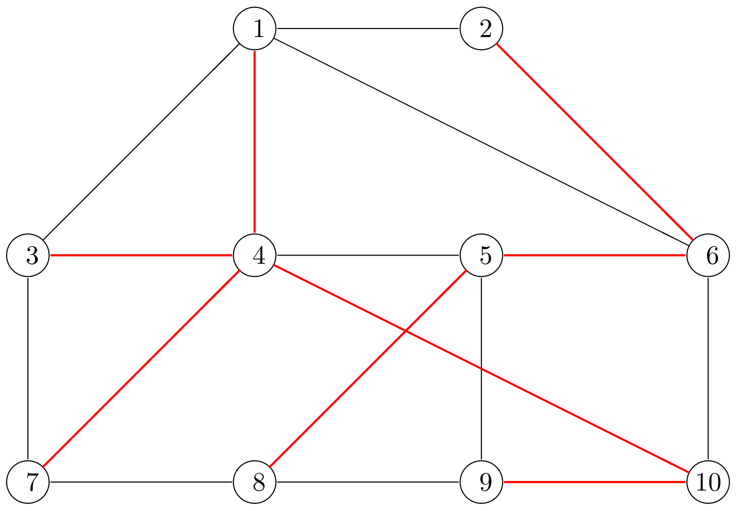


Figure 3.4: The edges in red form a forest

The problem to find a maximum weight forest can be formulated as an integer program as follows. We define a 0-1 variable for each edge $e \in E$ as follows:

$$x_e = \begin{cases} 1 & \text{if edge } e \text{ is selected in the forest;} \\ 0 & \text{otherwise.} \end{cases}$$

And here is the integer program:

$$\begin{aligned}
&\text{maximize} && \sum_{e\in E} w_e x_e && && (3.11)\\
&\text{subject to} && \sum_{e\in\delta(i)\cap\delta(j),i,j\in V'} x_e \le |V'|-1 && \text{for all } V'\subseteq V: 2\le |V'|\le |V| && (3.12)\\
& && x_e\in\{0,1\} && \text{for all } e\in E. && (3.13)
\end{aligned}$$

The constraints (3.12) limit the number of chosen edges with both endvertices in any given subset V' of the nodes to at most $|V'|-1$. Since a cycle contains as many edges as vertices, the constraints (3.12) prevent the graph (V,E') from containing cycles.

Notice that the number of subsets of V of size 2 or larger equals $2^{|V|}-|V|-1$. Thus, the number of constraints (3.12) is exponential in the size of the problem. Thus, the size of the formulation is exponential in the size of the input, although the problem is trivially solvable by a greedy algorithm.

The spanning tree problem is a variant of the spanning forest problem, where the set of feasible solutions is restricted to those acyclic subgraphs that are connected. It is well-known that for an acyclic subgraph the requirement of connectedness is equivalent to the requirement of having $|V|-1$ edges, i.e., $|E'|=|V|-1$. Thus, by adding the constraint $\sum_e x_e = n-1$ to (3.11)-(3.13), a correct formulation of the minimum weight spanning tree problem arises.

3.4.2 The Knapsack Problem

We are given a set of n items, each with a weight a_j and a value c_j for $j=1,\ldots,n$. Feasible solutions are the subsets of the set of items with cumulative weight at most b. The objective is to find a feasible solution of maximum value. The problem formulation contains binary variables x_j which indicate whether element j with $j=1,\ldots,n$ is in the knapsack:

$$x_j = \begin{cases} 1 & \text{if item } j \text{ is selected in the knapsack;}\\ 0 & \text{otherwise.}\end{cases}$$

And here is the integer program:

$$\begin{aligned}
&\text{maximize} && \sum_{j=1}^{n} c_j x_j && && (3.14)\\
&\text{subject to} && \sum_{j=1}^{n} a_j x_j \le b && && (3.15)\\
& && x_j\in\{0,1\} && \text{for all } j=1,2,\ldots,n. && (3.16)
\end{aligned}$$

In this text, we will come back extensively to the knapsack problem. There is a book devoted to this problem, see Kellerer et al. [8].

3.5 Problems allowing distinct formulations

In this section we explicitly show that a problem may admit multiple formulations. These formulations may have different properties; from our point of view, formulations that have potential for solution methods are the interesting ones. We look at the Traveling Salesman Problem (Section 3.5.1), the Uncapacitated Facility Location (Section 3.5.2), and the Bin Packing Problem (Section 3.5.3).)

3.5.1 The Traveling Salesman Problem

Probably the most well-known problem in Combinatorial Optimization is the Traveling Salesman Problem (TSP). The TSP is the prototype combinatorial optimization problem. No other problem has received as much attention as the TSP, and no other problem has captured the imagination as an easy-to-describe, yet hard-to-solve problem. A description of the problem is as follows. Given is a set of n "cities" and a distance $c_{i,j}$ between each pair of them. The goal is to find a tour of minimum length, that is to start in some city, visit each other city once, and to return to the city where the tour was started. More formally, given an $n \times n$ matrix $C = c_{ij}$, find a permutation π of $\{1, 2, \ldots, n\}$ such that $\sum_{i=1}^{n-1} c_{\pi(i),\pi(i+1)} + c_{\pi(n),\pi(1)}$ is minimum.

Different formulations of the TSP exist. Here is a conventional one, using binary variables x_{ij} indicating whether city j is visited directly after city i:

$$\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} && && (3.17)\\
\text{subject to} \quad & \sum_{i=1}^{n} x_{ij} = 1 && \text{for all } j = 1, \ldots, n && (3.18)\\
& \sum_{j=1}^{n} x_{ij} = 1 && \text{for all } i = 1, \ldots, n && (3.19)\\
& \sum_{i \in S}\sum_{j \in S} x_{ij} \le |S| - 1 && \text{for all } S \subset V, |S| \ge 2 && (3.20)\\
& x_{ij} \in \{0, 1\} && \text{for all } i, j = 1, \ldots, n. && (3.21)
\end{aligned}$$

Constraints (3.20) are called the subtour elimination constraints. An equivalent way of writing them is:

$$\sum_{i \in S}\sum_{j \notin S} x_{ij} \ge 1 \text{ for all nonempty } S \subset V.$$

Notice that this formulation has a polynomial number of variables (n^2), and an exponential number of constraints ($O(2^n)$). The latter fact might be considered a disadvantage of formulation (3.17) - (3.21).

There is, however, an alternative for this formulation which uses additional real variables u_i, one such variable for each city i. The interpretation of this variable is the position of city i in the tour, while putting the position of city 1 first. Next, by replacing constraints (3.20) by the following constraints:

$$u_i - u_j + nx_{ij} \leq n - 1 \text{ for all } i, j = 2, \ldots, n, i \neq j, \tag{3.22}$$

$$u_1 = 1, \tag{3.23}$$

we arrive at a formulation that is called the *Miller-Tucker-Zemlin (MTZ)* formulation of the TSP. It is an interesting exercise to verify the correctness of the MTZ-formulation. More concrete: why is it that (3.22) exclude subtours? The answer lies in noticing that (i) any solution satisfying (3.18), (3.19), and (3.21) consists of a collection of subtours (or a feasible solution, that is, a single tour). In case there are subtours, then there is a subtour that does not contain city 1. Let us now, for each pair of consecutive cities i and j in this subtour, sum the corresponding inequalities (3.22). The u-variables will cancel out, and the resulting lefthand side will be larger than the resulting righthand side, which means that this subtour will be forbidden by (3.22). Thus, any subtour not containing city 1 will be forbidden, and hence, the only possible solution is a single tour.

There is more than one book devoted to the TSP. We mention: Applegate et al. [2] and Lawler et al. [9].

3.5.2 Uncapacitated Facility Location

We are given a set of m facilities and n clients. Let us call the set of facilities $M \equiv \{1, 2, \ldots, m\}$, and let us call the set of clients $N \equiv \{1, 2, \ldots, n\}$. Each of the facilities can (but need not be) be opened to serve clients. Each client must be served by a facility. The cost for opening facility i is f_i, $i \in M$; the cost for serving client j by facility i is c_{ij}, $i \in M, j \in N$. This problem can be formulated in two ways with the following sets of variables, defined for each $i \in M, j \in N$:

$$x_{i,j} = \begin{cases} 1 & \text{if facility } i \text{ serves client } j; \\ 0 & \text{otherwise.} \end{cases}$$

$$y_i = \begin{cases} 1 & \text{if facility } i \text{ is open;} \\ 0 & \text{otherwise.} \end{cases}$$

The first formulation makes use of the fact that there is an upper bound on the number of clients that are served by a facility, namely the total number of clients n.

$$
\begin{aligned}
\text{minimize} \quad & \sum_{i \in M} \left(f_i y_i + \sum_{j \in N} c_{i,j} x_{i,j} \right) && && (3.24)\\
\text{subject to} \quad & \sum_{i \in M} x_{i,j} = 1 && \text{for all } j \in N && (3.25)\\
& \sum_{j \in N} x_{i,j} \leq n y_i && \text{for all } i \in M && (3.26)\\
& x_{i,j}, y_i \in \{0,1\} && \text{for all } i \in M, j \in N. && (3.27)
\end{aligned}
$$

In the second formulation, each of the constraints (3.26) is disaggregated into n new constraints, leading to constraints (3.30).

$$
\begin{aligned}
\text{minimize} \quad & \sum_{i \in M} \left(f_i y_i + \sum_{j \in N} c_{i,j} x_{i,j} \right) && && (3.28)\\
\text{subject to} \quad & \sum_{i \in M} x_{i,j} = 1 && \text{for all } j \in N && (3.29)\\
& x_{i,j} \leq y_i && \text{for all } i \in M, j \in N && (3.30)\\
& x_{i,j}, y_i \in \{0,1\} && \text{for all } i \in M, j \in N. && (3.31)
\end{aligned}
$$

Which of these formulations is preferable is not only a matter of comparing the number of constraints or variables. In fact, as will be shown in the sequel, large formulations with many constraints and/or variables are usually better from a solver's point of view. This depends on the techniques that are used to solve the problem. Since these techniques very often rely heavily on linear programming, the quality of the formulation depends almost always on the accuracy of the linear programming relaxation, i.e., the problem which results when the integrality constraints are removed. For the uncapacitated facility location problem formulation (UFL2) is better than (UFL1), since the constraints (3.30) imply the constraints (3.26).

3.5.3 The Bin Packing Problem

The Bin Packing Problem (BPP) can be described as follows. Given are n items that need to be stored in bins. The capacity of a bin equals C (this can measure volume (m^3), weight (kg), time (s), length (m), volume (m^2), ...). Each item uses a given a_i units of capacity. The problem is to assign each item to a bin such that the sum of the a_i's of items assigned to the same bin does not exceed the bin's capacity C, while using a minimum number of bins. A straightforward formulation uses, for $i = 1, \ldots, n$ and

$j = 1, \ldots, n$ binary variables

$$x_{i,j} = \begin{cases} 1 & \text{if item } i \text{ is assigned to bin } j, \\ 0 & \text{otherwise,} \end{cases}$$

and

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

The formulation is now:

$$\begin{aligned}
\text{(BPP-1)}\quad \min \quad & \sum_{j=1}^{n} y_j & & (3.32)\\
\text{s.t.} \quad & \sum_{i=1}^{n} a_i x_{i,j} \leq C y_j & \text{for } j = 1, \ldots, n \qquad & (3.33)\\
& \sum_{j=1}^{n} x_{i,j} = 1 & \text{for } i = 1, 2, \ldots, n \qquad & (3.34)\\
& x_{i,j} \in \{0,1\}, y_j \in \{0,1\} & \text{for all } i, j. \qquad & (3.35)
\end{aligned}$$

Let us now consider a formulation that uses a variable that corresponds to a set of items that may correspond to a set of items fitting in a bin. More precisely, we refer to a possible way of packing a bin as a *packing*. For each possible packing, that is, for each set of items S with $\sum_{i \in S} a_i \leq C$, we introduce a binary variable z_S indicating whether itemset S is used for a bin, or not. This allows us to write down the following formulation.

$$\begin{aligned}
\text{(BPP-2)}\quad \min \quad & \sum_{S} z_S & & (3.36)\\
\text{s.t.} \quad & \sum_{S: S \text{ contains } i} z_S = 1 & \text{for } i = 1, \ldots, n \qquad & (3.37)\\
& z_S \in \{0,1\} & \text{for all } S. \qquad & (3.38)
\end{aligned}$$

Convince yourself that constraints (3.37) imply that, for each item, a set must be chosen containing this item. There is quite a difference between formulation BPP-1 and formulation BPP-2. Of course, each of them is correct, yet the number of constraints, and in particular the number of variables differs quite a bit. Indeed, the number of variables in BPP-2 is exponentially large, and this may - at first sight - look like an insurmountable obstacle for solving it.

3.6 Combinatorial Optimization: a general formulation

In this section we (informally) argue that each combinatorial optimization problem can be formulated as an integer program. In a combinatorial optimization problem a *finite ground set* E is given. To each element $e \in E$ a weight w_e is attached. A family $\mathcal{S}$ of subsets of E is identified as the set of *feasible solutions*. This family depends on the particular problem. The weight of a set $E' \subseteq E$ is the cumulative weight of its elements, i.e., $w(E') = \sum_{e \in E'} w_e$. The associated optimization problem is to find the maximum (or minimum) weight feasible solution $E' \in \mathcal{S}$, i.e.,

$$\max_{E' \in \mathcal{S}} \{w(E')\}$$

The set of feasible solutions $\mathcal{S}$ is usually given implicitly. It is described by the properties of feasible solutions; it may be very large. For instance, in case of the matching problem, the ground set equals the set of edges, and the set $\mathcal{S}$ is the collection of edge-sets that are matchings. In case of the knapsack problem, the ground set equals the set of items, and the set $\mathcal{S}$ equals the collection of item-sets that can be put together in the knapsack.

Many examples of problems that fit in the above formulation are found in graph theory (see the first section). Among them are well-known problems like the shortest path problem, the minimum spanning tree problem, and the traveling salesman problem. The *shortest path problem* is defined as follows. In a graph $G = (V, E)$ the feasible solutions are the subsets of the edges that form paths between two specified vertices s and t. Among the paths between s and t we want to find the one with a minimum number of edges, or if a length function is given on the edges, we want to find a path of minimum total length. In the *minimum spanning tree problem* a graph $G = (V, E)$ is given together with a weight function on the edges. A feasible solution is a set of edges that forms a tree. Among the trees we want to find one with minimum weight. This problem is easily solvable by a greedy algorithm, as is well known. However, if we restrict the set of feasible solutions to trees that form paths, the problem becomes the Hamiltonian path problem, which is highly intractable. Thus, in general, problems do not become simpler when the set of feasible solutions is reduced.

To formulate a combinatorial optimization problem in mathematical terms, we introduce decision variables for all elements of the ground set E. Each decision variable denotes a choice, namely whether the corresponding element is chosen or not. So, a variable can have two values, which are usually taken from $\{0, 1\}$. A set $E' \subseteq E$ can be described by a binary vector $x_{E'} = (x_e)_{e \in E}$ with $n = |E|$ components as follows:

$$x_e = \begin{cases} 1 & \text{if element } e \text{ is in } E'; \\ 0 & \text{otherwise.} \end{cases}$$

The finite set of vectors $X \subseteq \mathbb{R}^n$ corresponding to feasible solutions from $\mathcal{S}$ can then be described by means of constraints. In many cases, these constraints are linear and involve binary variables. The objective function w is usually a linear function of the components of $x \in X$. The problem is then

$$\max\{wx \mid Ax \leq b \ , \ x \in \{0,1\}^n\}$$

where A and b depend on the problem at hand. This type of formulation is often called an *Integer (or Binary) Linear Program*, (ILP).

Up to now, we have introduced binary variables to model decisions of the yes-no type, more specifically, to decide whether an element is selected or not. These variables are used further to describe the constraints that determine the feasible solutions. In all the examples, these constraints can be written as linear functions with a bound imposed on them. Similarly, the objective can be formulated as a linear function of the variables. We point out that we can generalize the formulations of combinatorial problems with respect to all three items, i.e., the variables, the constraints, and the objective.

Clearly, the decisions in combinatorial problems may be more complicated than simple yes/no decisions. One may have to introduce integer variables or even real variables, like in linear programming, to model certain decisions. And, of course, a problem formulation may contain both types of variables (*mixed* integer programming). The combinatorial nature (a finite (or countable) number of feasible solutions) may not be so evident in these problems. However, the number of "interesting" feasible solutions is usually still finite in such problems. For instance, in linear programming, the interesting feasible solutions are the vertices of a polyhedron. The number of vertices is usually finite.

One may consider any function of the variables with a bound imposed on it as a constraint. Moreover, logical compositions of constraints, like implications and disjunctions (logical "or") can be used to model the restrictions of a problem. In abstracto, any relation on the variables that restricts the set of feasible solutions can be used as a constraint.

The objective of a problem can be any function of the variables, thus it is not restricted to linear functions.

The above observations lead to the following abstract formulation of optimization problems. We distinguish between the real variables, denoted by the vector x, and the integral variables, denoted by the vector y. A formulation of a combinatorial optimization problem contains the following items:

- a vector of n real decision variables:

$$x = (x_1, \dots, x_n)$$

- a vector of m integral decision variables:

$$y = (y_1, \dots, y_m)$$

- a set C of k constraints:

$$C = \{C_1, \dots, C_k\}$$

- an objective function on the variables:

$$f(x; y).$$

The set of *feasible solutions* consists of vectors $(x; y)$ that satisfy all the constraints. Each variable has a *domain* D, usually the set of real numbers $\mathbb{R}$, or the set of integer $\mathbb{N}$. The solution space of the problem is the Cartesian product of the domains of the variables, i.e., $\mathbb{R}^n \times \mathbb{N}^m$.

For many solution techniques, especially the ones that we are going to discuss, it is of eminent importance to restrict the domains of the variables as much as possible. Some constraints imply lower and upper bounds on the value of a variable, directly or indirectly. For instance, binary variables have an explicit lower and upper bound. If this is the case, we usually take this into account in the problem formulation explicitly, i.e., if a real variable x_i has a lower bound l_i and an upper bound u_i, then we describe its domain as $[l_i, u_i]$; if y_j is an integral variable, with a lower bound l_i and an upper bound u_i, then we describe its domain as $\{l_j, \dots, u_j\}$.

3.7 Pitfalls

A mathematical programming formulation of some optimization problem consists of two types of symbols: (i) those symbols that represent *parameters*, (ii) those symbols that represent *variables*. Before stating the model, one should clearly define the meaning of each symbol that is used in the model and mention its type (parameter or variable). A parameter is a symbol whose value is known, for instance the symbol a_i in constraints (3.33) are known, and hence can be classified as parameters. Variables are symbols whose value we do not know; in fact, solving the model will hopefully give us the optimum values for these symbols. For instance, the $x_{i,j}$ symbols in (3.32)-(3.35) are clearly variables. There are some "popular" mistakes when specifying a model:

- a constraint cannot consist of parameters only,
- when using indices, make sure that the value of each index is specified,
- when variables are multiplied, alert the reader that the model is not linear.

3.8 Satisfiability (SAT)

In the previous sections, we have formulated many discrete optimization problems as an integer program. Like integer programming, satisfiability (SAT) is also a strong "language" to describe combinatorial problems. In an instance of a satisfiability problem, we are given a set of n variables, say $\{x_1, \ldots, x_n\}$, and a set of m clauses C over the literals $\{x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n\}$. Each variable x_i $(1 \leq i \leq n)$ can be set to either TRUE or FALSE, and the problem is to decide whether there exists a truth assignment such that, in each clause, at least one of its literals is true. Notice that, in contrast to integer programming, this problem is phrased as a question having a yes/no answer, whereas integer programming is naturally formulated as an optimization problem.

Example 3.8.1 *Here is an instance of SAT with $n = 4$ variables, and $m = 6$ clauses:*

$$C = (x_1 \vee \bar{x}_2 \vee \bar{x}_3 \vee x_4) \wedge (\bar{x}_1 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_4) \wedge (x_3) \wedge (x_2 \vee x_4) \wedge (\bar{x}_3 \vee \bar{x}_4).$$

Question: does there exist a satisfying truth assignment?

As shown by the example above, an instance of SAT is, in fact a logical formula in a particular form. This form can be described as a conjunction of clauses, where each clause is a disjunction of terms; in other words, the form is an AND of OR's. This particular form is called *conjunctive normal form*, or CNF.

3.8.1 Three-satisfiability (3-SAT)

A (seemingly) special case of SAT arises when we restrict the number of literals in each clause. Indeed, by having at most three literals in each clause, a problem known as 3-SAT arises.

It is in fact possible to rephrase an arbitrary instance of SAT with n variables, and m clauses (each of arbitrary length), into a version that uses n' variables and contains m' clauses, each with no more than three literals. Here n', m' are polynomially bounded in the numbers n and m which means that the rewriting of the problem yields a problem of comparable size. From this, it actually follows that even 3-satisfiability is apparently not so easy.

Given an instance C of SAT, we will build a new instance C' of 3-SAT such that (i) each clause in C' contains at most three literals, and (ii) C' is satisfiable if and only if C is satisfiable.

Let C_j be an arbitrary clause in a given instance of SAT, i.e., $C_j = (\ell_{j,1} \vee \ldots \vee \ell_{j,k})$, $k \geq 4$. Introduce $(k-3)$ new variables $y_1, \ldots, y_{k-3}$. Replace the clause $C_j = (\ell_{j,1} \vee \ldots \vee \ell_{j,k})$ by the following set of $k-2$ clauses:

$$(\ell_{j,1} \vee \ell_{j,2} \vee y_1) \wedge (\bar{y}_1 \vee \ell_{j,3} \vee y_2) \wedge \ldots \wedge (\bar{y}_{t-2} \vee \ell_{j,t} \vee y_{t-1}) \wedge \ldots \wedge (\bar{y}_{k-4} \vee \ell_{j,k-2} \vee y_{k-3}) \wedge (\bar{y}_{k-3} \vee \ell_{j,k-1} \vee \ell_{j,k}).$$

We refer to this set of clauses as C'_j.

We leave it to the reader to verify that if the clause C_j is satisfiable, there exists a truth assignment for the variables y_t $(1 \leq t \leq k-3)$ such that each clause in C'_j is satisfiable; moreover, the opposite direction is true as well, that is, if each clause in C'_j is satisfied, there must be an $\ell_{j,t}$ set to true.

Also observe that a single clause containing k terms is replaced by $k-2$ clauses containing, in total, $3(k-2)$ terms.

3.8.2 Two-satisfiability (2-SAT)

In this section we describe an integer formulation of the satisfiability problem restricted to formulations with clauses containing at most two literals (also known as 2-SAT). So here we have a number of clauses of the form $C_j = (x_p \vee y_q)$ or $C_j = (x_p \vee \bar{y}_q)$ or $C_j = (\bar{x}_p \vee y_q)$ or $C_j = (\bar{x}_p \vee \bar{y}_q)$. Here x_p and y_q refer to variables that can take the value true or false. The boolean setting of the variables should be such that all clauses evaluate true. This can easily be written in the form of an integer linear program, in which variables take value 1 (representing a true setting) or 0 (representing a false setting). The condition that a clause C_j be true is then easily written as:

$x_p + y_q \geq 1$, $x_p + (1 - y_q) \geq 1$, $(1 - x_p) + y_q \geq 1$, $(1 - x_p) + (1 - y_q) \geq 1$, respectively; or, when moving the constants to the right and using only $\leq$ constraints, $-x_p - y_q \leq -1$, $-x_p + y_q \leq 0$, $+x_p - y_q \leq 0$, $+x_p + y_q \leq +1$, respectively. We then ask ourselves, does there exist a binary vector such that a system $Ax \leq b$ has a solution? We will show later in the section how this IP formulation can be solved in a reasonable number of elementary steps.

Fourier-Motzkin elimination

We first study an approach to solve a linear system of inequalities in the real numbers: given an $m \times n$ matrix $A = (a_{i,j})$, a vector $b \in \mathbb{R}^m$, does there exists an $x \in \mathbb{R}^n$ such that $Ax \leq b$? The rationale behind the method is to eliminate one variable at a time, modifying the inequality system so that the resulting system has a solution if and only if the system with one more variable had one.

Let us illustrate how to eliminate variable x_1. Consider the given system and partition the set of row indices I into $I = I_0 \cup I_+ \cup I_-$, where row $i \in I_\sigma$ if and only if $\mathrm{sgn}(a_{i1}) = \sigma$, where $\sigma \in \{0, +, -\}$. Partition the vector x into $x^T = (x_1\, \hat{x}^T)$, and partition the ith row of A into $a_i = (a_{i1}\, \hat{a}_i)$. Observe that a vector x satisfies $Ax \le b$ if and only if

$$\begin{array}{rll} & \hat{a}_i\hat{x} \le b_i & \text{for } i \in I_0, \\ a_{i1}x_1 + & \hat{a}_i\hat{x} \le b_i & \text{for } i \in I_+, \\ -a_{i1}x_1 - & \hat{a}_i\hat{x} \ge -b_i & \text{for } i \in I_-, \end{array}$$

that is, if and only if

$$\begin{array}{rll} & \hat{a}_i\hat{x} \le b_i & \text{for } i \in I_0, \\ x_1 \le & \frac{1}{a_{i1}}\left(-\hat{a}_i\hat{x} + b_i\right) & \text{for } i \in I_+, \\ x_1 \ge & \frac{1}{-a_{i1}}\left(\hat{a}_i\hat{x} - b_i\right) & \text{for } i \in I_-. \end{array}$$

Observe that the latter two sets of inequalities can be rephrased as:

$$\frac{1}{-a_{k1}}\left(\hat{a}_k\hat{x} - b_k\right) \le x_1 \le \frac{1}{a_{i1}}\left(-\hat{a}_i\hat{x} + b_i\right) \quad \text{for } i \in I_+, k \in I_-.$$

This implies that the system $Ax \le b$ has a solution if and only if the system $A'\hat{x} \le b'$ has a solution where the latter system is given by

$$\begin{array}{rll} & \hat{a}_i\hat{x} \le b_i & \text{for } i \in I_0, \\ \frac{1}{-a_{k1}}\left(\hat{a}_k\hat{x} - b_k\right) \le & \frac{1}{a_{i1}}\left(-\hat{a}_i\hat{x} + b_i\right) & \text{for } i \in I_+, k \in I_-. \end{array}$$

By applying this process, we eliminate a variable. And after having applied this process n times, we have arrived at a system where we only need to compare a number of constants in order to decide whether the original system had a solution. Of course, one should realize that while the number of variables may be decreasing, the number of constraints is increasing, and in fact is increasing rapidly. The system may end up having $4\left(\frac{m}{4}\right)^{2^n}$ rows.

Solving 2-SAT by integer programming

Consider the MIP formulation of the 2-satisfiability problem as a system $0 \le x \le 1$, $Ax \le b$. Observe that the constraints corresponding to the clauses, represented by $Ax \le b$, have the property that for all i: $\sum_j |a_{ij}| = 2$. Moreover all coefficients a_{ij} and b_i are integer. Finally, we realize that we seek an **integer** solution x. The method sketched above only works for $x \in \mathbb{R}^n$.

So now we formulate a modification of the above procedure so as to establish whether a system $Mx \leq d$, with integer coefficients, m_{ij} and d_i, and with the property $\sum_j |m_{ij}| \leq 2$ has an integer solution x or not.

Let $(\mathcal{M}^0, \delta^0) = (M, d)$ denote the original system;

```
**For** $i = 1, \ldots, n$ **do**
    (0) Set $(\mathcal{M}, \delta) = (\mathcal{M}^{i-1}, \delta^{i-1})$.
    (1) Replace in $(\mathcal{M}, \delta)$ any constraint of the form $\ell \leq 2x_i \leq u$, by: $\lceil \frac{\ell}{2} \rceil \leq x_i \leq \lfloor \frac{u}{2} \rfloor$. In case the highest lower bound on $x_i$ exceeds the lowest upper bound on $x_i$: return("system has no solution") and abort further processing.
    (2) Eliminate variable $x_i$ as in the linear case, yielding the system $(\mathcal{M}', \delta')$. In case the system contains a constraint $0 \leq \beta$, with $\beta < 0$, return("system has no solution") and abort further processing. This may happen for example when combining $x_j + 1 \leq x_i \leq x_j - 1$.
    (3) For each pair $j, k$, $j < k$, and each signature $\alpha = \pm 1, \beta = \pm 1$, consider all constraints $\alpha x_j + \beta x_k \leq \gamma_\ell$, and keep only the one with lowest value $\gamma_\ell$.
    (4) For each variable $x_j$, and each coefficent $C \in \{-2, -1, 1, 2\}$, consider all bounds $Cx_j \leq \delta_\ell$, and only keep the ones with lowest right hand side. Let the resulting system be denoted by $(\mathcal{M}'', \delta'')$.
    (5) Set $(\mathcal{M}^i, \delta^i) = (\mathcal{M}'', \delta'')$.
**end for**
```

Observe that after iteration i the set of constraints in the incumbent system $(\mathcal{M}^i, \delta^i)$ contains never more than $4n$ single variable bounds, and $4n(n-1)/2$ constraints on pairs of variables. Each elimination in step (2) creates – from two constraints involving x_i – a constraint with two variables, with coefficients ± 1, or, a constraint with a single variable and a coefficient in $\{-2, -1, 1, 2\}$, or a constraint $(0 \leq \beta)$ with no variables. At all times the right hand side coefficients are integer, as we only divide by ± 1. The sets of integer solutions to system $(\mathcal{M}, \delta)$, before and after step (1) are the same. Also the set of solutions to systems $(\mathcal{M}', \delta')$ and $(\mathcal{M}'', \delta'')$ are also the same. Any solution to $(\mathcal{M}, \delta)$ is also a solution to $(\mathcal{M}', \delta')$, whereas a solution to $(\mathcal{M}', \delta')$ can always be extended to a solution to $(\mathcal{M}, \delta)$.

If the original satisfiability problem has a solution, then so has its MIP formulation, and the same solution satisfies each system $(\mathcal{M}^i, \delta^i)$. Therefore the procedure cannot be aborted in step (1) or (2).

If in the final system $(\mathcal{M}^n, \delta^n)$ all right hand sides are non-negative, then in the nth iteration, step (2), all lower bounds on x_n in $(\mathcal{M}^{n-1}, \delta^{n-1})$ (as observed in the construction of $(\mathcal{M}', \delta')$), are integer and lower than the lowest upper bound on x_n. Choose $x_n = \hat{x}_n :=$ highest lower bound. Then $\hat{x} = (\hat{x}_n)$ is a solution to $(\mathcal{M}^{n-1}, \delta^{n-1})$. We proceed by induction to produce a solution of the original system. Suppose, we are given a solution $\hat{x} = (\hat{x}_i, \hat{x}_{i+1}, \ldots, \hat{x}_n)$ to system $(\mathcal{M}^{i-1}, \delta^{i-1})$. That is, by construction, also a solution of the system $(\mathcal{M}', \delta')$ in round $i-1$. In the construction of $(\mathcal{M}', \delta')$ variable x_{i-1} was sandwiched between lower and upper bounds in terms of variables $x_i, \ldots, x_n$, and as we have a solution $\hat{x}$, this means that the highest of lower bounds is at most the lowest of the upper bounds, in terms of values $\hat{x}_i, \ldots, \hat{x}_n$. So taking $x_{i-1} = \hat{x}_{i-1} :=$ highest of lower bounds results in a solution $(\hat{x}_i, \hat{x})$ to the system $(\mathcal{M}^{i-2}, \delta^{i-2})$.

3.9 Exercises

Exercise 1

Consider the stable set problem on an undirected graph $G = (V, E)$. Show that the formulation (3.8)-(3.10) is a correct formulation of the stable set problem.

Exercise 2

Consider the matching depicted in Figure 3.2. Is it maximal? Is it maximum? Can you find a minimum maximal matching, i.e., a maximal matching consisting of as few edges as possible?

Exercise 3

Consider the graph depicted in Figure 3.3. Find a maximum matching (by whatever means), and prove that it is optimum. Answer the same question for independent set.

Exercise 4

Consider the formulation of the matching problem. Give the dual of the corresponding linear programming relaxation.

Exercise 5

Consider the formulation of the independent set problem. Give the dual of the corresponding linear programming relaxation.

Exercise 6

Consider an undirected graph $G = (V, E)$. A edge cover E' is a subset of the edges such that each node is incident to at least one edge in E'. Formulate the problem of finding a minimum cardinality edge cover as an integer linear program.

Exercise 7

Consider an undirected graph $G = (V, E)$. A node cover V' is a subset of the nodes such that each edge is incident to at least one node in V'. Formulate the problem of finding a minimum cardinality node cover as an integer linear program.

Exercise 8

A clique partitioning of an undirected, complete graph G is a partitioning of the vertices into subsets $V_1, V_2, \ldots, V_k$ such that the subgraph induced by each V_i is a complete graph itself ($i = 1, \ldots, k$). Consider now a complete graph G and an arbitrary weight w_{ij} for each edge of the graph (notice that w_{ij} can be negative; in fact, the problem is only interesting when there are both positive and negative edge weights). The objective is to find a clique partitioning of maximal weight, that is, a clique partitioning such that the cumulative weight of the edges that have both vertices in one and the same component is maximum. Formulate this problem as an integer linear programming problem. (Hint: use a variable for each edge).

Exercise 9

Consider the TSP.

- When the distance matrix C is known to be symmetric, can you simplify formulation (3.17) - (3.21) by "merging" constraints (3.18) and (3.19)?
- Which of the two formulations given in Section 3.5.1 is stronger?

Exercise 10

Given is a set of axis-aligned rectangles in the plane. Each of these rectangles needs to be stabbed, either by a horizontal line or by a vertical line. The problem is to find a set horizontal and vertical lines, stabbing each rectangle at least once, with minimum cardinality. We call this the *rectangle stabbing problem*. Formulate this problem as an integer linear programming problem. (Hint: first, give a formulation for the instance depicted in Figure 3.5).

Exercise 11

Given an instance of the rectangle stabbing problem as depicted in Figure 3.5, what fractional solution is the optimal linear programming relaxation (of the integer program that you just wrote down) for this instance?

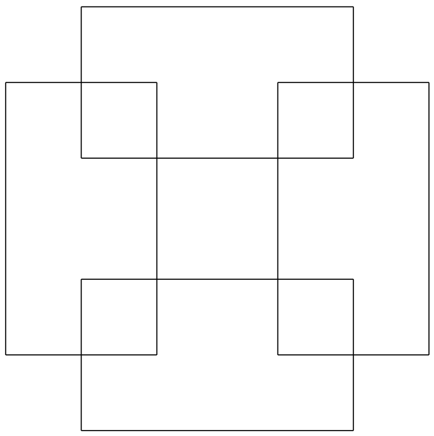


Figure 3.5: A rectangle stabbing instance

Chapter 4

A glimpse at computational complexity

In order to fully appreciate the field of combinatorial optimization, it is necessary to understand, at least at an intuitive level, some of the basic concepts of computational complexity. Indeed, without an explicit notion of time, combinatorial optimization as an interesting research field ceases to exist. Computational complexity is regarded as a field within theoretical computer science dealing with fundamental, and extremely deep questions like: "what tasks can be carried out by a computer?", or "how much time does a given computational task require by any algorithm?"

In this chapter, we attempt to introduce some elements of computational complexity, in an informal and hand-waving way. There are many books devoted to this topic; we mention a classic Garey and Johnson [6].

4.1 Computational performance criteria

What do we expect from an algorithm for a combinatorial optimization problem? Well, one possible answer would be that this algorithm should always return an optimal solution of the problem; by the way, an algorithm that always returns an optimal solution is called *exact*. Is that the only game in town? Certainly not. Many situations are conceivable where we want to trade-off exactness for speed; more concrete, finding an approximate solution fast (i.e., within seconds) can be hugely preferred over finding an optimum in hours (or worse). Thus, some notion of time is required; more concrete: we might want an algorithm to be fast or efficient. Combining these two expectations is a crucial thing. Of course, the time required to solve a problem increases with the *size* of this problem, where the size can be measured by the amount of data needed to describe a particular instance of the problem (see Section 4.2).

Let us take a look at an example. Suppose that we want to solve a 0-1 linear programming problem involving n variables $x_j \in \{0, 1\}$, $j = 1, \ldots, n$. We can certainly find an optimal solution by listing all possible vectors $(x_1, x_2, \ldots, x_n)$, checking for each of them whether it is feasible or not, computing the value of the objective function for each such feasible solution, and, finally, retaining the best solution found in the process. If we decide to go that way, then we must consider 2^n vectors. For $n = 50$, that means $2^{50} \approx 10^{15} = 1,000,000,000,000,000$ vectors! If our algorithm is able to enumerate one million (1,000,000) solutions per second, the whole procedure takes 10^9 seconds, or about 30 years. And for $n = 60$, the enumeration of the 2^{60} solutions would take about 30000 years!!

Notice that *adding* 10 variables to the problem increases the computing time by a *multiplicative factor* of $2^{10} \approx 1000$. So, with $n = 80$ variables (a rather modest problem size), the same algorithm would run for 30 billion years, which is about twice the age of the universe. Not really efficient, by any practical standards...

Let us look at this issue from another vantage point. Consider the well-known Moore's law: Gordon Moore, co-founder of the chips giant Intel, prophetized in 1965 that the number of transistors per square inch on integrated circuits would double every 18 months per year starting from 1962, the year the integrated circuit was invented (see the original paper for more details). In other words, your PC processor works twice faster every year and a half, meaning that its speed is multiplied by 100 in 10 years.[1] So, if you were able to enumerate 2^n solutions in one hour in 1993, you can enumerate $100 * 2^n < 2^{n+7}$ solutions in 2003. This great increase in computing speed thus allows you to "gain" only 7 variables in 10 years !!

Conclusion: exhaustive enumeration is not feasible in practice for large-scale (or even medium-scale) combinatorial optimization problems. Furthermore, we should not count on technical progress alone to improve the situation in any significant way. Only algorithmic, or mathematical, advances can help in this respect.

So, how much progress can we expect on the theoretical front? Before we provide a tentative answer to this question, let us try to formulate more precisely some of the notions that have just been sketched.

[1]This rate has slowed down since Moore made his claim. It is now generally admitted that the number of transistors doubles every two years.

4.2 About problems, instances, and algorithms

Formally speaking, a *(computational) problem* is a generic question whose formulation includes a number of undetermined parameters. Here are some simple examples.

1. Matrix addition problem: The parameters are n, A and B where $n \in N$, and A and B are two $n \times n$ matrices. Question: find $A + B$.

2. Shortest path problem: The parameters are a graph $G = (V, E)$, two vertices $s, t \in V$, and the length $\ell(e) \geq 0$ of every edge $e \in E$. Question: find a shortest path from s to t.

3. Travelling salesman problem: The parameters are a graph $G = (V, E)$, and the length $\ell(e) \geq 0$ of every edge $e \in E$. Question: find a shortest travelling salesman tour in G.

4. Satisfiability: The parameters are n, m, a set of variables $\{u_1, \ldots, u_n\}$ and a set of m clauses C over the literals $\{u_1, \bar{u}_1, u_2, \bar{u}_2, \ldots, u_n, \bar{u}_n\}$. Question: does there exist a truth assignment for the variables such that, in each clause, at least one of its literals is TRUE?

5. Hamiltonicity: The parameter is a graph $G = (V, E)$. Question: does there exist a Hamiltonian cycle in G?

An *instance* of a problem Q arises when the values of all undetermined parameters of Q are specified (or, more intuitively, by specifying the input file that contains the numerical data). So, a problem can also be viewed as a collection of instances. Notice that an instance admits an answer, but a problem does not (try to give the answer of the matrix addition problem above!). We will use the symbol I for an instance.

The *size* of an instance I, denoted by $s(I)$, is the number of bits needed to represent I. It is determined both by the number of parameters and by their magnitude. (Intuitively, this can be viewed as the size of the input file of a computer program which solves I.)

An *algorithm* for a problem Q is a step-by-step procedure that describes how to compute a solution for every instance I of Q. To compare the efficiency of different algorithms for a same problem Q, we can determine the time required by each algorithm to solve an instance of Q. Notice that this obviously depends on the particular instance which is to be solved, but also on the speed of the computer, on the skills of the programmer, etc. Therefore, we need again to define this notion in more formal way.

The *running time* of an algorithm A on an instance I, denoted by $t_A(I)$, is the number of elementary operations (additions, multiplications, comparisons, ...) performed by A when it runs on the instance I.

Determining the running time of an algorithm for each particular instance I is not an easy task. However, it is often easier to estimate the running time as a function of the *size* of the instance.

Consider again the examples defined above.

1. Matrix addition problem:

   Instance size: $\approx 2n^2$.

   Algorithm: any naive addition algorithm.

   Running time: $\approx n^2$ (additions). We denote this by $O(n^2)$, meaning that the running time grows at most like n^2.

2. Shortest path problem:

   Instance size: $O(n^2)$ where $n = |V|$.

   Algorithm 1: enumerate all possible paths between s and t.

   Running time of Algorithm 1: there can be $O(n!)$ paths, leading to $O(n!)$.

   Algorithm 2: Dijkstra's algorithm.

   Running time: $O(n^2)$ operations.

3. Travelling salesman problem:

   Instance size: $O(n^2)$ where $n = |V|$.

   Algorithm: enumerate all possible tours.

   Running time: $O(n!)$.

4. Satisfiability:

   Instance size: $O(n \times m)$.

   Algorithm: enumerate all possible truth assignments.

   Running time: $O(2^n)$.

5. Hamiltonicity:

   Instance size: $O(n^2)$ where $n = |V|$.

   Algorithm: enumerate all possible Hamiltonian cycles.

   Running time: $O(n!)$.

Notice that, in all these examples, we chose to ignore the size of a number, that is, we did not take into account the number of bits needed to store some number - we come back to this issue in Section 4.4.

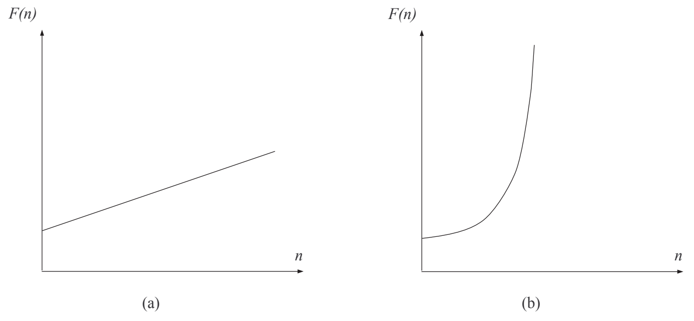


Figure 4.1: (a) Linear: $F(n) = an + b$; (b) Exponential: $F(n) = a\,2^n$

In view of these examples, we are led to the following concept: the *complexity* of an algorithm A for a problem Q is given by the function

$$F_A(n) = \max\{t_A(I) \mid I \text{ is an instance of } Q \text{ with size } s(I) = n\}. \tag{4.1}$$

This function is sometimes called the "worst-case complexity" of A: indeed, the definition focuses on the worst-case running time of A on an instance of size n, rather than on its average running time.

Figure 4.1 represents different types of complexity behaviors for algorithms.

An algorithm A is called *polynomial* if $F_A(n)$ is a polynomial function (or is bounded by a polynomial) in n and *exponential* if $F_A(n)$ grows faster than any polynomial function in n. Intuitively, we can probably accept the idea that a polynomial algorithm is more efficient than an exponential one.

For instance, the obvious algorithms for the addition and/or the multiplication of matrices are polynomial. So is the Gaussian elimination algorithm for the solution of systems of linear equations. On the other hand, the simplex method (or at least, some variants of it) for linear programming problems is known to be exponential while interior point methods are polynomial. This clearly illustrates the emphasis on the "worst-case" running time which was already underlined above: indeed, in an average sense, the simplex algorithm is an efficient method.

The complete enumeration approach for shortest path, matching, stable set or travelling salesman problems is exponential, since all these problems have an exponential number of feasible solutions. But polynomial algorithms exist for the shortest path problem and the matching problem.

For stable set, the travelling salesman problem, or for 0-1 integer programming problems, by contrast, only exponential algorithms are known.

Summarizing this discussion, the complexity of an algorithm is related to the hardness of a problem. In particular, if, for some problem Q, there exists an exact algorithm with polynomial complexity, the problem is said to be solvable in polynomial time, or *efficiently solvable*. Further, the problem resides in the complexity class P; we will explore hardness of problems, and complexity classes in more depth in the next section.

4.3 The class NP

In this section we restrict our discussion to so-called *decision problems*. Decision problems are problems that admit a yes/no answer only. For instance, both Hamiltonicity and Satisfiability (see above) are each decision problems. Or: given an edge-weighted graph $G = (V, E)$ with a source s and a sink t, does there exist a path between s and t with length no more than 100? is a decision problem. Although optimization problems are different from decision problems, it is clear that an algorithm that solves a decision problem can be used to solve the corresponding optimization problem (and vice versa).

Let us consider the decision problem Hamiltonicity. Clearly, when given a graph, it is Hamiltonian, or it is not. Suppose that for some particular complicated graph you and your boss disagree, i.e., you think the graph is Hamiltonian, she thinks it is not. How are you going to convince that the graph is Hamiltonian, i.e., that this particular graph represents a yes-instance? Well, actually, there is a very short way to end the discussion in your favor: producing the cycle! When your boss faces the cycle, she will have no choice but accepting that the graph is indeed Hamiltonian. In other words, the cycle you produced acts as a *certificate* for showing that the instance is a yes-instance. Notice that the amount of work she needs to perform when you confront her with the cycle is quite limited: it consists of checking whether the cycle you propose is indeed a cycle. Hence, this amount of work is polynomial in the size of the input.

How is your boss going to convince you that the graph is non-Hamiltonian? That is not so easy, at least, there does not seem to exist a short certificate proving that the graph is not Hamiltonian. This asymmetry between yes-instances and no-instances is a fundamental distinction. Moreover, the presence of a short certificate (i.e., one computable in polynomial time) proving that an instance is a yes-instance, implies that Hamiltonicity is in the class NP.

More generally, NP is the class of decision problems for which a certificate proving that an instance is a yes-instance, can be verified in polynomial time. Almost all decision problems we have come across in discrete optimization are in the class NP. We mention a few examples: for Knapsack, verifying that a given subset of items satisfies the knapsack constraint, and attains a given profit is indeed doable in polynomial time. For Matching, verifying that a given set of edges constitutes a matching is an easy exercise. Also, Hamiltonicity of a graph is in NP since it is easily verified whether a given set of edges constitutes a Hamiltonian cycle. However, certainly not all decision problems are in NP: what about the question: given a graph, is it non-Hamiltonian? Indeed, there does not seem to be an easy way of convincing someone of the non-Hamiltonicity of a graph. There is actually a complexity class for this type of questions: co-NP - suffice it to say that co-NP consists of the class of decision problems for which a easily verifiable certificate exists showing that the instance is a no-instance; we will not pursue this topic here further.

A deep result, attributed to Cook, concerns the existence of a hardest problem in the class NP. He proved that Satisfiability is a hardest problem in the class NP. Any decision problem in the class NP can be formulated as an instance of Satisfiability.

Theorem 4.3.1 *(Cook) Satisfiability is a hardest problem in NP.*

This is a powerful, and almost breathtaking result. It has given us a tool to prove, for many many other problems that they are also a hardest problem in the class NP. Incidentally, a problem that is "a hardest problem in the class NP" will be called NP-complete. Knowing that Satisfiability is NP-complete opens up a proof technique, where we reduce some other problem to Satisfiability implying that this problem is at least as hard as Satisfiability, and hence also NP-complete. Take for instance 0-1 Integer Programming, formulated as follows:

- 0-1 Integer Programming (IP): Given n binary variables, and a set of linear inequalities in these variables. Question: does there exist a feasible solution?

Theorem 4.3.2 *If there is a polynomial time algorithm for IP, then we have a polynomial time algorithm for Satisfiability.*

Proof Sketch: First, we build, given an arbitrary instance of Satisfiability, an instance of IP. Then, we show equivalence between yes-instances of the two problems.

Given an instance of Satisfiability, we create a binary variable in IP for every variable present in Satisfiability. Thus, the number of variables in IP equals the number of variables in the instance of Satisfiability. Further, every clause in the instance of Satisfiability gives rise to a linear constraint in the binary variables. Thus, the number of constraints in IP equals the number of clauses in the instance of Satisfiability. An individual constraint simply consists of the sum of the variables corresponding to the literals present in the clause, where a positive literal (u_i) is represented by x_i, and a negative literal ($\bar{u}_i$) is represented by $1 - x_i$. The right hand side of each constraint equals 1, the operator is "$\geq$". This completes the instance of IP.

We claim that (i) if the instance of Satisfiability is a yes-instance, then there exist a feasible solution to the IP, and (ii) if there exists a feasible solution to the IP, then there exists a satisfying truth assignment.

Ad (i) If the instance of Satisfiability is a yes-instance, there exists a truth assignment to the variables such that in each clause at least one of its literals is true. We propose to set $x_i := 1$ if, in this truth assignment, variable u_i is set to TRUE; otherwise, we set $x_i := 0$. Since, by assumption, in each clause, there is at least one true literal, it follows that the left-hand side of the corresponding constraint must feature a "1". Hence, each constraint is indeed satisfied, and the IP has a feasible solution.

Ad (ii) Suppose the IP admits a feasible solution. If, in this feasible solution, a binary variable $x_i = 1$, we set the corresponding variable u_i to TRUE; otherwise, we set the variable to FALSE. It is not difficult to observe that feasibility of the instance of IP necessarily leads to the instance of Satisfiability being satisfiable.

□

As an example of the argument above, consider this instance of Satisfiability. We have $n = 3$ variables $\{u_1, u_2, u_3\}$, and we have $m = 4$ clauses:

$$(u_1 \vee \bar{u}_2 \vee \bar{u}_3) \wedge (u_1 \vee \bar{u}_3) \wedge (\bar{u}_2 \vee u_3) \wedge (\bar{u}_1 \vee u_2 \vee \bar{u}_3).$$

The corresponding instance of IP consists of the following set of 4 inequalities in 3 binary variables:

$$\begin{aligned} x_1 + (1 - x_2) + (1 - x_3) &\geq 1 \\ x_1 + (1 - x_3) &\geq 1 \\ (1 - x_2) + x_3 &\geq 1 \\ (1 - x_1) + x_2 + (1 - x_3) &\geq 1. \end{aligned}$$

We have now argued that the existence of a polynomial time algorithm for solving IP, i.e., an algorithm that correctly tells you whether the given instance is feasible or not, can be used to decide Satisfiability. Simply "translate" the instance of Satisfiability to IP, and run your algorithm. And since the Satisfiability is a hardest problem in NP (NP-complete), and since IP is in the class NP, IP must be NP-complete as well.

Corollary: IP is NP-complete.

We have seen here a trick (to which we refer as a *reduction*) that has turned out to be enormously powerful. The equivalence between the yes-instances of the two problems involved in a reduction is a crucial ingredient of any correct reduction. Reductions have been used many many times in order to prove that some decision problem is NP-complete. In fact, for almost any decision problem in the class NP, it is known whether it is in the class P, or whether it is NP-complete; only a handful (famous) problems resist classification. Reductions have connected many distinct problems, and have yielded information concerning the difficulty of decision problems.

Let us give some other examples of this trick.

Consider the following problem called *Makespan on two machines*, or MS2 for short. Its decision variant can be phrased as follows: given n jobs, each with a processing time (or length) p_j, two machines, and an integer B. The question is: does there exist a schedule (i.e., an assignment of each job to one of the two machines) such that the makespan (the latest completion time of any job in the schedule) equals at most B? (Of course the traditional scheduling assumptions apply: no machine can process more than one job at a time, no job can be processed by two machines, no interruptions are allowed).

Let us now recall this other problem called Partition. Given a set S of n integers s_i $(i = 1, \ldots, n)$, does there exist a subset $S' \subset S$ such that $\sum_{i \in S'} s_i = \sum_{i \in S \setminus S'} s_i$? Let us assume that Partition is NP-complete (because it actually is). This knowledge about the hardness of Partition will allow us to infer that MS2 is hard as well.

Theorem 4.3.3 *MS2 is NP-complete.*

Proof: First, we build, given an arbitrary instance of Partition, an instance of MSP2. Then, we show equivalence between yes-instances of the two problems.

More concrete: for each integer s_i in the input of Partition $(i = 1, \ldots, n)$, define a corresponding job i. So, since there are n integers in the input of Partition, there are n jobs in the corresponding instance of MSP2. The length of each job is chosen as follows: $p_i := s_i$. We set $B := \frac{\sum_i s_i}{2}$. This completes the instance of MS2.

We now show the equivalence of the yes-instances of the two problems. If the instance of Partition is a yes-instance, that is, if we can partition the n integers in two sets that have equal sum, then we put all jobs corresponding to integers in one set of partition on one machine, and all remaining jobs on the other machine. Since Partition is a yes-instance, both machines will end at B, and the answer to the instance of MS2 is yes. Also, if the makespan of the resulting instance is B, then each machine processes a set of jobs whose total length is no more than B. Hence, the two sets of Partition follow, and Partition must be a yes-instance. □

Clearly, this reduction is an easy one, in the sense that problem MS2 is quite close to Partition. Much more elaborate reductions have been designed relating Satisfiability to Hamiltonicity for instance.

Returning to optimization problems as stable set, the TSP, and 0-1 integer programming, it is widely suspected that there does not exist *any* polynomial algorithm for these problems. This is a typical feature of so-called *NP-hard* problems which we define (very informally again) as follows. We use the phrase "NP-hard" for optimization problems whose corresponding decision variant is NP-complete.

Definition 4.3.1 *A problem P is NP-hard if it is as least as hard as the 0-1 integer programming problem.*

Thus, in light of Definition 4.3.1, one can show that a problem is NP-hard by showing that any algorithm for solving the problem, is able to solve 0-1 integer programming.

The next claim has resisted all proof attempts (and there have been many) since the early 70's, but the vast majority of mathematicians, computer scientists, and operations researchers believe that it holds true.

$P \neq NP$ **conjecture.** *If a problem is NP-hard, then it cannot be solved by a polynomial algorithm.*

The $P \neq NP$ conjecture, if true, expresses a deep and fundamental fact of complexity theory. Its implications are of enormous importance for the development of algorithms in operations research and related areas. Indeed, a large number of combinatorial optimization problems turn out to be NP-hard, and hence difficult to solve.

Proposition 4.3.1 *The following problems are NP-hard:*

- *travelling salesman;*
- *stable set;*
- *graph coloring;*
- *knapsack;*
- *assembly line balancing;*

- *three-dimensional assignment;*
- *facility location;*
- *jobshop scheduling;*
- *several hundreds of other CO problems.*

It is quite remarkable that most of the problems in the above list can in fact be formulated as *special cases* of the 0-1 LP problem. This is obvious, in particular, for the knapsack problem, but it is also true (though less obvious) for graph equipartitioning, or for the travelling salesman problem, or for graph coloring, etc. Thus, the actual meaning of Definition 4.3.1 is that all these NP-hard problems are somehow equivalent, in the sense that an efficient algorithm *for any of them* would immediately provide an efficient algorithm *for all of them*.

From a practical point of view, however, some NP-hard problems turn out to be more difficult than others. For instance, instances of the knapsack problem are usually quite easy to solve as compared with instances of general 0-1 LP problems. Nevertheless, NP-hard problems seem to be intrinsically tougher than linear systems, LP problems or shortest path problems. We explore this topic in the next section.

4.4 Some NP-hard problems are harder than others

Let us revisit knapsack (see Section 3.4.2): we are given a set of n items, each with an integral weight a_j and an integral value c_j for $j = 1, \ldots, n$. Feasible solutions are itemsets whose cumulative weight is at most b. The objective is to find a feasible solution of maximum value.

Consider the following algorithm for solving it (see e.g. Vazirani [13]). Let C be the most valuable item, i.e., $C = \max_j c_j$. No feasible solution will have a value exceeding nC. For each $j \in \{1, 2, \ldots, n\}$, and for each $c \in \{1, \ldots, nC\}$, let $S_{j,c}$ reflect the item subset in $\{1, \ldots, j\}$ whose total value equals exactly c, and whose total size is minimized. Let $A(j, c)$ denote the size of the set $S_{j,c}$, where $A(j, c) = \infty$ if no such set exists. Evidently, we know $A(1, c)$ for each $c \in \{1, \ldots, nC\}$. Here is the recurrence:

$$A(j+1, c) \;=\; \begin{cases} \min\{A(j,c), a_{j+1} + A(j, c - c_{j+1})\} & \text{if } c_{j+1} < c, \\ A(j,c) & \text{otherwise.} \end{cases}$$

Consider the running time of this algorithm (that we refer to as *KnapDP*). For each element of $A(j, c)$, we need to perform a constant number of operations, hence we arrive at a complexity of $O(n^2C)$ for this algorithm. This seems polynomial, doesn't it? But that cannot be true; it would mean that we can solve Knapsack in polynomial time, which would mean that P=NP. No, instead we should realize what exactly the size of the input of an instance of Knapsack is: we need to encode all integers a_j, c_j $(j = 1, \ldots, n)$ and b. This means that an appropriate description of the length of the input, i.e., the number of bits used to encode the instance I is:

$$\text{size}(I) = \log\, b + \sum_{j=1}^{n}(\log\, a_j + \log\, c_j).$$

For an algorithm with complexity Cn^2 to be polynomial for Knapsack, there would need to exist a polynomial function $p(\cdot)$, and constants N_0 and K, such that:

$$Cn^2 \leq K \times p(\text{size}(I)) \;\text{ for size}(I) \geq N_0.$$

We can reformulate this as follows: do there exist constants N, M, k such that:

$$Cn^2 \leq M \times \text{size}(I)^k \;\text{ for size}(I) \geq N?$$

If the answer is yes, then the running time of KnapDP would be bounded by a polynomial in the input size, and KnapDP would be a polynomial time algorithm for Knapsack. That, however, is not the case. We know that the following inequalities are true:

$$\log\, C \leq \text{size}(I) = \log\, b + \sum_{j=1}^{n}(\log\, a_j + \log\, c_j) \leq \log\, nA + n\log\, A + n\log\, C,$$

where A denotes $\max_j a_j$. In case the answer to the question above is yes, it follows that:

$$Cn^2 \leq M(\log\, nA + n\log\, A + n\log\, C)^k \text{for } \log\, C \geq N.$$

Thus:

$$Cn^2 \leq M(2n\log\, A + n\log\, C)^k \;\text{ for } \log\, C \geq N.$$

Consequently:

$$Cn^2 \leq M(3n\log\, C)^k \;\text{ for } \log\, C \geq N.$$

And finally:

$$\frac{C}{(\log\, C)^k} \leq M3^k n^{k-2} \;\text{ for } C \geq 2^N.$$

This apparently must hold for a fixed n, but it contrasts with a known fact, namely that for each k: $\lim_{C\to\infty} \frac{C}{(\log\, C)^k} = \infty$.

Concluding: an algorithm with running time $O(n^2C)$ such as KnapDP, is not a polynomial-time algorithm for Knapsack.

Of course, it is true that all this depends on how we encode the input: we are using a binary encoding. If the world uses a so-called *unary encoding*, i.e., if one represents the number C by a string of C "1"s, then the picture changes. Indeed, under such an encoding, the input size is (much) larger than under a binary encoding, and algorithms have more room to work with, and may in fact become polynomial time. In fact, KnapDP is a polynomial time algorithm under a unary encoding - such algorithms are also called *pseudo-polynomial*. Notice that this phenomenon only plays a role when numbers are present in the input for a problem. The phrase "strongly NP-complete" is often used for number problems that remain NP-complete even under a unary encoding, while the phrase "weakly NP-complete" is used for a number problem that becomes polynomial solvable. Thus, problems like Knapsack, Partition, MS2 are weakly NP-complete (or NP-complete in the weak sense). An example of a number problem that is strongly NP-complete is 3-Partition.

Consider the algorithm KnapDP once more. Its complexity may not be polynomial in the size of the input; that fact relies on the values c_j and b not being upper bounded by a polynomial in n. Indeed, if each c_j, as well as b, would not exceed n^{100}, KnapDP would be a polynomial time algorithm. Can we trade-off accuracy in representing the numbers c_j and b for speed? That is indeed the idea that we want to pursue. Let us introduce an error parameter $\epsilon > 0$, and, instead of insisting on an optimum solution, let us accept that KnapDP delivers a solution whose value is at least $(1-\epsilon) \times OPT$ (where OPT refers to the optimum value of the original instance defined by a_j, c_j, and b). A typical value of ϵ would be 0.01. We now redefine the values c_j as follows, using $K \equiv \frac{\epsilon C}{n}$:

$$c'_j := \lfloor \frac{c_j}{K} \rfloor.$$

Observe that C is not polynomially bounded in n (otherwise KnapDP is already polynomial); hence we expect c'_j to be small compared to c_j.

Let S denote the set of items selected by running KnapDP on the knapsack instance defined by a_j, c'_j $(1 \le j \le n)$ and b.

Lemma 4.4.1 $\sum_{j \in S} c'_j \ge (1-\epsilon)OPT$.

Proof: For any item j $(1 \le j \le n)$, since $c'_j = \lfloor \frac{c_j}{K} \rfloor \ge \frac{c_j}{K} - 1$, it is true that $Kc'_j \ge c_j - K$. Let O denote the optimal itemset (of the original instance). It follows that:

$$\sum_{j\in O} c_j - K\sum_{j\in O} c'_j \le |O|K \le nK.$$

Clearly, since the itemset O is a feasible solution to the modified instance, KnapDP will return a solution with value at least $\sum_{j\in O} c'_j$, which, in terms of the original instance, corresponds to the value $K\sum_{j\in O} c'_j$. Thus

$$\sum_{j\in S} c_j \ge \sum_{j\in S} Kc'_j \ge K\sum_{j\in O} c'_j \ge \sum_{j\in O} c_j - nK = OPT - \epsilon C \ge (1-\epsilon)OPT.$$

□

Theorem 4.4.1 *KnapDP finds a solution to Knapsack within a factor of* $(1-\epsilon)$ *of OPT. Its running time equals* $O(n^2\lfloor\frac{C}{K}\rfloor) = O(n^2\lfloor\frac{n}{\epsilon}\rfloor)$.

Observe that this running time is polynomial in n, and in $\frac{1}{\epsilon}$. KnapDP is an example of a *fully polynomial time approximation scheme*, an FPTAS. When the input of an optimization problem is extended with an error parameter $\epsilon > 0$, the existence of a set of algorithms (one for each $\epsilon > 0$) that deliver a feasible solution within a factor of $(1-\epsilon)$ of the optimal solution with a polynomially bounded running time, means that the problem has a PTAS. When in addition the running time is polynomial in $\frac{1}{\epsilon}$, the problem has even an FPTAS.

When a problem is NP-complete, the next best thing one can hope for is the existence of an FPTAS. Such schemes need not exist; in fact, strong NP-completeness rules out the existence of an FPTAS (unless P=NP of course):

Theorem 4.4.2 *If problem Q is strongly NP-hard, the existence of an FPTAS for problem Q implies P=NP.*

4.5 Exercises

Exercise 1

State for each of the following decision problems whether you think that it is contained in NP and/or co-NP. If you claim containment, then provide a corresponding certificate. If you claim non-containment, then state your intuition about it.

(a) Instance: A logical formula F in CNF. Question: Does F possess at most one satisfying truth assignment?

(b) Instance: A logical formula F in CNF. Question: Does F possess a unique satisfying truth assignment?

(c) Instance: A logical formula F in CNF. Question: Is there a truth assignment for which F evaluates to FALSE?

(d) Instance: A set of n cities; the distances between these cities; a bound B. Question: Is the length of the shortest TSP-tour equal to B?

(e) Instance: An edge-weighted, undirected graph G; a bound B. Question: Does G have a spanning tree of weight at least B?

(f) Instance: An edge-weighted, undirected graph G. Question: Is the minimum weight perfect matching in G unique?

Exercise 2

Consider a decision problem X, and assume that every instance of X can be rewritten in polynomial time into an equivalent instance of SAT.

(a) True or false: Then X lies in NP.

(b) True or false: Then X is NP-hard.

Exercise 3

A set of closed intervals on the real line is called *independent*, if no two of these intervals share a common point. The problem INDEPENDENT INTERVAL SET asks for a given set S of intervals and a given bound k, whether there exists an independent subset of at least k intervals.

(a) Decide whether INDEPENDENT INTERVAL SET $\leq_p$ INDEPENDENT SET.

(b) Decide whether INDEPENDENT SET $\leq_p$ INDEPENDENT INTERVAL SET.

Exercise 4

Prove NP-hardness of the PARTITION problem. An instance consists of positive integers $a_1, \ldots, a_n$ with $\sum_{i=1}^{n} a_i = 2A$. The question is whether there exists an index set $I \subseteq \{1, \ldots, n\}$ with $\sum_{i \in I} a_i = A$.

Exercise 5

Prove NP-hardness of the ZERO-CYCLE problem. An instance consists of an undirected graph $G = (V, E)$ together with a weight function $w : E \to \mathbb{Z}$. The question is whether the graph contains a simple cycle so that the sum of edge weights along the cycle equals zero.

Exercise 6

Prove NP-hardness of the HAMILTONIAN CYCLE problem in undirected bipartite graphs.

Exercise 7

An instance of the HAMILTONIAN PATH problem consists of a graph $G = (V, E)$ and two vertices $s.t \in V$. The question is to decide whether the graph contains a simple path connecting s to t and visiting every vertex in the graph exactly once.

Prove NP-hardness of the HAMILTONIAN PATH problem in (a) undirected graphs; (b) directed graphs.

Exercise 8

An instance of the DEGREE CONSTRAINED SPANNING TREE problem consists of an undirected graph $G = (V, E)$ and a positive integer d, and asks whether G has a spanning tree with maximum degree bounded by d. Prove NP-hardness of the DEGREE CONSTRAINED SPANNING TREE problem.

Exercise 9

Prove NP-hardness of the SET-COVER problem. An instance consists of a ground-set X, subsets $S_1, \ldots, S_m$ of X, and an integer k. The question is whether there exists a sub-collection of k of the given subsets whose union is X?

Exercise 10

Consider the single clause $c = (x \vee y \vee z)$ and the following set S_c of ten clauses:

$$(x),\ (y),\ (z),\ (w);\quad (\bar{x} \vee \bar{y}),\ (\bar{x} \vee \bar{z}),\ (\bar{y} \vee \bar{z});\quad (x \vee \bar{w}),\ (y \vee \bar{w}),\ (z \vee \bar{w})$$

(a) Show: If a truth-setting for x, y, z satisfies the clause c, then it can be extended to a truth-setting for w such that seven clauses in S_c are satisfied.

(b) Show: If a truth-setting for x, y, z fails to satisfy clause c, then every extension to w satisfies at most six clauses in S_c.

(c) Deduce (by a reduction from 3-SAT) the NP-hardness of the following problem: Given a 2-SAT formula (where every clause contains at most two literals) and an integer k, decide whether there is a truth-setting that satisfies at least k clauses.

Exercise 11

Consider the following simplification of an arbitrary instance (L, C) of SAT where L represents the literals and C the clauses. Each instance (L, C) can be simplified recursively as follows:

- a clause c containing both literal a and its negation $\bar{a}$ can be deleted from C,
- if a literal a appears in clauses C always negated or always un-negated, make the occurrence true everywhere. Next, delete all corresponding clauses from C, and delete a from L,

- if a clause contains one occurrence only, i.e., $c = (a)$ (or $c = (\bar{a})$), set a true (false), and update all other clauses containing a or $\bar{a}$ accordingly; delete c and all trivially true clauses from C; delete a from L. If a clause becomes trivially false, instance (L, C) is not satisfiable.

This simplification takes polynomial time as each iteration removes at least one clause or one literal. After simplification, either (L, C) turns out to be not satisfiable, or each clause contains at least two distinct literals; each literal appears in at least two distinct clauses, in negated and in un-negated form.

Recall that it follows from these course notes that 3-SAT is NP-hard. Which of the following variants of SAT are polynomially solvable?

(a) Every clause contains exactly two literals

(b) Every clause contains at most one un-negated literal

(c) Every clause contains at most one negated literal

(d) Every variable occurs in at most two clauses

Exercise 12

A graph possesses a 3-coloring, if its vertices can be colored with three colors (red, blue, yellow) such that adjacent vertices always receive different colors.

Consider the following nine-vertex graph G_9. Two triangles y_1, y_2, y_3 and y_4, y_5, y_6 are connected to each other by the edge $[y_1, y_4]$. Furthermore there are three vertices a, b, c and the three edges $[a, y_2]$, $[b, y_3]$ and $[c, y_5]$. We consider 3-colorings of the vertices of G_9.

(a) Prove: If a, b, c are all colored by the same color f, then also y_6 must be colored f.

(b) Prove: Any coloring of a, b, c that assigns color f to at least one of a, b, c can be extended to a coloring of G_9 that assigns color f to vertex y_6.

(c) Deduce (by a reduction from 3-SAT): Deciding whether a given input graph has a 3-coloring is NP-complete.

Exercise 13

A single machine has to process n jobs $j = 1, \ldots, n$; job j has an integer processing time of p_j time units, and should be completed by its due date d_j. A job is said to be *late* if it is completed after its due date; otherwise it is *early.* A fundamental problem in scheduling theory is to schedule the jobs so as to minimize the *number of late jobs.* The Moore-Hodgson algorithm solves this problem in $O(n \log n)$ time: Schedule the jobs in order of non-decreasing due dates; as soon as a scheduled job is late, remove the job with the largest processing time from the schedule, and mark it to be late. We consider three generalizations of this problem.

(a) Each job j ($j = 1, \ldots, n$) has a given weight $w_j \in \mathbb{Z}^+$, and we want to minimize the *weighted number of late jobs*, that is, the sum of the weights of the late jobs. Prove that this problem is NP-hard.

(b) Each job j has a release date r_j (and hence cannot be processed before r_j). Prove that it is NP-hard to minimize the *number of late jobs.*

(c) Given is an acyclic directed graph G with the jobs as vertices. If G contains a directed path from vertex j to vertex k, then job k cannot be started before job j has been completed. We now want to minimize the *number of late jobs* subject to these *precedence constraints.* Prove that this problem is NP-hard.

Exercise 14

You are helping to organize a summer sports camp that covers n different sports (soccer, volleyball, etc). For each sport, the camp is supposed to have at least one councelor who is skilled in that sport. You have received application letters from m potential councelors that specify their skills. Your goal is to hire the smallest possible number of councelors so that all n sports are covered.

Formulate the resulting optimization problem as a decision problem. Prove that the decision problem is NP-complete.

Exercise 15

Harry Potter is looking for a Bowtruckle that has made itself invisible and is hiding in one of the vertices of a graph. Harry Potter repeatedly points his magic wand at some vertex and casts the SEMI-REVELIO spell. When the spell hits the Bowtruckle for the first time, the Bowtruckle leaves its vertex and moves to an adjacent vertex; as the Bowtruckle moves silently and invisibly, Harry is not aware of the move. When the spell hits the Bowtruckle for the second time, it finally becomes visible to Harry.

Harry wants to make the Bowtruckle visible while casting as few SEMI-REVELIO spells as possible. A vertex sequence is called *revealing*, if casting the SEMI-REVELIO spells at the vertices according to the sequence guarantees that the Bowtruckle eventually becomes visible (independently of its initial hiding place).

Problem: REVEALING SEQUENCE

Instance: A graph $G = (V, E)$; a bound k

Question: Does there exist a revealing vertex sequence of length at most k?

(a) Prove that REVEALING SEQUENCE lies in NP.

(b) Prove that REVEALING SEQUENCE is NP-hard.

Exercise 16

"A peasant had to transport to the far side of a river a wolf, a goat, and a bundle of cabbages. The only boat he could find was one which would carry only two of them. For that reason he sought a plan which would enable them all to get to the far side unhurt. Let him, who is able, say how it could be possible to transport them safely?" In a safe transportation plan, neither wolf & goat nor goat & cabbage can be left alone together. There exists a solution where the peasant makes seven boat trips across the river.

Consider the following generalization to arbitrary graphs $G = (V, E)$: Now the peasant has to transport a set V of items/vertices across the river. Two items are connected by an edge in E, if they are *conflicting* and thus cannot be left alone together without human supervision. The available boat has capacity $b \geq 1$, and thus can carry the man together with any subset $S \subseteq V$ of at most b items. The question is whether there exists a safe transportation plan that allows the peasant to get all of V safely to the other side.

Problem: Feasible Transportation Plan
Instance: A graph $G = (V, E)$; a boat of capacity $b \geq 1$.
Question: Does there exist a safe transportation plan for G and b?

Problem: Short Transportation Plan
Instance: A graph $G = (V, E)$; a boat of capacity $b \geq 1$.
Question: Does there exist a safe transportation plan for G and b, in which the peasant only makes three boat trips (one forward, one back, one forward)?

(a) Prove that FEASIBLE TRANSPORTATION PLAN is NP-hard.
(b) Prove that problem SHORT TRANSPORTATION PLAN is NP-hard.
(c) Prove that SHORT TRANSPORTATION PLAN lies in NP.

Exercise 17

In the SUBSET SUM problem, we are given positive integers $a_1, \ldots, a_n$ and b. The problem is to decide whether there exists an index set $I \subseteq \{1, \ldots, n\}$ with $\sum_{i \in I} a_i = b$. We introduce a Boolean array $A[0 \ldots n, 0 \ldots b]$, and we set the entry $A[m, c]$ to TRUE if and only if there exists an index set $I \subseteq \{1, \ldots, m\}$ with $\sum_{i \in I} a_i = c$.

(a) Show that the array entries can be computed in overall time $O(nb)$.
(b) Deduce that SUBSET SUM can be solved in time $O(nb)$.
(c) Does this result imply P=NP?

Exercise 18

Assume that some decision problem X has a solution algorithm that uses polynomial time to recognize NO-instances, but uses exponential time to recognize YES-instances. Show that X lies in P.

Exercise 19

Assume that P=NP holds.

(a) Show that there exists a polynomial time algorithm that constructs a satisfying truth assignment for YES-instances of SAT.

(b) Show that there exists a polynomial time algorithm that constructs a Hamiltonian cycle for YES-instances of HAMILTONIAN CYCLE.

Exercise 20

Suppose that there is a black box that takes as input an undirected graph $G = (V, E)$ and an integer bound k, and then behaves as follows:

- If G is not connected, then the box returns "Not connected".
- If G is connected and contains an independent set of size k, the box returns "NO!".
- If G is connected and does not contain any independent set of size k, it returns "YES!".

The box always finds its answer in polynomial time (measured in the size of G and k). Show that with such a box you can solve the INDEPENDENT SET problem in polynomial time.

Exercise 21

Consider the FACTORING problem:

Instance: An integer n (written in decimal)
Solution: A list of primes whose product equals n

Describe a decision problem X that is polynomial-time equivalent to FACTORING. (Given a black box for X, you can efficiently solve FACTORING. Given a black box for FACTORING, you can efficiently solve X.)

Chapter 5

Solution Methods

Different types of solution methods for combinatorial optimization methods exist, see Table 5 for an overview. Indeed, instances of combinatorial optimization problems cannot always be solved to optimality within a reasonable amount of computing time. As discussed in Chapter 4, combinatorial problems can be NP-hard (e.g. knapsack, stable set, node cover) which implies that the best-known algorithms that guarantee an optimal solution have an enumerative character (like a branch-and-bound approach). In this chapter, we take a pragmatic point of view; we discuss approximation algorithms in Section 5.1, and local search in Section 5.2.

	exact	non-exact
polynomial	e.g. Dijkstra, Kruskal	approx algorithms
exponential	branch-and-bound	local search

Table 5.1: A classification

5.1 Approximation algorithms

This section deals with approximation algorithms. These are algorithms that return a feasible solution fast (i.e. within polynomial time), but sacrifice the guarantee of optimality. Thus, whereas exact algorithms always output an optimum solution at the expense of potentially enormous computing times, approximation algorithms form a dual approach to solving combinatorial optimization problems. An approximation algorithm guarantees a fast solution, but not necessarily an optimal one. Of course, one still wants an approximation algorithm to produce solutions that have a value that does not differ too much from the optimum value. In order to be able to judge approximation algorithms, the concept of worst case analysis is discussed in Section 5.1.1. Further, we present approximation algorithms for two specific problems, namely node cover (Section 5.1.2) and the TSP (Section 5.1.3). Finally, we discuss a general way of obtaining an approximation algorithm that is based on the so-called LP-relaxation of a formulation of a combinatorial optimization problem (Section 5.1.4).

5.1.1 Worst case analysis

How to measure the quality of a nonoptimum solution? By comparing its value (say V) to the optimum value (OPT), of course! But how exactly? Are we interested in the difference VAL - OPT? Or in the ratio VAL/OPT? Consider for instance the TSP. If you are told that the difference between your solution's value VAL and the optimum solution value OPT equals 945, this fact gives actually little information. Indeed, notice that multiplying the distances by some constant will allow you to obtain any arbitrary difference between VAL and OPT, while the instance remains essentially unchanged. However, multiplying the distances with a constant does not affect the ratio VAL/OPT. Thus, at least for this case, the measure VAL/OPT seems more suited to measure the quality of a non-optimum solution, at least for the TSP.

The worst case ratio (WCR) of an algorithm A for a minimization problem P is defined as follows. Given an instance I of problem P, let the value of the solution generated by algorithm A be $A(I)$, and let the optimum value be denoted by OPT(I); we will sometimes simply write OPT, and ignore the "(I)"-part. The ratio

$$R(I) = \frac{A(I)}{OPT(I)}$$

is a measure of the quality of the solution found. (Notice that other measure are certainly possible as well!). Now, the smallest upper bound on $R(I)$, measured over all instances I of P is called the worst case ratio (WCR) of algorithm A, i.e.,

$$WCR(A) = \sup_I \frac{A(I)}{OPT(I)}.$$

Notice that this ratio is at least 1. For a maximization problem P we define a similar quality measure as follows.

$$WCR(A) = \inf_I \frac{A(I)}{OPT(I)}.$$

This ratio is at most 1 (and not smaller than 0). How can one find the WCR of a certain algorithm? Very often, this amounts to applying a problem specific analysis. In such an analysis two things must be argued:

- one has to argue that *for all* instances I of P it is true that $\frac{A(I)}{OPT(I)} \leq R$, and that
- there exists an instance I of P for which the ratio $\frac{A(I)}{OPT(I)}$ is equal (or arbitrarily close) to R.

Then one can conclude that $WCR(A) = R$. Notice that the WCR depends on the algorithm, thus different algorithms for the same problem can yield different WCRs (as we shall see in this chapter). Of course, one could ask the question: how well can we approximate a certain problem when using only polynomial time algorithms. More formal, let $POLYTIME(Q) = \{A| \ A$ is a polynomial time algorithm for $Q\}$, then one is interested in something that can be formulated as:

$$\inf_{A \in POLYTIME(Q)} WCR(A).$$

This is certainly a valid question, and a number of results in this direction have been obtained; however, we will not go into this issue.

5.1.2 Node Cover

Recall that given a graph $G = (V, E)$, a *node cover* is a subset of the vertices $W \subseteq V$ with the property that each edge in E is incident to at least one vertex in W. The objective in this problem is to find a node cover with a minimum number of nodes. This problem is NP-hard. Let us now describe two approximation algorithms for node cover.

Algorithm 1 (Input: a graph $G = (V, E)$; output: a node cover W)

```
$W = \emptyset, E' := E$
**while** $E' \neq \emptyset$
**do**
    Choose a node $v \in V$ with largest degree;
    $W := W \cup \{v\}$;
    Update $E'$, that is remove from $E'$ all edges incident to $v$;
**od**
```

Algorithm 2 (Input: a graph $G = (V, E)$; output: a node cover W)

```
$W = \emptyset, E' := E$
```

```
while $E' \neq \emptyset$
do
    Choose an arbitrary edge in $E'$;
    $W := W \cup \{v, w\}$;
    Update $E'$, that is remove from $E'$ all edges incident to $v$ or $w$;
od
```

Algorithm 1 is a greedy type of algorithm. At first glance it seems to be better than Algorithm 2, as it uses at least part of the structure of the graph. And indeed, in instances arising from practical applications, it turns out that Algorithm 1 outperforms Algorithm 2 with respect to the number of nodes in the cover found. However, as we are about to discover, the WCR of Algorithm 2 is much better than the WCR of Algorithm 1.

Consider the following instance depicted in Figure 5.1.

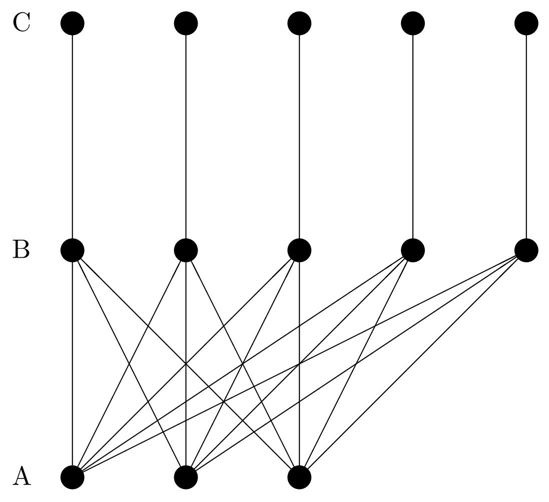


Figure 5.1: An instance for node cover.

What is the optimum? It is not hard to figure out that one needs at least 5 nodes for a node cover in the instance above, and moreover, that taking all nodes from layer B indeed constitutes a feasible node cover. Thus, this is an optimum solution. Let us now apply algorithm 1 to this instance and let us break ties by choosing nodes in the lowest possible layer first (that is nodes of layer A enjoy priority over nodes in layer B which enjoy priority over nodes in layer C). What happens? Algorithm 1 selects first the nodes from layer A and then the nodes from layer B concluding with a node cover consisting of 8 nodes. Even worse, we can generalize this instance as follows: let the number of nodes in layer B be n, then there are also n nodes in layer C and there are at most $n-1$ nodes in the bottom layer. Again, each node in layer A is connected to each node in layer B and a node in layer C is connected only to its "companion node" directly beneath it. Thus the optimum node cover consists of all nodes in layer B with optimal value n, whereas Algorithm 1 will find a solution consisting of all nodes of layer A and B, with a value equal to almost twice the optimum value. What can we deduce from this example concerning the WCR of Algorithm 1? Well, we can only say that it is at least 2. One cannot conclude that the WCR equals 2 since there may exist instances on which Algorithm 1 fares even worse.

And these examples exist. Consider Figure 5.2.

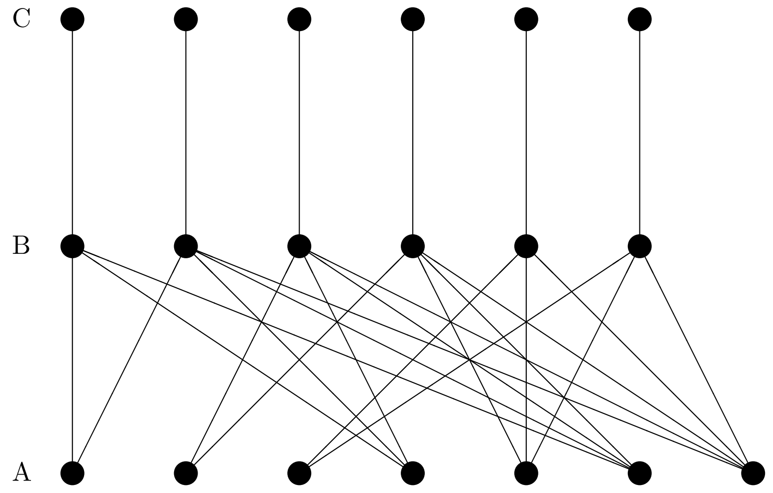


Figure 5.2: Another instance for node cover.

In this instance we see that the optimum node cover still consists of all middle nodes (i.e. value 6), whereas the node cover constructed by Algorithm 1 will consist of all nodes in the two bottom layers (i.e. a solution with value 13). From this instance alone we can conclude that the WCR of Algorithm 1 must be worse than 2.1666. However, by generalizing this instance an even more dramatic statement can be made:

Theorem 5.1.1 *For each $r \geq 0$, there exist instances I of node cover such that $A(I) \geq r \cdot OPT(I)$*

In other words, the WCR of Algorithm 1 is unbounded; it is impossible to find a constant R such that $\frac{A(I)}{OPT(I)} \leq R$. Let us motivate this theorem by generalizing the instance in Figure 5.2. The structure of this instance is as follows. In case of 6 middle nodes we do the following: partition the nodes from B into 3 pairs and join the nodes in each pair with a node from layer A. Then we partition the nodes in layer B into two triples, and again join all nodes in a triple with a new node from layer A. Repeat this for quadruples and quintuples, and so on, possibly leaving out some nodes from layer B, and always adding a new node in layer A. Then, when applying Algorithm 1, there is always a node from the bottom layer with highest degree, consequently Algorithm 1 will find a node cover consisting of $L(n) + n$ nodes, where $L(n)$ is the number of nodes in layer A. How large is $L(n)$? Observe that $L(n) = \sum_{j=2}^{n-1} \lfloor n/j \rfloor$. We leave the exact proof of Theorem 5.1.1 as an exercise.

What about Algorithm 2? It is easy to establish a lower bound of 2 on the WCR. Indeed, when simply taking a graph consisting of $2n$ vertices and n edges forming a perfect matching, one observes that n is the value of a minimum node cover, whereas Algorithm 2 selects all $2n$ nodes. However, it turns out that this is the worst that can happen for Algorithm 2:

Theorem 5.1.2 $WCR(Algorithm 2) = 2$.

The argument is as follows. Of course, any node cover must cover all edges chosen by Algorithm 2. However, these edges do not have a node in common, and therefore each edge must be covered by a different node in any node cover. Thus no node cover can be smaller than half the size of the node cover found by Algorithm 2. Together with the example sketched above Theorem 5.1.2 now follows.

5.1.3 The Traveling Salesman Problem (TSP)

In a sense the TSP is a harder problem to solve than node cover. This follows from the well known fact that, unless P=NP, no polynomial time algorithm for the TSP exists that has a bounded WCR (which contrasts with Algorithm 2 in the previous section).

What we can do is to restrict our instances. Recall that the input to the TSP is a distance matrix D. In the sequel we restrict ourselves to instances for which the distances satisfy the *triangle inequality*, that is, we have that $d_{ik} \leq d_{ij} + d_{jk}$ for all i, j, k. Observe that many practical problems satisfy this restriction. Indeed, TSP instances coming from actual "travel settings" are quite likely to obey the triangle inequality.

The double tree algorithm (DT).
Algorithm DT consists of four phases. In the first three phases, we construct an Euler-cycle that we convert into a Hamilton circuit in Phase 4.

Phase 1. Construct a minimum spanning tree with respect to the distance function d.
Phase 2. Double all edges in the tree. Notice that the resulting graph is Eulerian.
Phase 3. Determine an Euler cycle in the Eulerian graph determined in Phase 2. Since the Eulerian

graph is connected (it contains the minimum spanning tree as a subgraph), the cycle contains each vertex at least once.

Phase 4. Convert the Euler cycle into a Hamilton cycle by applying *shortcuts*, i.e., replace a pair of consecutive edges $\{i, j\}$ and $\{j, k\}$ in the Euler cycle by $\{i, k\}$. We are only allowed to do this if j appears somewhere else in the Euler cycle.

Example Consider the following 7-city TSP instance.

	1	2	3	4	5	6	7
1		1	2	2	3	4	5
2			1	1	2	3	4
3				2	3	4	5
4					1	2	3
5						1	2
6							1
7							

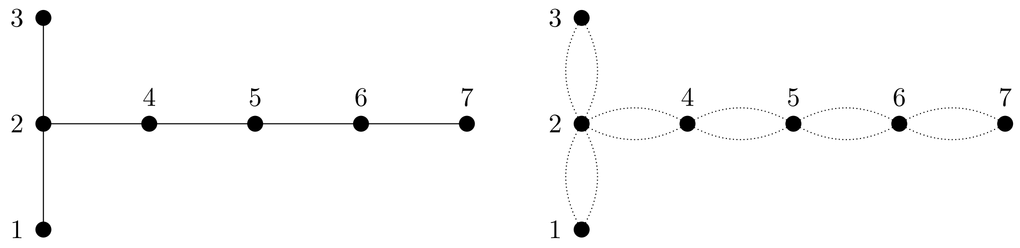


Figure 5.3: Phases 1 and 2.

The Euler-cycle generated in phase 3 is 1232456765421.

The operation described in Phase 4 can be performed on any cycle. Its effect is that a new cycle is constructed with one edge less; in this cycle, there is one vertex that is visited one time less in comparison with the previous cycle. Due to the assumption that the length function satisfies the triangle inequality, it follows that applying a shortcut does not increase the length of the cycle. Let us formally record this observation in a lemma.

Lemma 5.1.1 *Let $G = (V, E)$ be a complete graph, and let $d : E \mapsto \mathbb{R}^+$ be a distance function on the edges, which satisfies the triangle inequality. Let C be a cycle in this graph with total length $d(C)$. If C'*

is a cycle constructed from C by applying shortcuts, then we have that the total length of C' is no more than $d(C)$.

Let us now be more specific concerning Phase 4. We apply a shortcut on each second appearance of a vertex j. Therefore, each vertex remains in the cycle at least once. After the shortcut has been applied to all second appearances of the vertices, the cycle contains each vertex exactly once, that is, we have constructed a Hamilton cycle.

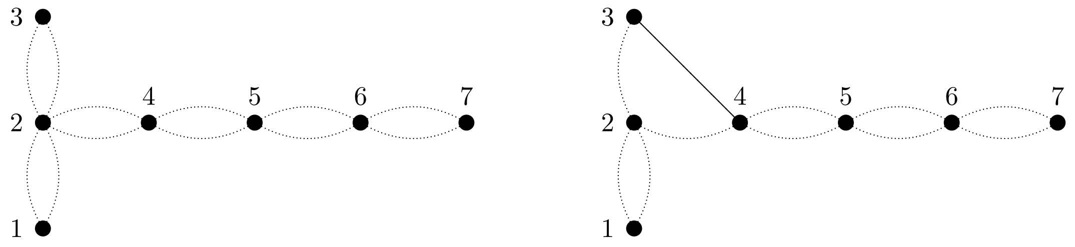


Figure 5.4: Phase 4: replacing 324 by 34

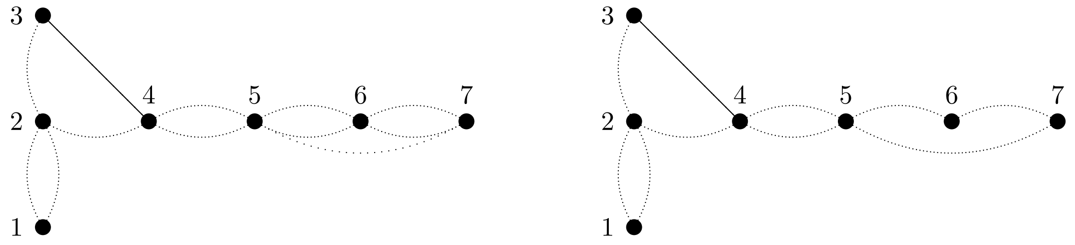


Figure 5.5: Phase 4: further replacements

We now show that Algorithm DT will never produce a solution with length more than twice the length of an optimal solution. In other words:

Theorem 5.1.3 $WCR(DT) \leq 2$.

Given an instance I, let $\mathrm{OPT}(I)$ denote the length of an optimal tour, and let $z_{DT}(I)$ denote the length of the tour constructed by algorithm DT. Finally, let $z_T(I)$ denote the length of a minimum spanning tree. We first prove that $z_T(I) \leq \mathrm{OPT}(I)$ for all instances I; we then complete the proof by showing that $z_{DT}(I) \leq 2 \cdot z_T(I)$.

1. Consider any optimal tour. If we delete an arbitrary edge from it, then we obtain a . . . spanning tree.

By definition, the length of *a* spanning tree does not exceed the length of a minimal spanning tree, and hence, we know that its length amounts to at least z_T. Concluding, we have that $z_T \leq \text{OPT}$.

2. The total length of the edges in the Eulerian graph that is constructed by doubling the minimum spanning tree is equal to $2 \cdot z_T$. From Lemma 5.1.1, it follows that the length z_{DT} of the Hamiltonian circuit that is obtained by applying shortcuts, amounts to no more than the length of the Eulerian cycle, which is equal to $2 \cdot z_T$.

Combining both results, we get $z_{DT} \leq 2 \cdot z_T \leq 2 \cdot \text{OPT}$.

The tree-matching algorithm (TM).

From the analysis above, it follows that, if we want to improve our worst-case ratio, then we can try to decrease the length of the Eulerian cycle. In the sequel we will do so. This means that we use the same structure of algorithm DT. In fact, Phases 1, 3, and 4 in Algorithm TM are identical to the corresponding phases in Algorithm DT. Thus, we only change the phase in which the Eulerian Cycle is constructed. Recall that a graph is Eulerian if and only if it is connected, and each vertex has even degree. It seems obvious to start with a minimum spanning tree to make sure that the graph is connected. The only problem left is to take care of the vertices with odd degree, which we denote by V_0; notice that the number of vertices with odd degree is even. We see that we can get even degree in each of these vertices by adding a perfect matching on these vertices V_0 (see Chapter 3 for the definition of a perfect matching). This is exactly what happens in phase 2 of algorithm TM, where we will compute a minimum weight perfect matching M on the vertices in V_0.

Consider the example again. The initial minimum spanning tree contains 4 vertices of odd degree, namely 1, 2, 3, and 7. The minimum weight perfect matching of these vertices consists of the edges $\{1, 2\}$ and $\{3, 7\}$. Thus, we get the following extension of the minimum spanning tree.

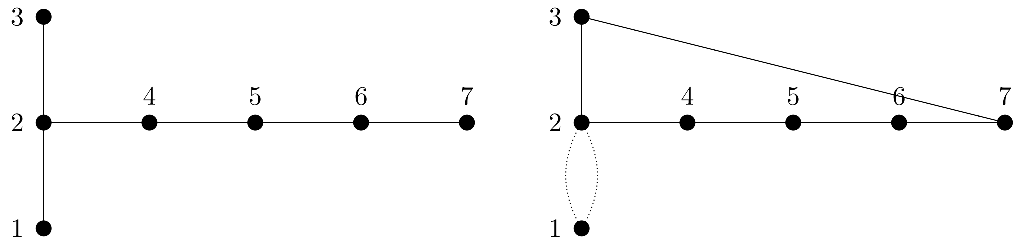


Figure 5.6: Phases 2 of algorithm TM.

An Euler-cycle in this graph is 1237654$\underline{2}$1, where 2 is the only vertex that appears more than once, and therefore we remove one of its occurrences.

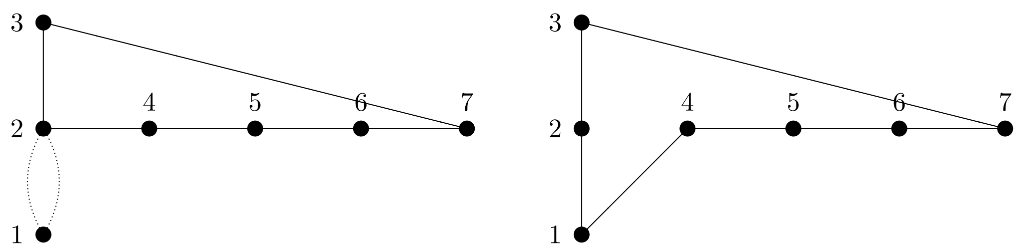


Figure 5.7: Replacing 421 by 41.

This small change with respect to algorithm DT results in a better worst-case behaviour. For any instance of the TSP (satisfying the triangle-inequality), algorithm TM constructs a tour with length no more than $\frac{3}{2}$ times the length of an optimal tour. To prove this, it suffices to show that the length z_M of the matching M is no more than $\frac{1}{2}$ times OPT. Namely, then $z_{TM} \leq z_T + z_M \leq \text{OPT} + \frac{1}{2} \cdot \text{OPT} = \frac{3}{2} \cdot \text{OPT}$.

The proof makes use of a shrinking argument. Consider any optimal tour with value OPT. Apply shortcuts such that only the vertices in V_0 remain in the cycle; this yields a cycle C on the vertices in V_0 with length no more than OPT. C can be partitioned into two perfect matchings M_1 and M_2 on V_0 by 'walking' along the circuit and putting the first edge in M_1, the second edge in M_2, the third edge in M_1, etc. Notice that the cycle contains an even number of edges, because $|V_0|$ is even.

Since M_1 and M_2 are perfect matchings on V_0, we have that $d(M) \leq d(M_1)$ and $d(M) \leq d(M_2)$. Hence, we have that $2 \times d(M) \leq d(M_1) + d(M_2) = d(C) \leq \text{OPT}$, which was to be proved.

Notice that for both algorithms, our analysis of the worst-case ratio depends heavily on the assumption that the triangle inequality holds. It can be shown that, in case the triangle inequality fails to hold, the worst-case ratio is unbounded (as could be inferred from the beginning of this section).

5.1.4 A general approach using Integer Programming

Sofar, the approximation algorithms we discussed in this Chapter are very specific to the problem under investigation. When confronted with an arbitrary discrete optimization problem, these algorithm provide little clues on how to proceed. Indeed, there seems little hope of turning the $\frac{3}{2}$-approximation algorithm for the TSP into an algorithm working for an arbitrary discrete optimization problem. Therefore, we discuss here a more general approach towards specifying approximation algorithms for discrete optimization problems.

First, we envision that our problem can be formulated as an integer program. In its general form, we state the formulation, referred to as (IP), as follows.

$$v_{IP} = \text{minimize} \quad \sum_{j=1}^{n} c_j x_j \tag{5.1}$$

$$\text{subject to} \quad Ax \geq b \tag{5.2}$$

$$x_j \in \{0,1\} \quad \text{for all } j = 1, 2, \ldots, n. \tag{5.3}$$

Next, we introduce its corresponding *Linear Programming relaxation*, or LP relaxation for short. This LP relaxation arises when we replace the integrality constraints (5.3) by the linear constraints $0 \leq x_j \leq 1$ for all j. It looks as follows:

$$v_{LP} = \text{minimize} \quad \sum_{j=1}^{n} c_j x_j \tag{5.4}$$

$$\text{subject to} \quad Ax \geq b \tag{5.5}$$

$$0 \leq x_j \leq 1 \quad \text{for all } j = 1, 2, \ldots, n. \tag{5.6}$$

Clearly, the word "relaxation" is appropriate, as we immediately observe that a solution feasible for the IP (5.1)-(5.3), is also feasible for the LP relaxation (5.4)-(5.6). One may well wonder however, whether studying the LP relaxation brings us in the "right direction". Indeed, when aiming for an algorithm that outputs a feasible, hopefully good, solution (that necessarily has a value larger than v_{IP}), how can the LP relaxation be useful as v_{LP} is at most equal to v_{IP}?

The answer is that the fractional values of the variables in the LP relaxation may be used to *round* to either 0 or 1 in a way that a feasible solution to (IP) arises. And if the rounding process can be carried out with sufficient care, then the cost incurred when transforming the LP relaxation to a feasible solution can be bounded.

An important concept in this process is the so-called *integrality gap*: the ratio between the value of the integer optimum v_{IP} and the value of the LP relaxation v_{LP}. More formally, the integrality gap of a formulation IP is defined as :

$$IntGap(IP) = \sup_I \frac{v_{IP}(I)}{v_{LP}(I)}.$$

Notice that this ratio is at least 1 as we assume that (IP) has a minimization objective. Any approximation algorithm based on rounding the LP relaxation has a worst-case ratio that is at least as large as the integrality gap.

Consider, as an illustration of the approach sketched above, the Set Cover problem. Given a universe of elements $U = \{1, 2, \ldots, n\}$, given m sets $S_j \subseteq U$ $(j = 1, \ldots, m)$, select as few sets as possible such that each element of U is in at least one selected set. We call the *frequency* of an element $i \in U$, the number of sets that contain i; and we let k be the maximum frequency. Set Cover is well-known to be a hard problem; convince yourself that Node Cover (see Section 5.1.2) is actually a special case of Set Cover.

Here is an IP formulation of Set Cover, using a binary variable x_j denoting whether set S_j is selected or not ($j = 1, \dots, m$).

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{n} x_j && && (5.7)\\ \text{subject to} \quad & \sum_{j: i \in S_j} x_j \geq 1 && \text{for } i = 1, 2, \dots, n && (5.8)\\ & x_j \in \{0,1\} && \text{for all } j = 1, 2, \dots, m. && (5.9) \end{aligned}$$

We employ the following algorithm A to find a feasible solution to an instance of Set Cover. Let x^* denote the solution of the LP relaxation corresponding to Set Cover's IP.

Algorithm A:

Step 1: Solve the LP relaxation corresponding to Set Cover's IP, and denote the resulting solution by x^*. Of course, in general, x^* contains fractional entries.

Step 2: If $x_j^* \geq \frac{1}{k}$, set $x_j := 1$, else, set $x_j := 0$.

Theorem 5.1.4 *Algorithm A is a k-approximation algorithm for Set Cover.*

Proof: We need two argue (i) that the solution x found by algorithm A is feasible, and (ii) that the cost of this solution is at most equal to $k \cdot OPT$, where OPT denotes the optimal value of the instance of Set Cover.

Ad (i): Each element $i \in U$ occurs in at most k sets. By constraint (5.8) (which is present in the LP-relaxation, and hence x^* satisfies this constraint), there is at least one variable x_j^* out of the at most k variables that correspond to sets S_j containing element i, whose value is at least $\frac{1}{k}$. Hence Step 2 of Algorithm A ensures that each element is in a selected set, and hence the solution returned by A is feasible.

Ad (ii): Let $c(A)$ denote the cost of the solution x found by A. Observe that: $c(A) \leq k \cdot \sum_i x_i^* = k \cdot v_{LP} \leq k \cdot v_{IP} = k \cdot OPT$. □

5.2 Local search

In this section, we study a class of solution strategies called *local search* methods. An important characteristic is that local search methods, in general, are not exact, i.e., a local search method may not find an optimum solution. Another property is that no mathematical programming formulation is needed to devise a local search method. Many specializations of local search methods have been developed, and are known under names as Tabu Search Method, Simulated Annealing, Genetic Algorithms, Evolutionary Algorithms, and so forth. At the basis of each of these methods is local search. We refer to Aarts and Lenstra [1] for an overview of local search.

5.2.1 An example: the TSP

As mentioned before, one of the most well-studied combinatorial optimization problems is the TSP. Now, let us suppose we are given some feasible solution to an instance of the TSP, i.e., we are given a tour. Our tour need not be a best solution - however, currently, it is the best solution we have got. Our challenge is to find a way to improve this solution.

To do so, we intend to look at solutions that are in some sense *close* to our given tour. In other words, we want to find solutions that *resemble* our tour. So in what way can a solution be different, yet similar? Well, one measure of similarity is the number of edges that two tours have in common. Clearly, each tour consists of n edges, so for tours to be different they can have at most $n-1$ edges in common. It is however immediately clear that two tours having $n-1$ edges in common must be identical. So let us consider tours that share $n-2$ edges with our given tour; notice that when removing two non-adjacent edges from a tour, there is a unique pair of edges that must be added in order to find a new tour distinct from the previous one. Thus, there are $\frac{1}{2}n(n-3)$ different tours that have $n-2$ edges in common with our given tour. We refer to this set of tours as the *neighborhood* of our given tour, and the *size* of a neighborhood refers to the number of tours in the neighborhood.

The idea of local search is to inspect each tour in this neighborhood, and find out whether there is one with a length shorter than our current one. There are two possibilities: either each tour in the neighborhood has a length that is at least as large as our current tour's length, or there is a shorter tour in our neighborhood. In the first case, we call our current solution a *local optimum* (cf Chapter **??**!), and we stop. In the second case, we replace the tour we currently have, by this better tour, and iterate. In order to give a more systematic description, let us introduce some notation. We use σ to denote a tour. Further, we use:

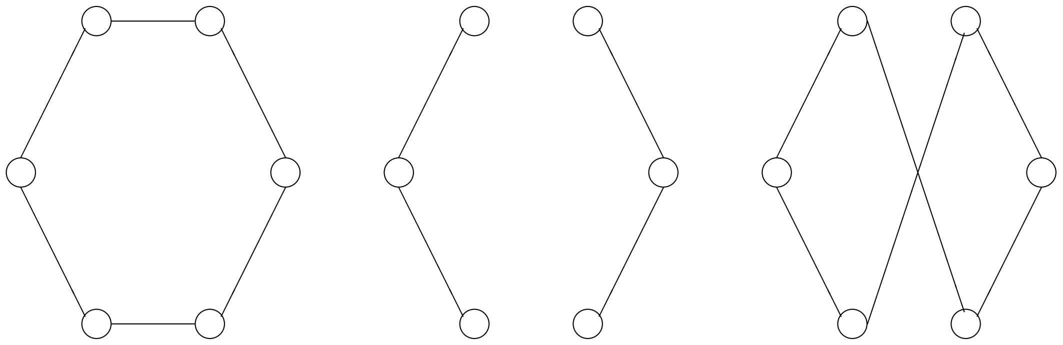


Figure 5.8: Local Search for the TSP

- $N(\sigma)$ is the set of tours that share exactly $n-2$ edges with tour σ, and
- $\ell(\sigma)$ denotes the length of a tour σ.

Notice that this definition of neighborhood implies that if some tour $\pi \in N(\sigma)$, it holds that $\sigma \in N(\pi)$. The method can now be compactly described as follows.

Algorithm 1 A local search method for the TSP

```
1: {Input: a given tour $\sigma$}
2: search $N(\sigma)$
3: if there exists a tour $\pi \in N(\sigma)$ with $\ell(\pi) < \ell(\sigma)$ then
4:     $\sigma := \pi$. Go to Step 2.
5:     else output $\sigma$.
6: end if
```

This method is known as 2OPT for the TSP. Since the number of solutions to the TSP is finite, the method ends. Various fundamental questions can be asked:

- how many iterations are needed to reach a local optimum?
- how bad can a local optimum be (in terms of tourlength)?
- how much computational effort is required to inspect a neighborhood?

Answers to these questions largely depend on the particular instance; there are known results in literature (partially) answering these questions.

One of the surprising facts is that, despite the apparent simplicity of 2OPT, in practice, it achieves high-quality results for the TSP. Solutions with a tourlength within 5% above the minimum tour length are routinely found by 2OPT in a limited number of iterations.

Of course, the 2OPT neighborhood is not the only possible neighborhood for the TSP. Other neighborhoods exist: an easy idea is to (re)define the neighborhood of a tour as the set of tours sharing at least $n-3$ edges with a given tour. Indeed, removing three edges from a given tour gives rise to a neighborhood called 3OPT; it is good to realize that for each triple of non-adjacent edges in a tour, there are 8 distinct tours in the resulting neighborhood. Thus, the size of this neighborhood, when compared to the size of the neighborhood of 2OPT, grows to $O(n^3)$. One may wonder which of these two neighborhoods should be preferred. Clearly, there is a trade-off between on the one hand the size of the neighborhood (the larger the size, the more likely it is that it contains better solutions), and on the other hand the computational effort needed to search the neighborhood (the larger size, the more time it takes to search the neighborhood).

Are there other neighborhoods for the TSP conceivable? One idea is to see a solution as an ordered list of cities, and next identify a single city, remove it from its current position in the list, and reinsert it at some other position in the list. Clearly, a new feasible solution arises, and there are around n solutions in this *insertion* neighborhood. A closer look to this neighborhood reveals that one can see this as a form of restricted 3OPT: indeed, removing a city and reinserting it elsewhere can be achieved by changing three edges, two of which are adjacent. This remains true if one redefines the neighborhood as reinserting a *segment* of consecutive cities; this neighborhood is known as the Or-neighborhood.

The most famous local search algorithm for the TSP is the Lin-Kernighan algorithm which is based on a delicate interplay of 2OPT and 3OPT. It is considered to be one of the most successful heuristics for the TSP, finding often optimum solutions even for very large instances, see Helsgaun [7]. Notice however, there is no guarantee that when applied to a particular instance, it finds an optimum solution.

5.2.2 Another example: matching

One may rightfully wonder whether the ideas in the previous section are restricted to the TSP. Or is it possible to use a similar idea for other discrete optimization problems? Let us investigate to what extent we can apply these ideas to the Matching Problem.

Again, when comparing two solutions to the perfect weighted matching problem, these two solutions may share no edge (we then view the solutions as completely distinct), or they may share up to $n-2$

edges (we then view the two solutions as very much alike). Thus, again, the number of edges in common serves as a measure of similarity of solutions. When two solutions are similar, we can describe this as one solution being in the *neighborhood* of the other solution; the concept of neighborhood can thus also be readily applied to the Matching Problem. Thus, to become more concrete, we can define the neighborhood of a solution to the perfect weighted matching problem as the set of solutions that share $n-2$ edges - indeed these solutions are quite like the original solution. Then, nothing stands in our way to specify a local search method for the perfect matching problem.

Let $\mathcal{M}$ denote the set of matchings in a given graph G. Define

- $N(M) = \{M' \in \mathcal{M} : \ |M' \cap M| \geq n-2\}$,
- $w(M)$ as the total weight of matching M.

Algorithm 2 A local search method for the Matching Problem

```
1: {Input: a given matching $M \in \mathcal{M}$}
2: search $N(M)$
3: if there exists a matching $M' \in N(M)$ with $w(M') > w(M)$ then
4:     $M := M'$. Go to Step 2.
5:     else output $M$.
6: end if
```

From a practical point of view, Algorithm 2 is not very relevant. Indeed, the availability of an efficient *exact* procedure for the Matching Problem renders this local search procedure superfluous in practice. However, the point is that it is possible to apply local search to the matching problem. Of course, there can be different ways in which one can define a neighborhood - each neighborhood gives rise to a local search algorithm. Thus, in order to specify a local search method, the key issue is to specify the neighborhood of a solution.

5.2.3 Local Search in general

The two previous subsections should have paved the way for the realization that local search can be applied generally. Indeed, the only ingredient that is needed is the possibility to define a neighborhood for any particular solution.

Thus, the key idea is to view a solution of a combinatorial optimization problem as a point in some metric space. Then, two solutions are represented by two points in space. Now, these two distinct solutions of a problem may share some properties, or they may not.

Clearly, when it comes to implementing a local search method, different choices can be made to search in a particular neighborhood. Two reasonable alternatives are as follows:(i) find a best solution in the neighborhood. If the value of this solution is better than the current best solution, swap, if not, stop. (ii) find any solution in the neighborhood whose value is better than the current best solution, and swap. Notice that one can view the problem of searching the neighborhood as an optimization problem itself. We state here a generic local search algorithm using a neighborhood N that should be specified for the particular problem at hand.

Algorithm 3 A local search method in general

```
1: {Input: a given solution $s$}
2: search $N(s)$
3: **if** there exists a solution $s' \in N(s)$ with $w(s) > w(s)$ **then**
4:     $s := s'$. Go to Step 2.
5:     **else output** $s$.
6: **end if**
```

Different comments are in order. For instance the final solution (local optimum) found depends on the starting solution - each starting solution yields a certain local optimum. It might be interesting to try Algorithm 3 for various different starting solutions, thereby finding a diverse set of local optima. Another consideration is the time spent searching the neighborhood. Should we exhaustively search the neighborhood? Or should we be content with the first solution better than our current one? These questions do not have a single answer - this depends on the problem, and even on the instance.

Let us now discuss various ways in which we can modify local search.

5.2.4 Extensions

Simulated Annealing

Basic local search as described in Subsection 5.2.3 has a very rigid criterion for accepting a new solution as the current one: if the solution's quality is better than the current one it is accepted, else it is not. Thus, when inadvertently starting out with a solution that is a local optimum, the method comes to a halt immediately, even if there is ample time available to search the solution space.

In Simulated Annealing, a solution from the neighborhood is selected at random. Next, the criterion for accepting that solution is less rigid than in basic local search. One common feature is that a solution with a quality that is better than the current one is always accepted. However, even if the solution from the neighborhood is worse, there is a probability that this solution will be accepted in Simulated Annealing. Of course, at all times the method keeps track of the best solution found; however, it is possible that the method steps to a solution that is worse than the current one. This probability of accepting a worse solution is based on two things: the amount of worsening of the objective value, and a parameter called the *temperature*. The value of this temperature depends on the amount of time that the method has been running; it is decreasing over time. A high temperature indicates a large chance of accepting worse solutions. Over the course of the algorithm's running the temperature decreases, until it is so low that in practice it boils down to basic local search. More specifically, let us use $P(\text{accept solution } s')$ as the probability that solution s' is accepted, and let us use $w(s)$ denote the value of solution s. For a maximization problem, the probably of accepting solution s' when our current solution is solution s is given by:

$$P(\text{accept solution } s') = \begin{cases} 1 & \text{if } w(s') > w(s) \\ e^{\frac{w(s')-w(s)}{T_k}} & \text{otherwise} \end{cases}$$

The parameter T_k stands for the temperature at iteration k. Observe that the probability of accepting a worse solution is larger when T_k is large (so in the beginning of the algorithm's running), and when the difference between $w(s')$ and $w(s)$ is not so large (so when the quality of solution s' is not much worse than the quality of solution s.) A formal description is as follows.

Observe that the crucial step in Algorithm 4 decides whether to step to a solution s' by comparing the value

$$e^{\frac{w(s')-w(s)}{T_k}}$$

to a number uniformly generated from the interval $[0, 1]$. Further, note that we did not specify how T_k depends on the iteration k. This is called the *cooling schedule*; it should be specified as a non-increasing function of k, and it should approach 0 when k goes to ∞.

By the way, the name of this method, "Simulated Annealing", is derived from an analogy with slowly cooling a particular material. In such a cooling process, the atoms of this material find a minimum-energy state, provided the cooling is done slowly. The idea is that slowly decreasing the temperature would lead to an optimum solution where the value of a solution corresponds to the energy of a particular state.

Algorithm 4 Simulated Annealing

```
1: {Input: a given solution $s$}
2: $k := 1$
3: repeat
4: generate some $s' \in N(s)$
5: **if** $w(s') > w(s)$ **then**
6:     $s := s'$.
7:     **if** $e^{\frac{w(s')-w(s)}{T_k}} > \text{random}[0,1]$ **then**
8:         $s := s'$
9:         $k := k + 1$
10:    **end if**
11: **end if**
12: until some stopcriterion
```

The main advantage of Simulated Annealing is that in the beginning of the algorithm, the solutions considered are more or less random selections from the solution space, since the algorithm will step to any selected solution. This allows the algorithm to "explore" the solution space, and (hopefully) converge to that part of the solution space that contains the global optimum. Practical experience with Simulated Annealing indicates that, when given enough time, it finds high-quality solutions. A cynic might add that, when given enough time, compete enumeration will also surely solve the problem. To get a simulated annealing algorithm running, one needs to specify a neighborhood, and one needs to specify the temperature, and how it decreases over time. And one needs a starting solution.

Tabu Search

The challenge of overcoming a local optimum is also the motivation for tabu search. In contrast to Simulated Annealing, however, Tabu Search is deterministic. It distinguishes itself from basic local search by selecting the best solution from the neighborhood *even if that solution is worse than the current one.* The astute reader might notice that the iteration after selecting a worse solution might not be very interesting, since we expect to return to our local optimum. That, however, is forbidden (tabu); in fact, the algorithm dynamically maintains a tabu list: a set of solutions to which the algorithm is forbidden to return. However, as an exception to this rule, if a solution on the tabu-list is good enough (if it achieves a certain *aspiration level*), the method still accepts this solution.

Algorithm 5 Tabu Search

```
1: {Input: a given solution $s$}
2: $L := \emptyset$
3: repeat
4: generate the best $s' \in N(s)$
5: **if** $s' \notin L$ **then**
6:     $s := s'$.
7:     update tabu listL.
8: **end if**
9: until some stopcriterion
```

Genetic Algorithms

We restrict ourselves to a very brief, very basic description of genetic algorithms. The terminology used to describe this class of methods comes from biology, since the process of evolution is the inspiration for this class of methods.

Given is a set P_0 of (feasible) solutions called the *the population.* The algorithm will apply local search to each of these solutions so that P_0 contains local optima only, say there are K of them. Then, these solutions in P_0 will be used to build a new set of solutions, called P_1 that will be different from (and hopefully better than) P_0. The two main procedures that the algorithm uses to find a new set of solutions is to combine two solutions from P_0 into a a single solution that is then in P_1 (*recombination*). Another procedure is called *mutation* where a solution from P_0 is modified into a new solution. Of course, how to exactly recombine two solutions into a new one, and how exactly to mutate a single solution remains problem specific; this where ingenuity is needed to create a set of solutions with better value (called the *fitness*). Local search is applied to these solutions in order to find again local optima, and in fact the best K of them are now the population P_1. Of course this process repeats itself until some stopping criterion is satisfied.

Variable Neighborhood Search

An intuitive appealing idea is as follows. Suppose, for some problem, different neighborhoods exist. Then, a solution that is a local optimum for one neighborhood need not be a local optimum for the other neighborhood as well. Thus, one can run a straightforward local search method with respect to neighborhood N_1 until the method reaches a local optimum; then we invoke neighborhood N_2, and see whether a solution is contained in N_2 that beats this local optimum (wrt N_1). If so, we can revert back to the original neighborhood to proceed, and use neighborhood N_2 to 'escape' from a solution that is locally optimal for N_1. Or we can continue to use neighborhood N_2, and see where it leads us; of course, when we have found a solution that is locally optimal wrt neighborhood N_2, we can revert back to N_1 and see whether the solution is also locally optimal wrt N_1. This idea, of using multiple distinct neighborhoods to drive the local search algorithm, is called Variable Neighborhood Search. The final solution is a solution that is simultaneously a local optimum for many different neighborhoods.

5.2.5 Local Search and Linear Optimization

At first sight, local search and linear optimization do not seem very related. Indeed, when given a solution to a linear optimization problem, say a vector x, with value cx and satisfying $Ax \leq b$, how does one define the neighborhood of x? Should one consider all vectors y that satisfy $Ay \leq b$ such that $|y - x| \leq \delta$ for some given δ? That does not seem a viable direction. There is, however, an elegant point of view that allows us to see the simplex method as a local search method. Recall that the simplex method works with feasible solutions that display a specific structure: these solutions are extreme solutions, or in other words, each solution can be partitioned into basic and nonbasic variables. Thus $x = (x_B \; x_N)$. The simplex method swaps in each iteration a nonbasic variable with a basic variable. Thus, when we see the neighborhood of an (extreme) solution x as those (extreme) solutions that have $m-1$ basic variables in common, the local search method that follows from this definition of a neighborhood is nothing else as the simplex method.

Motivated by this example, we say that a neighborhood is *exact* when each local optimum is in fact a global one. The discussion before allows us to conclude that the neighborhood defined by having $m-1$ basic variables in common is an exact neighborhood.

5.3 Exercises

Exercise 1

Consider the following greedy algorithm for the knapsack problem. Sort the objects by decreasing ratio of profit and size, and reindex the items such that the order of objects is $1, 2, \ldots, n$. Next, greedily pick objects in this order while ensuring that the capacity of the knapsack is not exceeded.

- Show that this method can behave arbitrarily bad, i.e., has unbounded worst case ratio.
- Modify this method by identifying the smallest k such that the total size of the first k objects exceeds the capacity. Next, find the best of the following two solutions: $\{1, 2, \ldots, k-1\}$ and $\{k\}$. Show that this algorithm is a 2-approximation algorithm.

Exercise 2

Consider the argument given in the beginning of Section 5.1.1 used to motivate the measure VAL/OPT as a measure for the quality of a nonoptimum solution. Does this argument apply to the bin packing problem? Explain why not, and propose a different quality measure.

Exercise 3

Consider the node packing problem (or Independent Set). Consider the following greedy algorithm.

```
*Algorithm Greedy* (Input: a graph $G = (V, E)$; output: an independent set $IS$)
$IS = \emptyset, G' = (V', E') := G$
**while** $V' \neq \emptyset$
**do**
    Choose a node $v \in V'$ with the smallest degree in $G'$;
    $IS := IS \cup \{v\}$; $V' := V' \setminus \{v\}$;
    Update $G'$, that is, for each $w \in V'$ with $\{v, w\} \in E'$, set $V' := V' \setminus \{w\}$, and $E' := E' \setminus \{v, w\}$;
**od**
```

Can you find an instance where this method fails? That is, an instance where this method does not find an optimum solution? What does this tell you about the worst-case ratio of Greedy?

Exercise 4

Consider the matching problem. What can you tell about the WCR of the following greedy algorithm for matching?

```
*Algorithm* (Input: a graph $G = (V, E)$; output: a matching $M$)
$M = \emptyset, G' = (V', E') := G$
**while** $E' \neq \emptyset$
**do**
    Choose an arbitrary edge $\{v, w\} \in E'$;
    $M := M \cup \{v, w\}$;
    Update $G'$, that is $V' := V' \setminus \{v, w\}$ and remove from $E'$ all edges incident to $v$ or $w$;
**od**
```

Exercise 5

An *edge coloring* of a graph is a coloring of the edges such that no two edges connected to the same vertex have the same color. The edge coloring problem is to minimize the number of colors used to color a graph. The greedy algorithm for edge coloring colors edge after edge. When coloring an edge it will first consider colors that are already in use before assigning a new color. What can you say concerning the WCR of this algorithm?

Exercise 6

A TSP instance is called *geometric* if each of the cities can be represented by a point in the plane, i.e. each city lies at coordinates (x_i, y_i), $i = 1, \ldots, n$. What do the result in the previous exercise tell you about WCR(DT) for the geometric TSP?

Exercise 7

The GREEDY heuristic for VERTEX COVER repeats the following step until the graph has no more edges: "Select a vertex v of highest degree, put v into the vertex cover, and delete all edges incident to v." Ties are broken arbitrarily. Show that the approximation guarantee of this GREEDY heuristic is (a) strictly worse than 2; (b) strictly worse than 1000.

Exercise 8

Find TSP instances (with triangle inequality) that illustrate that our analysis of the Double-Tree algorithm (worst case ratio 2) and our analysis of the Christofides algorithm (worst case ratio 3/2) are essentially tight.

Exercise 9

Show that for the general TSP (without triangle inequality), the existence of a polynomial time approximation algorithm with worst case ratio 2 would imply P=NP.

Exercise 10

An instance of the METRIC STEINER TREE problem consists of a complete graph on a vertex set $R \cup S$, and non-negative weights $w(e)$ on the edges that satisfy the triangle inequality. The vertices in R are called *required* vertices, and the vertices in S are called *Steiner* vertices. The goal is to find a minimum weight tree that contains all required vertices and some of the Steiner vertices.

A simple approximation algorithm uses the minimum spanning tree for the vertices in R as approximation of the Steiner tree. Determine the worst case ratio of this algorithm.

Exercise 11

An instance of BIN PACKING consists of n items with real sizes $a_1, \ldots, a_n \in [0, 1]$. The goal is to pack these items into the smallest possible number of unit size bins. The FIRST FIT algorithm for BIN

PACKING works through the item list $a_1, \dots, a_n$ one by one, and always packs the current item into the earliest (leftmost) bin into which it will fit.

(a) Show that the worst case ratio of FIRST FIT is at most 2.

(b) Show that the worst case ratio of FIRST FIT is at least 5/3.

Exercise 12

Show that for the BIN PACKING problem, the existence of a polynomial time approximation algorithm with worst case ratio 4/3 would imply P=NP.

Exercise 13

An instance of the MAKESPAN minimization problem consists of n jobs with (positive integer) lengths $p_1, \dots, p_n$ and of m identical machines. The goal is to assign the jobs to the machines, so that the largest work-load is minimized. Consider the following approximation algorithm for MAKESPAN that improves on LIST SCHEDULING: Sort and rename the jobs, such that $p_1 \geq p_2 \geq \cdots \geq p_n$, and then apply LIST SCHEDULING to the sorted instance. Analyze this improved approximation algorithm LSI for MAKESPAN with $m = 4$ machines.

(a) Show that LSI has worst case ratio at most 5/4.

(b) Find a bad instance that matches this ratio 5/4.

Exercise 14

An instance of the 3-SET PACKING problem consists of a ground set X together with a system $S_1, S_2, \dots, S_m$ of 3-elements subsets of X. The goal is to find a maximum cardinality subsystem of pairwise disjoint subsets.

Find a polynomial time approximation algorithm that finds a solution whose cardinality is at least 1/3 of the optimal cardinality.

Exercise 15

Give a polynomial time approximation algorithm with worst case ratio 1/2 for the ACYCLIC SUBGRAPH problem: An instance consists of a directed graph $G = (V, A)$. The goal is to find a maximum cardinality subset of the arcs that induces an acyclic subgraph.

Exercise 16

Design a polynomial time algorithm that takes as input a 3-colorable graph G and finds a proper coloring of G with $O(\sqrt{n})$ colors. (Hence: the input graph is guaranteed to be 3-colorable, but the corresponding 3-coloring is not known.)

Exercise 17

State an ILP formulation of the KNAPSACK problem (see Section 3.4.2), and formulate the corresponding LP relaxation. Analyze the integrality gap of the LP relaxation.

Exercise 18

A graph $G = (V, E)$ with n vertices $v_1, \ldots, v_n$ forms an instance of the INDEPENDENT SET problem.

(a) Find an ILP formulation of INDEPENDENT SET: For every vertex v_k introduce a corresponding indicator variable x_k. Introduce appropriate constraints, and express the objective function in terms of the x_k.

(b) Relax your ILP formulation to an LP formulation with continuous variables x_k. Prove: For every integer R, there exists an instance of INDEPENDENT SET, for which the ratio between the objective values of the LP and the ILP is at least R.

(c) What can you say about the integrality gap of this LP relaxation?

Exercise 19

Consider the following ILP formulation of the MAKESPAN minimization problem on identical machines with job processing times $p_1, \ldots, p_n$.

$$\begin{array}{lll}
\min & C & \\
\text{s.t.} & \sum_{i=1}^{m} x_{ij} \ = \ 1 & \text{for } j = 1, \ldots, n \\
& \sum_{j=1}^{n} x_{ij} p_j \ = \ L_i & \text{for } i = 1, \ldots, m \\
& L_i \ \leq \ C & \text{for } i = 1, \ldots, m \\
& p_j \ \leq \ C & \text{for } j = 1, \ldots, n \\
& x_{ij} \in \{0, 1\} & \text{for } i = 1, \ldots, m \text{ and } j = 1, \ldots, n
\end{array}$$

Note that the variables x_{ij} are integral, while the variables L_i and C are continuous.

(a) Show that this ILP correctly models the MAKESPAN minimization problem.

(b) Formulate the corresponding LP relaxation.

(c) Analyze the integrality gap in dependence of the number m of machines.

Exercise 20

An instance of the MAX BISECTION problem consists of an undirected graph with vertex set $V = \{1, 2, \ldots, 2n\}$ and positive real edge weights $w(i, j)$ for $[i, j] \in E$. The goal is to partition V into two parts V_1 and V_2 of size n, so that the total weight of the edges between V_1 and V_2 is as large as possible. Consider the following ILP.

$$\begin{aligned}
\max \quad & \sum_{[i,j]\in E} w(i,j)\, z_{i,j} \\
s.t. \quad & z_{i,j} \;\le\; x_i + x_j && \text{for } [i,j] \in E \\
& z_{i,j} \;\le\; 2 - x_i - x_j && \text{for } [i,j] \in E \\
& \textstyle\sum_{i=1}^{2n} x_i \;=\; n \\
& x_i \in \{0,1\} && \text{for } i \in V \\
& z_{i,j} \in \{0,1\} && \text{for } [i,j] \in E
\end{aligned}$$

(a) Show that this ILP correctly models the MAX BISECTION problem.

(b) Show that an optimal solution x and z of the ILP satisfies $z_{i,j} = x_i + x_j - 2x_ix_j$.

Next, let us consider the LP relaxation of this ILP, where the integrality constraints $x_i \in \{0,1\}$ and $z_{i,j} \in \{0,1\}$ are relaxed to the continuous constraints $0 \le x_i \le 1$ and $0 \le z_{i,j} \le 1$. Furthermore we define the auxiliary value $F(x) \;:=\; \sum_{[i,j]\in E} w(i,j)\,(x_i + x_j - 2x_ix_j)$.

(c) Prove that any feasible solution x and z of the LP relaxation satisfies the inequality $F(x) \;\ge\; \frac{1}{2}\sum_{[i,j]\in E} w(i,j)\, z_{i,j}$.

(d) Consider a solution for the LP in which the two variables x_i and x_j are both fractional. Argue that it is possible to increase one variable by $\epsilon > 0$ and to decrease the other one by the same ϵ, such that the value of $F(x)$ does not decrease and one of the two variables becomes integer.

(e) Use these arguments to design a polynomial time approximation algorithm for MAX BISECTION that yields at least 1/2 of the optimal objective value.

Exercise 21

An instance of the MAX CUT problem consists of an undirected graph $G = (V, E)$. The goal is to find a partition $V = L \cup R$ that maximizes the number of edges between L and R.

The FLIP algorithm starts with the trivial partition with $L = V$ and $R = \emptyset$. As long as the objective value can be improved by moving one vertex from L to R or from R to L, FLIP moves the corresponding vertex. When there is no further improvement possible, FLIP outputs the current partition. Show that FLIP has worst case ratio 1/2.

Exercise 22

An instance of the MAX SATISFIABILITY problem consists of a set X of logical variables $x_1, \ldots, x_n$;

a set C of clauses $c_1, \dots, c_m$ over X; a positive real weight w_c for every clause $c \in C$. Throughout this exercise we will assume that C does not contain contradictory unit-clauses; this means that for any variable x, at most one of the two clauses (x) and $(\neg x)$ is in C. The goal is to find a truth-setting of the variables in X that maximizes the overall weight of all satisfied clauses. For example, for the three clauses $c_1 = (\neg x_1)$, $c_2 = (\neg x_2)$, $c_3 = (x_1 + x_2)$ with weights $w_1 = w_2 = 3$ and $w_3 = 4$, the optimal solution satisfies two clauses with an overall weight of 7.

This exercise analyzes the quality of the bound $W = \sum_{c \in C} w_c$ for the optimal objective value of MAX SATISFIABILITY without contradictory unit-clauses (denoted by OPT); clearly OPT$\leq W$. Define $\phi = \frac{1}{2}(-1 + \sqrt{5}) \approx 0.618$ as the positive root of $\phi^2 + \phi = 1$.

(a) Show that there always exist truth-settings with objective value at least ϕW. (Hint: Set x_i=TRUE with an appropriately chosen probability p_i. Show that the expected value of the overall weight of the satisfied clauses is at least ϕW.)

(b) Use the method of conditional expectation with the result in (a), and get a deterministic polynomial time approximation algorithm with worst case guarantee ϕ.

(c) Find sets C of clauses, for which OPT comes arbitrarily close to ϕW.

Exercise 23

Consider n jobs J_j ($j = 1, \dots, n$) with positive integer processing times p_j on two identical machines. The goal is to find a schedule with machine loads L_1 and L_2 that minimizes the following objective value:

(a) $|L_1 - L_2|$

(b) $\max\{L_1, L_2\}/\min\{L_1, L_2\}$

(c) $(L_1 - L_2)^2$

(d) $L_1 + L_2 + L_1 \cdot L_2$

(e) $\max\{L_1, L_2/2\}$

For each of these problems, you are asked to either design a PTAS or to find a convincing argument against the existence of a PTAS.

Bibliography

[1] Aarts, E.H.L. and J.K. Lenstra (2003) *Local Search in combinatorial optimization*, Princeton University Press.

[2] Applegate, D., R. Bixby, V. Chvátal, and W. Cook (2006). *The Traveling Salesman Problem. A computational study*, Princeton University Press, Princeton.

[3] Boyd, S. and L. Vandenberghe (2004) *Convex Optimization*, Cambridge University Press, Cambridge.

[4] Chvátal, V. (1983) *Linear Programming*, W.H. Freeman, New York.

[5] Cook, W., W. Cunningham, W. Pulleyblank, and A. Schrijver (1998). *Combinatorial Optimization*, John Wiley and Sons, New York.

[6] Garey, M., and D. Johnson (1979). *Computers and Intractability*, W.H. Freeman, New York.

[7] Helsgaun, K. (2000). *Effective implementation of the Lin-Kernighan traveling salesman heuristic*, European Journal of Operational Research **126**, 106-130.

[8] Kellerer, H., U. Pferschy, and D. Pisinger (2004). *Knapsack Problems*, Springer, Berlin.

[9] Lawer, E., J.K. Lenstra, A. Rinnooy Kan, and D. Shmoys (editors) (1985). *The Traveling Salesman Problem*, Wiley, New York.

[10] Nemhauser, G., and L. Wolsey (1988). *Integer and Combinatorial Optimization*, John Wiley and Sons, New York.

[11] Rockafellar, R.T. (1970). *Convex Analysis*, Princeton University Press, Princeton.

[12] Schrijver, A. (2003). *Combinatorial Optimization. Polyhedra and Efficiency*, Springer, Berlin.

[13] Vazirani, V. (2001). *Approximation Algorithms*, Springer, Berlin.

[14] Wolsey, L. (1998). *Integer Programming*, John Wiley and Sons, New York.