Network & Integer Optimization: From Theory to Application

401-3902-21

Rico Zenklusen

Spring term 2023

This script has profited enormously from the input and help of many people. In particular, I want to thank Georg Anegg, Martin Nägele, Ivan Sergeev, and all students who provided very valuable feedback and help on a wide variety of aspects, which significantly increased the quality of this script.

Rico Zenklusen

Contents

1 Basics on Graphs

Graphs are a central structure in Discrete Mathematics with numerous real-world applications. This comes at no surprise, as graphs and networks are ubiquitous in everyday life and can be used to model transport networks, social networks, telecommunications networks, and neural networks, just to name a few examples. By exploiting the underlying graph-theoretical structures, problems in such networks can often be solved fast even for enormously large instances.

The aim of this chapter is to provide a very brief introduction to some of the basics linked to graphs and networks. After some motivational examples, we first introduce terminology and notation commonly used in Graph Theory. Moreover, we talk about the most common data structures to store graphs. This allows us to understand the input size of a problem that involves graphs and also clarifies how much time certain simple graph operations take. Finally, we discuss one of the most basic graph algorithms, namely breadth-first search, and some of its implications. This first algorithmic example also helps us to combine and better understand the previously introduced concepts, and is a first non-trivial example of how to analyze a graph algorithm.

We would like to emphasize that the terms "graph" and "network" are used interchangeably in many contexts. Sometimes, however, the term networks is used to describe graphs with additional information, for example on their edges or vertices.

1.1 Some motivational examples

In this introductory section, before formally defining graphs, we describe some classical problems that can be modeled and solved using graphs.

Example 1.1: Road network

Five cities, A, B, C, D, and E, are linked by the road network shown in Figure 1.1. The cities are represented by *vertices*, the streets as connections between two vertices, which are also called *edges*. The edges are labeled with the respective distances (in km) between the connected cities.

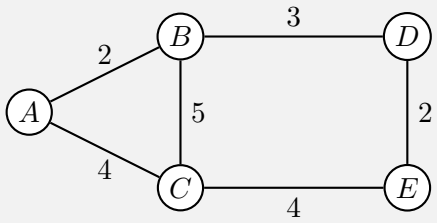


Figure 1.1: A road network represented as a graph.

One typical problem is to find the distance (i.e., the length of a shortest path) between any two cities as well as a corresponding path of that length. For example, the distance between city A and city E is 7 kilometers, and A-B-D-E is a possible path between A and E of this length.

The abstraction of the road network in Figure 1.1, consisting of vertices and edges, is a *graph*. The edges here are labeled with numbers—representing the distances in the example—so we are talking about a *weighted graph*. If Example 1.1 contained a one-way street, we could illustrate this in the graph by adding an arrow pointing in the allowed direction. Such an edge is called *directed edge*, or simply *arc*. If every edge is directed in a graph, we are talking about a *directed graph*.

Example 1.2: Utility network

Between five locations, a supply network is to be set up (e.g., for water, gas, electricity, or internet access). The goal is that each pair of locations is connected, where a connection can go through other intermediate stations. In Figure 1.2, the locations are represented as vertices and the possible connections as edges between the vertices. The numbers on the edges correspond to the installation cost for the respective connection. The problem is to find a minimum cost set of edges that connect all locations. The highlighted edges in Figure 1.2 show an optimal solution for this particular example.

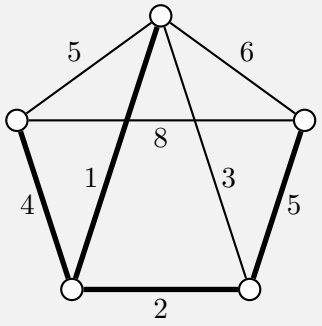


Figure 1.2: A utility network respresented as a graph.

Example 1.3: Assignment problem

A set of n jobs $\{j_1, \ldots, j_n\}$ are to be completed by m workers $\{a_1, \ldots, a_m\}$. Not every job can be completed by every worker: for each job, there is a given list of workers who are qualified to perform it. The question is whether it is possible to assign to each job one worker who is qualified for it. Each worker can be assigned to at most one job. Figure 1.3 shows a toy problem of this type with five jobs and six workers. The problem is represented as a graph. The top row of vertices corresponds to the jobs and the bottom one to the available workers. There is an edge between a job and a worker if and only if the job can be performed by the worker. The highlighted edges indicate a possible allocation assigning one worker to each job.

job	qualified workers
j_1	a_1, a_3, a_4
j_2	a_1, a_2, a_3, a_5
j_3	a_4, a_5, a_6
j_4	a_3, a_4
j_5	a_4, a_5, a_6

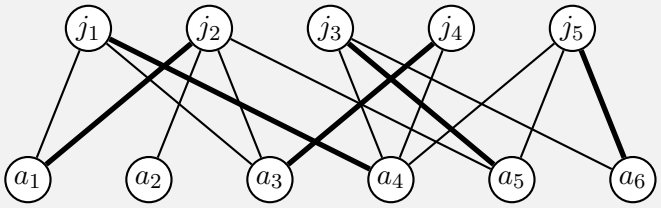


Figure 1.3: An assignment problem and its representation as a graph.

Example 1.4: Flow problem

In a system of water pipes there is a source s and a sink t. The system consists of junctions and pipes connecting these points. Each pipe can be used to let water flow in either one of the two directions. Moreover, each pipe has a maximum throughput, i.e., a maximum amount of water that can flow through the pipe per time unit. Junctions cannot store water, and no water is taken from or pumped into the system at junctions. Hence, for each junction, the water inflow must be equal to its outflow. A canonical optimization problem in this setting is to determine the largest so-called s-t flow, which is the maximum possible water flow—in terms of water per time unit—from s to t.
Figure 1.4 shows an example of such a pipe network. The edges represent pipes and the vertices represent junctions, except for the two labeled vertices, which represent the source s and the sink t. The black numbers on the edges, i.e., the ones on the right-hand side, are the known maximum throughput rates for the pipes. This concrete example allows a maximum s-t flow of 17 units. The blue numbers, which are to the left of the throughput values, indicate a possible flow in the direction of the blue arrows that realizes this maximum.

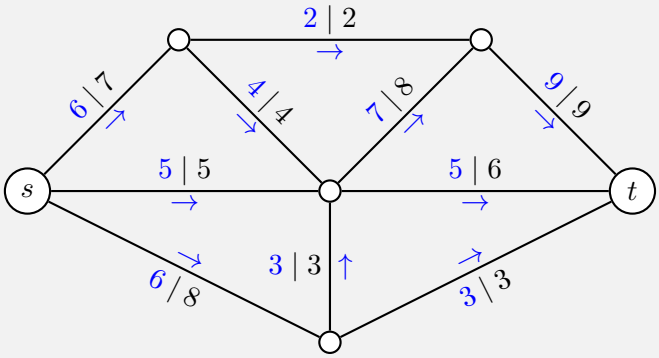


Figure 1.4: A flow problem together with a maximum s-t flow represented as a graph.

Example 1.5: Connection problem

In a telecommunications network, there is a central server in a network of locations. There are connections of different bandwidths between the locations. At certain stations, there are customers who would like to each have a connection of bandwidth 1 to the server. The task is to determine a maximum number of customers that can simultaneously be served by the server with a unit bandwidth each. The left part of Figure 1.5 shows such a problem where the server and locations are represented as vertices of a graph and the edges correspond to the connections. The numbers on the edges indicate the bandwidth of the respective connections and the tokens above vertices represent the clients. In this example situation, it turns out that a maximum of 4 clients can be connected to the server simultaneously. The right-hand side of Figure 1.5 shows one configuration serving 4 clients.

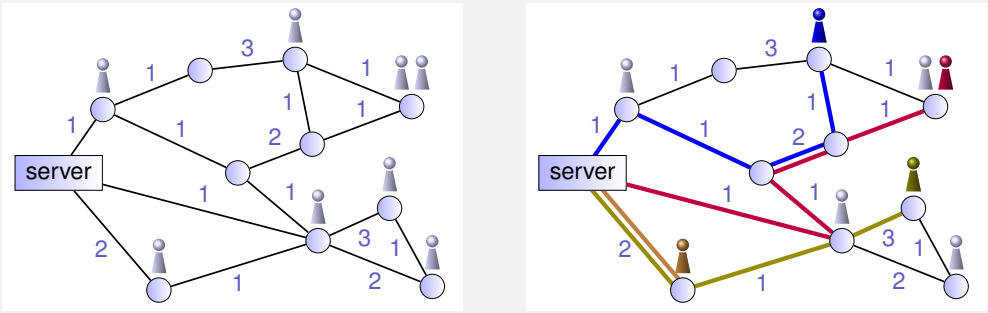


Figure 1.5: A connection problem and an optimal solution for it.

Example 1.6: Round trip

Our goal is to plan a trip through 20 given cities of the USA. It should be a round trip, i.e., the trip starts and ends in the same city. Driving times between any of the 20 cities are known upfront, and the task is to determine a road trip that minimizes the total driving time. Figure 1.6 shows an example.

Figure 1.6: A possible road trip through 20 given cities in the USA.

Here, the problem can be presented as a graph with 20 vertices, one per city, and all possible edges between these vertices. Each edge is assigned a value, representing the travel time between its endpoints. With this abstraction, the shortest round trip problem, also famously known as the Traveling Salesman Problem, can be formulated as follows: Find a shortest cycle in the graph that contains all vertices.

Example 1.7: Drilling circuit boards

One of the steps in the production process of an electronic circuit board is the drilling of holes. This is performed by automatic drilling machines, whose drill head wanders over the circuit board and drills one hole after the other. In order to optimize the time required for this task, the distance traveled by the drill head should be minimized. This problem is very similar to the round trip problem depicted in Example 1.6 and can be modeled in an analogous way. However, in contrast to the round trip problem, we do not need that the drill head returns back to the starting point after completion of the drilling. Figure 1.7 shows a section of a circuit board and a possible drilling order for that area.

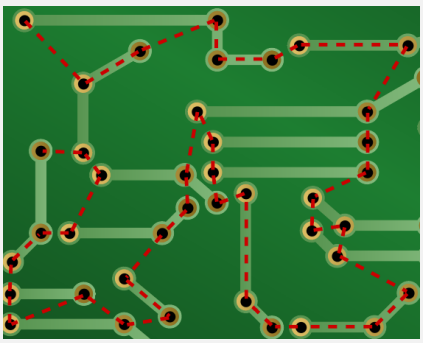
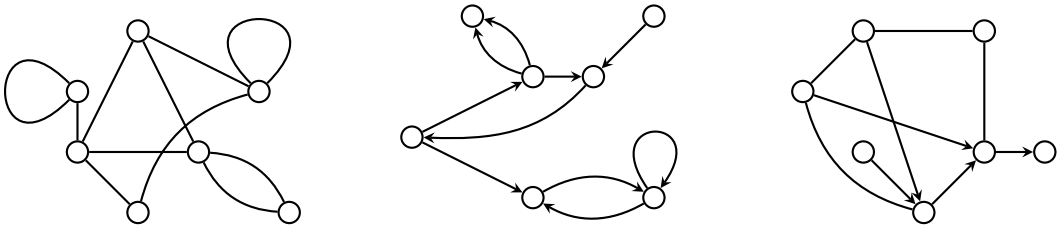


Figure 1.7: Section of a circuit board with a possible drilling order.

1.2 Basic terminology and notation

A graph consists of a set of *vertices* and connections between pairs of vertices, which are called *edges*. Edges can be *undirected* or *directed*, that is, oriented from one endpoint to another. A graph containing only undirected edges is an *undirected graph* (Figure 1.8a). If all edges are directed, we speak of a *directed graph* (Figure 1.8b). It is also possible that undirected and directed edges occur simultaneously. In this case we are talking about a *mixed graph* (Figure 1.8c).

Our main focus will be on directed and undirected graphs, which we will formally introduce shortly. In the context of graphs, various notations and terminology are used throughout the literature. We focus on one widespread formalization in the following.



(a) An undirected graph. (b) A directed graph. (c) A mixed graph.

Figure 1.8: Different types of graphs.

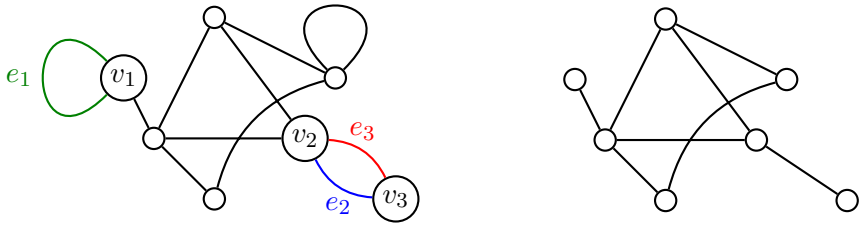
1.2.1 Undirected graphs

An *undirected* graph $G = (V, E)$ is a pair consisting of a finite set V of *vertices* and a finite set E of *edges*. Each edge $e \in E$ has two endpoints $u, v \in V$, which may be identical. The endpoints of an edge are denoted by $\text{endpoints}(e) = \{u, v\}$.

An edge $e \in E$ is called a *loop* if its two endpoints are identical, or in other words if $|\text{endpoints}(e)| = 1$. Two different edges $e_1, e_2 \in E$ are called *parallel* if $\text{endpoints}(e_1) = \text{endpoints}(e_2)$. An (undirected) graph without loops and parallel edges is called a *simple graph*. For example, in Figure 1.9a, we have

$$\text{endpoints}(e_1) = \{v_1\} \quad \text{and} \quad \text{endpoints}(e_2) = \text{endpoints}(e_3) = \{v_2, v_3\} \ ;$$

hence e_1 is a loop and the edges e_2 and e_3 are parallel. Thus, the graph in Figure 1.9a is not a simple graph. The graph in Figure 1.9b, on the other hand, has neither loops nor parallel edges and is thus simple.



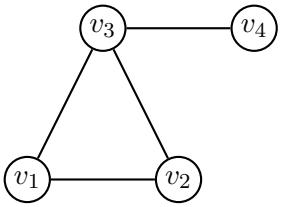
(a) A non-simple graph. (b) A simple graph.

Figure 1.9: Non-simple and simple graphs.

In simple graphs, we often describe an edge as the set of its endpoints, i.e.,

$$E \subseteq \binom{V}{2} := \{\{u, v\} \colon u, v \in V,\ u \neq v\} \ .$$

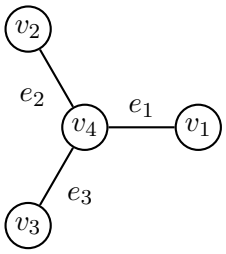
Figure 1.10 illustrates this notation, which is unambiguous in simple graphs, because for any set of endpoints there is at most one edge corresponding to it. In non-simple graphs, however, ambiguities may appear when referring to one of several parallel edges by its endpoints. Nevertheless, we also frequently use the above notation in the context of non-simple graphs in cases where there is little danger of confusion.



$$G = (V, E)$$
$$V = \{v_1, v_2, v_3, v_4\}$$
$$E = \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_1, v_3\}, \{v_3, v_4\}\}$$

Figure 1.10: Notation for simple graphs. Due to its convenience, this notation is also often used in the context of non-simple graphs.

An edge $e \in E$ and a vertex $v \in V$ are called *incident* if v is one of the endpoints of e, i.e., if $v \in \text{endpoints}(e)$. Two distinct vertices $u, v \in V$ are called *adjacent* (or *neighboring*) if there is an edge e with $\text{endpoints}(e) = \{u, v\}$. With the notation introduced above, u and v are adjacent if and only if $\{u, v\} \in E$. Figure 1.11 exemplifies the notions of incidence and adjacency.



v_1 is incident to e_1, but not to e_2 or e_3.

v_4 is adjacent to v_1, v_2, and v_3.

v_1 and v_2 are not adjacent.

Figure 1.11: Example of incidence and adjacency relationships.

1.2.2 Directed graphs

An *directed graph* $G = (V, A)$ consists of a finite set V of vertices and a finite set A of *directed edges*, which we call *arcs* to more easily distinguish them from their undirected counterparts. Every arc $a \in A$ has a *tail* $\text{tail}(a) \in V$ and a *head* $\text{head}(a) \in V$ and points from tail to head. In particular, the endpoints are given by $\text{endpoints}(a) = \{\text{tail}(a), \text{head}(a)\}$.

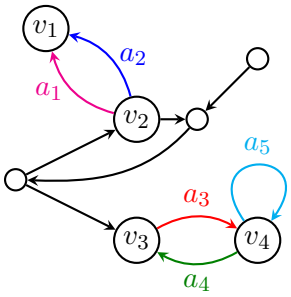
An arc a is a *loop* if $\text{tail}(a) = \text{head}(a)$. Two distinct arcs a_1 and a_2 are called *parallel* if $\text{tail}(a_1) = \text{tail}(a_2)$ and $\text{head}(a_1) = \text{head}(a_2)$. They are called *antiparallel* if $\text{tail}(a_1) = \text{head}(a_2)$ and $\text{head}(a_1) = \text{tail}(a_2)$. As for undirected graphs, a directed graph $G = (V, A)$ is called *simple* if it contains neither loops nor parallel arcs. However, a simple directed graph may contain antiparallel arcs. Figure 1.12 shows examples of a non-simple and a simple directed graph. In Figure 1.12a, we have

$$\left.\begin{aligned} \text{head}(a_1) = \text{head}(a_2) &= v_1 \\ \text{tail}(a_1) = \text{tail}(a_2) &= v_2 \end{aligned}\right\} \quad \Longrightarrow \quad a_1 \text{ and } a_2 \text{ are parallel,}$$

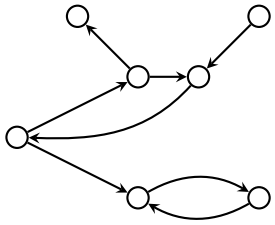
$$\left.\begin{aligned} \text{tail}(a_3) = \text{head}(a_4) &= v_3 \\ \text{head}(a_3) = \text{tail}(a_4) &= v_4 \end{aligned}\right\} \quad \Longrightarrow \quad a_3 \text{ and } a_4 \text{ are antiparallel, and}$$

$$\left.\begin{aligned} \text{head}(a_5) = \text{tail}(a_5) = v_4 \\ \text{or, equivalently: } \text{endpoints}(a_5) = \{v_4\} \end{aligned}\right\} \quad \Longrightarrow \quad a_5 \text{ is a loop.}$$

Hence, because the directed graph in Figure 1.12a has both parallel edges and moreover a loop, it is not simple. The directed graph in Figure 1.12b is simple.



(a) A non-simple directed graph.



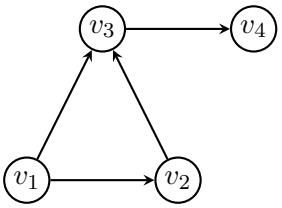
(b) A simple directed graph.

Figure 1.12: Non-simple and simple directed graphs.

Analogously to the undirected case, we can use a simplified notation for arcs that is unambiguous in simple graphs. More precisely, we can describe an arc $a \in A$ as an ordered pair $a = (u, v)$ of vertices, where $u = \text{tail}(a)$ and $v = \text{head}(a)$. With this notation it holds that

$$A \subseteq \{(u, v) \colon u, v \in V,\ u \neq v\} \ ,$$

where A is the arc set of a simple directed graph $G = (V, A)$. As before, this notation is unambiguous only if no parallel arcs exist. Similar to the undirected case, we nevertheless use this convenient notation also for directed graphs with parallel arcs, as long as there is no danger of ambiguity. Figure 1.13 illustrates this notation.



$$G = (V, A)$$
$$V = \{v_1, v_2, v_3, v_4\}$$
$$A = \{(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_3, v_4)\}$$

Figure 1.13: Notation for simple directed graphs. Due to its convenience, this notation is also used in the context of non-simple graphs.

1.2.3 Further basic notions

We continue with some very common and basic notation and terminology in the context of graphs. Because of the close relation between undirected and directed graphs, when it comes to terminology and notation, we discuss both cases together. In parenthesis, we indicate whether

the introduced notation is commonly used for undirected (undir.) graphs, directed (dir.) graphs, or both (undir./dir.).

- (undir./dir.) Edges that *cross* the subset of vertices $S \subseteq V$:
$$\delta(S) \coloneqq \begin{array}{l} \{\{u,v\} \in E \colon |\{u,v\} \cap S| = 1, u \neq v\} \ , \\ \{(u,v) \in A \colon |\{u,v\} \cap S| = 1, u \neq v\} \ . \end{array}$$
- (dir.) Arcs leaving the vertex set $S \subseteq V$:
$$\delta^+(S) \coloneqq \{(u,v) \in A \colon u \in S,\ v \notin S\} \ .$$
- (dir.) Arcs entering the vertex set $S \subseteq V$:
$$\delta^-(S) \coloneqq \{(u,v) \in A \colon u \notin S,\ v \in S\} \ .$$

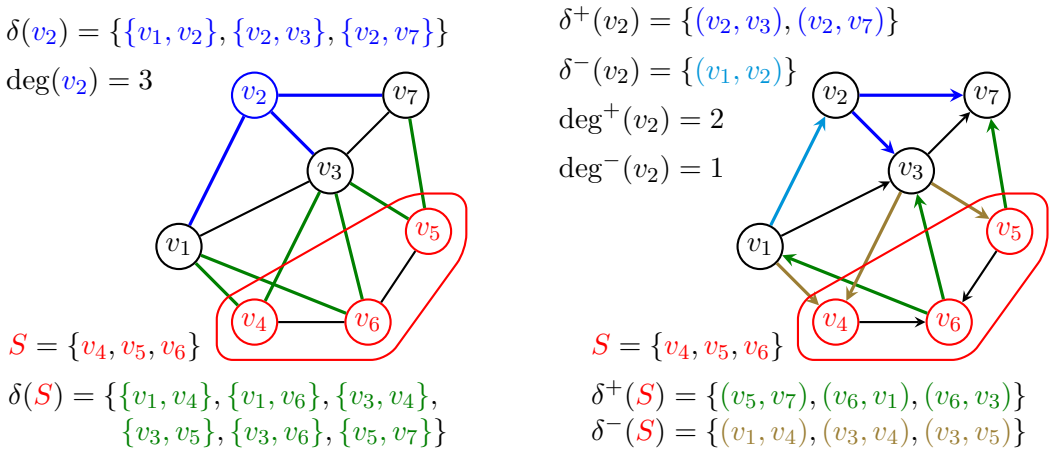
In particular, for a directed graph $G = (V, A)$ and a subset of vertices $S \subseteq V$, it always holds that $\delta(S) = \delta^+(S) \cup \delta^-(S)$.

For a single vertex $v \in V$, we use the following shorthands of the above notation:

$$\delta(v) \coloneqq \delta(\{v\}) \ , \quad \delta^+(v) \coloneqq \delta^+(\{v\}) \ , \quad \text{and} \quad \delta^-(v) \coloneqq \delta^-(\{v\}) \ .$$

- (undir./dir.) *Degree* of v: $\deg(v) \coloneqq |\delta(v)| + 2 \cdot |\{\text{loops at } v\}|$.
- (dir.) *Outdegree* of v: $\deg^+(v) \coloneqq |\delta^+(v)| + |\{\text{loops at } v\}|$.
- (dir.) *Indegree* of v: $\deg^-(v) \coloneqq |\delta^-(v)| + |\{\text{loops at } v\}|$.

Figure 1.14 exemplifies the introduced notation.



(a) Undirected graph.

(b) Directed graph.

Figure 1.14: Crossing edges and vertex degrees.

Having introduced the most important notations for dealing with graphs, we now present two simple yet very useful properties of the vertex degrees occurring in a graph.

Property 1.8: Sum of all vertex degrees

In an undirected graph $G = (V, E)$, the sum of the degrees of all vertices is even.

Proof of Property 1.8. If we sum up the vertex degrees, each edge is counted twice, once for each of its endpoints. Consequently

$$\sum_{v \in V} \deg(v) = 2|E| \ ,$$

implying that the sum of all vertex degrees is even. □

The proof of Property 1.8 is an example of a proof technique known as double counting, i.e., the same quantity gets counted in two different ways. A direct implication of Property 1.8 is the so-called *handshaking lemma*.

Property 1.9: Handshaking lemma

In every undirected graph, the number of odd-degree vertices is even.

Proof of Property 1.9. We partition the vertex set $V = V_1 \dot\cup V_2$, where

$$V_1 \coloneqq \{v \in V : \ \deg(v) \text{ odd}\} = \{\text{odd-degree vertices}\} \ , \text{ and}$$
$$V_2 \coloneqq \{v \in V : \ \deg(v) \text{ even}\} = \{\text{even-degree vertices}\} \ .$$

We now have

$$\sum_{v \in V_1} \deg(v) = \sum_{v \in V} \deg(v) - \sum_{v \in V_2} \deg(v) \ .$$

On the right-hand side of the last equation, the first sum is even by Property 1.8, and the second one is even as well because each summand is even. Thus it follows that the sum on the left-hand side is even. Notice that each term in the sum on the left-hand side is odd by definition of V_1; hence, the number of summands must be even, i.e., $|V_1|$ is even. □

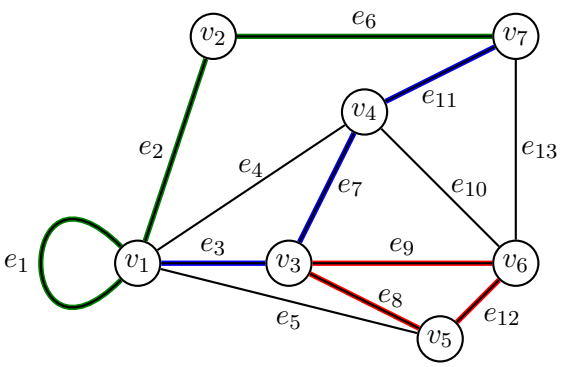
The reason why Property 1.9 is also known as the handshaking lemma is the following: Property 1.9 states that at every party the number of people who shake hands with an odd number of guests has to be even. To see this, consider a graph that has the party's guests as its vertices. Two people are connected by an edge if and only if they shake hands. Thus, the degree of a vertex is the number of guests with whom the corresponding person shook hands. The statement now follows immediately from Property 1.9.

A *walk* in an undirected graph $G = (V, E)$ is a sequence $v_1, e_1, v_2, \dots, v_{k-1}, e_{k-1}, v_k$ (possibly with repetitions) consisting of $k \in \mathbb{Z}_{\geq 1}$ vertices $v_1, \dots, v_k \in V$ and $k-1$ edges $e_1, \dots, e_{k-1} \in E$ with $e_i = \{v_i, v_{i+1}\}$ for all $i \in [k-1]$, where we recall that $[\ell] \coloneqq \{1, \dots, \ell\}$ for $\ell \in \mathbb{Z}_{\geq 1}$, and we use the convention $[0] = \emptyset$. We call such a walk a *walk between* v_1 *and* v_k. For an undirected walk, we consider the description $v_1, e_1, v_2, \dots, v_{k-1}, e_{k-1}, v_k$ and the description $v_k, e_{k-1}, v_{k-1}, \dots, v_2, e_1, v_1$ in the opposite direction as equivalent, i.e., they describe the same walk.

In a directed graph $G = (V, A)$, a walk is a sequence $v_1, a_1, v_2, \dots, v_{k-1}, a_{k-1}, v_k$ (possibly with repetitions) consisting of vertices $v_1, \dots, v_k \in V$ and arcs $a_1, \dots, a_{k-1} \in A$ with $a_i =$

$(v_i, v_{i+1}) \in A$ for all $i \in [k-1]$. In the directed case we are talking about a *walk from* v_1 *to* v_k or a v_1-v_k *walk*. This notation is also used for undirected walks, where in undirected graphs, each v_1-v_k walk is also a v_k-v_1 walk.

In both cases we define the *length* of a given walk with k vertices—where a vertex is counted as many times as it appears in the walk—as $k-1$, which is the number of edges in the walk, also counted with multiplicities if an edge appears multiple times. A walk is called a *path* if it contains no vertex more than once, i.e., $v_i \neq v_j$ for all $i, j \in [k]$ with $i \neq j$. A walk is called *closed*, if $v_1 = v_k$. A closed walk in which all vertices except for start and endpoint are pairwise distinct is called a *cycle*. Figure 1.15 illustrates the new terminology.



$v_1, e_1, v_1, e_2, v_2, e_6, v_7, e_6, v_2$ is a walk of length 4, but not a path, since v_1 and v_2 appear multiple times.

$v_1, e_3, v_3, e_7, v_4, e_{11}, v_7$ is a v_1-v_7-path of length 3.

$v_3, e_8, v_5, e_{12}, v_6, e_9, v_3$ is a closed walk with pairwise distinct vertices (up to start and endpoint), hence it is a cycle of length 3.

Figure 1.15: Walks, paths, closed walks, cycles, and their lengths.

To simplify the notation, we often denote a path or cycle simply by the set of edges it traverses, for example, a cycle C in a graph $G = (V, E)$ is represented as an edge set $C \subseteq E$. In addition, '$+$' and '$-$' are used for adding or deleting a single element to or from a set, i.e., $S + u - w := (S \cup \{u\}) \setminus \{w\}$.

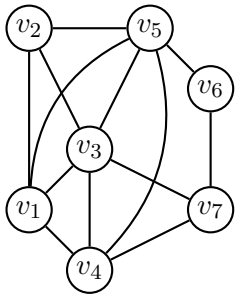
A *subgraph* of an undirected graph $G = (V, E)$ is a graph $H = (W, F)$ with $W \subseteq V$ and $F \subseteq \{e \in E \colon e \subseteq W\}$, i.e., F is a subset of those edges of G that have both endpoints in W. For a vertex set $W \subseteq V$, the subgraph *induced* by W is the subgraph of $G = (V, E)$ defined by

$$G[W] = (W, E[W]) \ ,$$

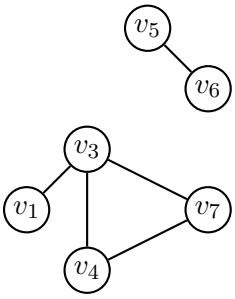
where $E[W] := \{e \in E \colon e \subseteq W\}$ is the set of all edges of G with both endpoints in W. The definitions of subgraph and induced subgraph are analogous for directed graphs. Figure 1.16 exemplifies these notions for undirected graphs.

Exercise 1.10: Seating arrangements

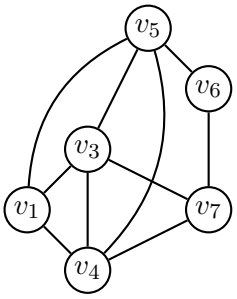
You invite n guests over for dinner. For each pair of guests it is known whether they are friends or not. The task is to create a seating plan for a round table such that guests sitting next to each other are friends. Formulate this problem as the graph-theoretic problem of finding a certain cycle in a well-defined graph.



(a) A graph $G = (V, E)$.



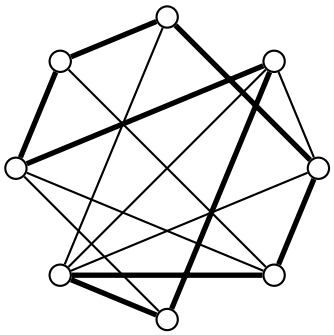
(b) A subgraph of G.



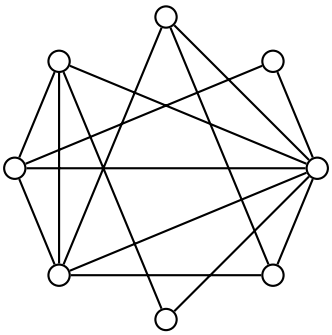
(c) Induced subgraph $G[V \setminus \{v_2\}]$.

Figure 1.16: Graphs, subgraphs, and induced subgraphs.

Solution
We construct a graph $G = (V, E)$ in which each vertex corresponds to a guest. Two vertices are connected by an edge if and only if the two corresponding guests are friends. A desired seating plan now corresponds to a cycle in G that contains all the vertices in V exactly once (see Figure 1.17a). Such a cycle is called a *Hamilton cycle*. Of course, not every graph contains a Hamilton cycle. For example, for the graph in Figure 1.17b, it can be shown that it does not contain a Hamilton cycle.



(a) A graph with a Hamilton cycle.

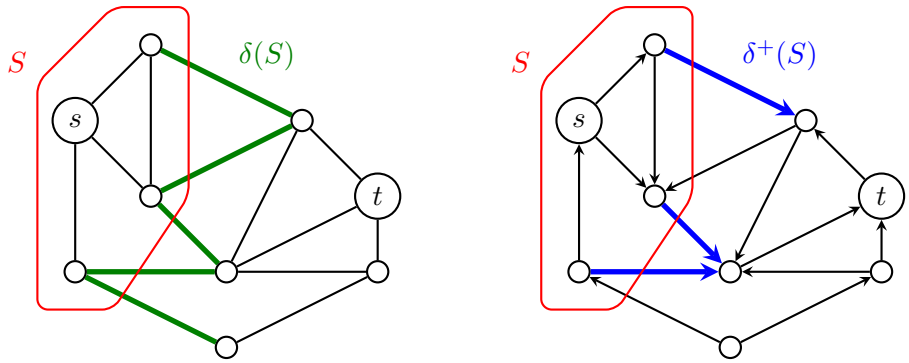


(b) A graph with no Hamilton cycle.

Figure 1.17: Hamilton cycles in graphs.

A *cut* in a graph $G = (V, E)$ is a nonempty subset $S \subsetneq V$ of vertices, where $S \subsetneq V$ is a shorthand notation for $S \subseteq V$ and $S \neq V$. One can also imagine a cut as a non-trivial partition of the vertex set V into the sets S and $V \setminus S$. This partition is non-trivial because both S and $V \setminus S$ are nonempty. For this reason, a cut is sometimes also written as the pair $(S, V \setminus S)$. Cuts are defined analogously in directed and undirected graphs. For a cut S in an undirected graph, the edges in $\delta(S)$ are also referred to as the *edges in the cut* S or as the edges *crossing* S. In the case of a directed graph, we denote the edges in $\delta^+(S)$ as the *edges in the cut* S. For two distinct vertices $s, t \in V$, an *s-t cut* in G is a cut $S \subseteq V$ such that $s \in S$ and $t \notin S$. In other words, an s-t cut separates s from t. Figure 1.18 illustrates the difference between the undirected and the

directed case.



(a) Undirected graph. (b) Directed graph.

Figure 1.18: An s-t cut S and the edges in the cut $\delta(S)$ (for the undirected case) and $\delta^+(S)$ (for the directed case).

Exercise 1.11

Let $G = (V, A)$ be a directed graph, S a cut in G, and $C \subseteq A$ a directed cycle in G. Show that $|\delta^+(S) \cap C| = |\delta^-(S) \cap C|$.

Solution

Let $H = (V, C)$ be the subgraph of G only containing the arcs in C. For each vertex set $W \subseteq V$, let $\delta^+_H(W) := \delta^+(W) \cap C$ and $\delta^-_H(W) := \delta^-(W) \cap C$. Using this notation, we therefore need to prove $|\delta^+_H(S)| = |\delta^-_H(S)|$. First we show

$$|\delta^+_H(S)| - |\delta^-_H(S)| = \sum_{v \in S} \left(\deg^+_H(v) - \deg^-_H(v)\right) \quad . \tag{1.1}$$

This equality can be verified by checking that each arc $a \in C$ contributes to both sides of the equation equally. An easy way to perform this check is by distinguishing the following three types of arcs: (i) $a \in \delta^+_H(S)$, (ii) $a \in \delta^-_H(S)$, and (iii) $a \notin \delta^+_H(S) \cup \delta^-_H(S)$.
Because C is a directed cycle, every vertex in H has the same indegree as outdegree, i.e., $\deg^+_H(v) = \deg^-_H(v)$ for all $v \in V$. Consequently, every single summand on the right-hand side of (1.1) is equal to zero. Hence, the difference on the left-hand side is zero, too. It follows that $|\delta^+_H(S)| = |\delta^-_H(S)|$, as desired.

1.3 Data structures for graphs

One of our main goals is to develop and analyze algorithms for graph optimization problems. To this end, we need to clarify how a graph is stored in a computer and how graph-related data can be accessed. This will both clarify the input size of a graph and the time we need to perform basic graph operations, both of which are key to talk about the running time of graph algorithms.

In this section we address these points by talking about data structures for graphs.

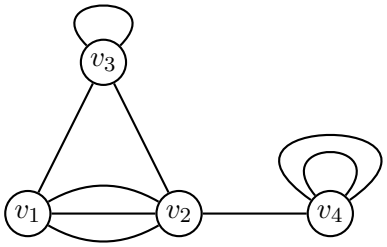
Two of the most widely used data structures for graphs are the representation as an *adjacency matrix* (sometimes called *neighborhood matrix*) and the representation as an *incidence list*. Unless explicitly indicated otherwise, we will always assume that the underlying data structure is an incidence list. However, we start by presenting the adjacency matrix before expanding on the incidence list, due to its simplicity. The choice of data structures is a crucial and often highly non-trivial step in algorithmics that comes with various trade-offs. Introducing another graph data structure besides the incidence list allows us to discuss and compare advantages and disadvantages of these two data structures in comparison to each other, and to highlight some example trade-offs faced when choosing a graph data structure.

1.3.1 The adjacency matrix

The *adjacency matrix* of an undirected graph $G = (V, E)$ is the symmetric matrix $M \in \mathbb{Z}_{\geq 0}^{V \times V}$ whose entries for all $u, v \in V$ are defined by

$$M(u,v) \coloneqq \begin{cases} |\{e \in E \colon \text{endpoints}(e) = \{u,v\}\}| & \text{if } u \neq v \ , \\ 2 \cdot |\{\text{loops at } v\}| & \text{if } u = v \ . \end{cases}$$

Figure 1.19 shows the adjacency matrix of an undirected graph with 4 vertices.



$$M = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{pmatrix} 0 & 3 & 1 & 0 \\ 3 & 0 & 1 & 1 \\ 1 & 1 & 2 & 0 \\ 0 & 1 & 0 & 4 \end{pmatrix} \end{array}$$

Figure 1.19: The adjacency matrix of an undirected graph.

The adjacency matrix of a directed graph $G = (V, A)$ is the matrix $M \in \mathbb{Z}_{\geq 0}^{V \times V}$ whose entries for all $u, v \in V$ are given by

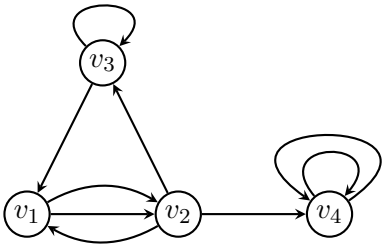
$$M(u,v) \coloneqq |\{a \in A \colon \text{tail}(a) = u, \text{head}(a) = v\}| \ .$$

Figure 1.20 shows the adjacency matrix of a directed graph with 4 vertices. As the example in Figure 1.20 highlights, the adjacency matrix of a directed graph is not necessarily symmetric.

For simple graphs, the corresponding adjacency matrices are $\{0,1\}$-matrices, i.e., matrices where each entry is either 0 or 1. In addition, for simple graphs, all entries on the diagonal of the adjacency matrix are 0.

1.3.2 The incidence list

An *incidence list* contains for each vertex a doubly linked list of those (directed or undirected) edges incident to the corresponding vertex. To access these lists, we can use an array indexed



$$M = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{pmatrix} 0 & 2 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix} \end{array}$$

Figure 1.20: The adjacency matrix of a directed graph.

by the vertices of the graph, which contains the pointers to the respective lists. More precisely, these pointers point to the first element of the respective lists. For each vertex v, accessing the first element of the list corresponding to v requires constant time. Figure 1.21 schematically shows the incidence list of a directed graph.

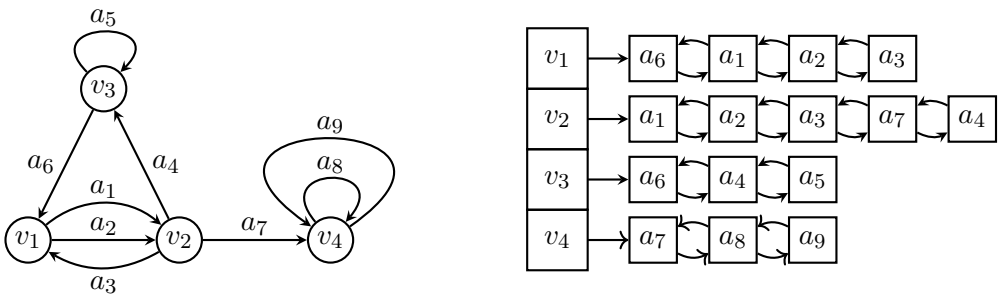


Figure 1.21: The incidence list of a directed graph.

For undirected edges e we use a data structure in which the set $\mathrm{endpoints}(e)$ is stored and can be queried. In the case of an arc a, it is possible to query $\mathrm{tail}(a)$ and $\mathrm{head}(a)$ in constant time. Further information about the edges, for example weights, can also be stored in the data structure for the edges.

1.3.3 Comparison: adjacency matrix and incidence list

Space requirements We consider a simple undirected graph $G = (V, E)$ (the directed case is analogous). Let $n := |V|$ be the number of vertices and $m := |E|$ the number of edges of G. An important difference between the adjacency matrix and the incidence list of G is the space required:

- Adjacency matrix: $\Theta(n^2)$.
- Incidence list: $\Theta(m + n)$.

Thus, for simple graphs the incidence list is typically a more compact encoding of the graph, because in this case $m = O(n^2)$. The reason for the significantly larger space requirement of the adjacency matrix is that even if two vertices are not connected by an edge, it stores a 0 to

capture this information. This difference is particularly pronounced if the graph is *sparse*, that is, if it has only few edges, for example $m = O(n)$. Here, the adjacency matrix needs quadratic space in n whereas the incidence list only requires linear space. This difference is crucial when dealing with graphs with millions of vertices but constant average degree—a regime that is very common in many applications.

Algorithmic complexity of some operations Table 1.1 lists, for both the adjacency matrix and the incidence list, the running times of several simple computational tasks in an undirected graph $G = (V, E)$, namely calculating the degree of a vertex or listing all incident edges, deciding whether a particular edge is part of the graph, and computing the number of edges $|E|$.

	$\deg(v)$ or $\delta(v)$	$\exists?\{u, v\} \in E$	$\lvert E\rvert$
adjacency matrix	$O(n)$	$O(1)$	$O(n^2)$
incidence list	$O(\deg(v))$	$O(\min\{\deg(u), \deg(v)\})$	$O(m + n)$

Table 1.1: Running time of simple computations in graphs.

Exercise 1.12

Let $G = (V, E)$ be a graph given by an incidence list and let $u, v \in V$ two vertices. Show that in time $O(\min\{\deg(u), \deg(v)\})$ it can be decided whether G contains an edge $\{u, v\}$.

We recall that, unless explicitly stated otherwise, we will always assume that graphs are given as incidence lists. The size of a graph with n vertices and m edges is therefore $\Theta(m+n)$. Thus, an algorithm whose input is a single such graph is polynomial if and only if its running time is upper bounded by a polynomial in $m + n$.

1.4 Breadth-first search (BFS): shortest paths and more

We now discuss one of the best-known search methods on graphs, namely breadth-first search, or BFS for short. To be exact, breadth-first search is an algorithmic idea that can be adapted to solve many different problems. We introduce it in the context of finding shortest paths in undirected graphs. Later we will see how the algorithm can be modified for directed graphs.

Let $G = (V, E)$ be an undirected graph. We define a distance function $d\colon V \times V \to \mathbb{Z}_{\geq 0} \cup \{\infty\}$ as follows:

$$d(u, v) := \min\{|P|\colon P \subseteq E,\ P \text{ is a walk between } u \text{ and } v\} \qquad \forall u, v \in V\ , \tag{1.2}$$

i.e., $d(u, v)$ is the length of the shortest u-v-walk in G. If there is no walk between u and v, then $d(u, v)$, as defined above, is the minimum over an empty set. For this case we set by convention $d(u, v) = \infty$. Notice that one can impose in (1.2) the additional requirement that P must be a path (instead of a walk) without changing the definition. Indeed, this would not change the value

of $d(u, v)$ because a shortest walk is always a path. Figure 1.22 shows a graph $G = (V, E)$ that contains next to each vertex v the distance to the fixed vertex v_1, i.e., the length of a shortest path to v_1.

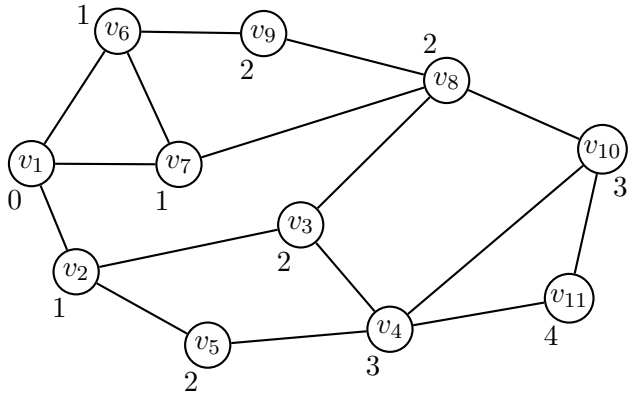


Figure 1.22: The number next to each vertex gives the distance to v_1.

Breadth-first search is an algorithm that efficiently calculates the distances $d(v_1, v)$ for all vertices $v \in V$ for a fixed vertex $v_1 \in V$. Algorithm 1 describes BFS in pseudocode.

Algorithm 1: Breadth-first search: computation of distances from a fixed vertex v_1

Input: $G = (V, E)$, $v_1 \in V$.
Output: $d(v_1, v)$ for all $v \in V$.

1. **Initialization:**
   $d(v_1, v) = \begin{cases} \infty & \text{if } v \in V \setminus \{v_1\}, \\ 0 & \text{if } v = v_1. \end{cases}$
   $L = \{v_1\}$. `// vertices to be processed`
   $k = 1$. `// shortest possible assignable distance`
2. **while** $(L \neq \emptyset)$ **do:**
   $L_{\text{new}} = \emptyset$.
   for $v \in N(L) := \{u \in V : \exists w \in L \text{ with } \{w, u\} \in E\}$ **do:**
   if $d(v_1, v) = \infty$ **then:**
   $d(v_1, v) = k$.
   $L_{\text{new}} = L_{\text{new}} \cup \{v\}$.
   $k = k + 1$.
   $L = L_{\text{new}}$.
3. **return** $d(v_1, v)$ for all $v \in V$.

The idea of breadth-first search is simple: We first consider the vertex v_1. For this vertex, we know that $d(v_1, v_1) = 0$, because an empty walk has no edges and therefore has length 0. In the following, we consider all vertices in the *neighborhood* $N(\{v_1\})$ of v_1, that is, all vertices

connected to v_1 by an edge. For these vertices v we set $d(v_1, v) = 1$. In the next step we consider all vertices that are adjacent to one of the vertices of distance 1 and have not yet been considered. For these vertices v we set $d(v_1, v) = 2$, and so on. Table 1.2 shows the sets L and $N(L)$ for each iteration of the while-loop Algorithm 1 runs through for the graph in Figure 1.22. The set L contains the vertices newly considered in the previous iteration, $N(L)$ their neighbors.

k	L	$N(L)$
1	$\{v_1\}$	$\{v_2, v_6, v_7\}$
2	$\{v_2, v_6, v_7\}$	$\{v_1, v_3, v_5, v_6, v_7, v_8, v_9\}$
3	$\{v_3, v_5, v_8, v_9\}$	$\{v_2, v_3, v_4, v_6, v_7, v_8, v_9, v_{10}\}$
4	$\{v_4, v_{10}\}$	$\{v_3, v_4, v_5, v_8, v_{10}, v_{11}\}$
5	$\{v_{11}\}$	$\{v_4, v_{10}\}$
6	$\{\}$	$\{\}$

Table 1.2: Breadth-first search table for the graph in Figure 1.22.

In the following we show that the function d computed by breadth-first search indeed corresponds to the distances from v_1. In addition, we show that the runtime of BFS is $O(m + n)$, where $n := |V|$ and $m := |E|$, which is linear in the size of the graph.

Theorem 1.13: Correctness of breadth-first search

For every graph $G = (V, E)$ and every vertex $v_1 \in V$, Algorithm 1 calculates the values $d(v_1, v)$ correctly for all $v \in V$.

Proof. To avoid confusion, let $d'(v_1, v)$ for $v \in V$ be the distances computed by Algorithm 1, and d the true distances in G as defined in (1.2). Furthermore, let $D'_\ell = \{v \in V : d'(v_1, v) = \ell\}$ and $D_\ell = \{v \in V : d(v_1, v) = \ell\}$ for all $\ell \in \mathbb{Z}_{\geq 0}$.

We show for every $k \in \mathbb{Z}_{\geq 0}$ that Algorithm 1, after k iterations of the while-loop, computed the set of all vertices $v \in V$ satisfying $d(v_1, v) \leq k$ as well as their distances correctly, i.e., $D_\ell = D'_\ell$ for all $\ell \leq k$. For this we use induction on k. For $k = 0$ the statement is certainly true, because only $v = v_1$ satisfies $d(v_1, v) = 0$ and only $v = v_1$ satisfies $d'(v_1, v) = 0$, So, suppose the statement holds for iteration k and consider iteration $k+1$ of the while-loop. In this iteration, we set $d'(v_1, v) = k + 1$ for all v that are adjacent to at least one vertex in $D'_k = D_k$ and have not yet been considered, hence $d'(v_1, v) = \infty$. Consequently,

$$D'_{k+1} = \{v \in V : d(v_1, v) > k, v \in N(D_k)\} \ .$$

We need to show that $D'_{k+1} = D_{k+1}$. For this we prove both inclusions $\subseteq$ and $\supseteq$.

$D'_{k+1} \subseteq D_{k+1}$: Let $v \in D'_{k+1}$. Because $v \in N(D_k)$, it holds that $d(v_1, v) \leq k + 1$: At least one neighbor w of v has distance k from v_1, thus there exists a v_1-w walk of length at most k. This walk can be extended to a v_1-v walk of length at most $k + 1$, which yields $d(v_1, v) \leq k + 1$. In addition, we have $d(v_1, v) > k$ because, due to the inductive

assumption, all vertices v with $d(v_1, v) \leq k$ have already been considered. Together it follows that $d(v_1, v) = k + 1$, so $v \in D_{k+1}$.

$D_{k+1} \subseteq D'_{k+1}$: Let $v \in D_{k+1}$. Hence, there is a v_1-v walk of length $k + 1$ with vertices $v_1, \ldots, v_{k+1}, v$. It follows that $v_{k+1} \in D_k$ needs to hold; for otherwise we could construct a v_1-v-path of length $< k + 1$. Thus, $v \in N(D_k)$, and consequently $v \in D'_{k+1}$.

This completes the inductive step and thus implies the statement. □

Theorem 1.14: Running time of breadth-first search

For every graph $G = (V, E)$ and every vertex $v_1 \in V$, Algorithm 1 has a running time bounded by $O(m + n)$.

Proof. The initialization in the first step of Algorithm 1 requires $O(n)$ time. For the second step, we first determine the time needed to compute the neighborhoods $N(L)$ calculated at the beginning of the for-loop. Let $L_1 = \{v_1\}, L_2, \ldots, L_{k-1}, L_k = \emptyset$ be the sets L occurring during the algorithm. Then L_i contains exactly those vertices of V that have distance $i - 1$ from v_1. Thus, $L_1, \ldots, L_k$ is a partition of all vertices that are in the same connected component as v_1.

The calculation of $N(L)$ takes $O\left(|L| + \sum_{v \in L} \deg(v)\right)$ time, because all neighbors of vertices in L are explored. Thus, all neighborhoods $N(L_1), \ldots, N(L_k)$ can be determined in a total running time of

$$O\left(\sum_{i=1}^{k} \left(|L_i| + \sum_{v \in L_i} \deg(v)\right)\right) = O\left(|V| + \sum_{v \in V} \deg(v)\right) = O(n + m) \ ,$$

where we use that $L_1, \ldots, L_k$ is a partition of V, and that the sum of all vertex degrees is exactly $2m$ (see proof of Property 1.8). Furthermore, each vertex $v \in V$ is contained in at most 3 different neighborhoods $N(L_j)$, namely in the iteration i such that $v \in L_i$, as well as in the one before and after. Indeed, a vertex can only be a neighbor of vertices with either the same distance to v_1 or with a distance to v_1 differing from $d(v_1, v)$ by exactly one unit. Thus, $\sum_{i=1}^{k} |N(L_i)| = O(n)$, and because every operation within the for-loop only requires constant time per iteration, the total running time of all operations within the for-loop is bounded by $O(n)$. The output of the distances $d(v_1, v)$ in the last step requires $O(n)$ time. Overall, the running time is therefore bounded by $O(m + n)$, as desired. □

Algorithm 1 can easily be adapted for directed graphs $G = (V, A)$. For this we define the *out-neighborhood* $N^+(L)$ as

$$N^+(L) = \{u \in V : \exists w \in L \text{ with } (w, u) \in A\} \ .$$

It suffices to replace $N(L)$ by $N^+(L)$ in Algorithm 1 to adapt it to the directed case.

1.4.1 Connectivity and connected components

A canonical application of BFS is to study the connectivity of a graph.

Definition 1.15: Connectivity in undirected graphs

Let $G = (V, E)$ be an undirected graph.

(i) Two vertices $s, t \in V$ are called *connected* in G if G contains an s-t walk.

(ii) The graph G is callled *connected* if each pair of vertices in G is connected.

With breadth-first search it is easy to check if a graph $G = (V, E)$ is connected. It suffices to perform a single breadth-first search with an arbitrary starting vertex $v_1 \in V$, because G is connected if and only if $d(v_1, v) < \infty$ holds for all $v \in V$.

In directed graphs $G = (V, A)$, connectivity can be defined analogously by "forgetting" the directions of the arcs. Moreover, there is a stronger notion of connectivity for directed graphs.

Definition 1.16: Connectivity in directed graphs

Let $G = (V, A)$ be a directed graph.

(i) G is called *connected* if the undirected graph G', obtained from G by ignoring arc directions, is connected.

(ii) Let $s, t \in V$. The vertex t can be *reached* from s in G if G contains a directed s-t walk.

(iii) The graph G is called *strongly connected* if every vertex in G can be reached from every other vertex.

Strong connectivity (of a directed graph) can also be checked through BFS, using only two BFS calls. To this end, we fix an arbitrary vertex v_1 and start BFS with v_1 as starting vertex. This run allows us to determine whether every vertex can be reached from v_1, that is, if $d(v_1, v) < \infty$ for all $v \in V$. If a vertex cannot be reached from v_1, then G is not strongly connected. Otherwise, we call BFS a second time with starting vertex v_1 but this time on the graph $\overline{G}$ obtained from G by reversing all arc directions. This allows us to determine whether in G, the vertex v_1 can be reached from all other vertices. If not, then G is not strongly connected. Otherwise, G is strongly connected. Indeed, if there is a vertex from which any other vertex can be reached and that can be reached from any other vertex, then the graph must be strongly connected. This follows from the fact that for any two vertices $u, w \in V$ there is a u-w walk because such a walk can be obtained by concatenating a u-v_1 walk with a v_1-w walk.

Definition 1.17: Connected components in undirected graphs

Let $G = (V, E)$ be an undirected graph. A *connected component* of G is an induced subgraph $G[W]$ such that $G[W]$ is connected and $W \subseteq V$ is maximal with respect to this property, i.e., for every $X \subseteq V$ with $X \supsetneq W$ the graph $G[X]$ is not connected.

We emphasize the crucial notion of maximality, which is ubiquitous in Mathematical Optimization. Namely, a set W is *maximal* with respect to a particular property if there is no strict superset of W that also fulfills the property. This does not necessarily mean that W is a set of

maximum cardinality among all the sets with the property. The converse, however, is true: If a set among all sets with a given property has maximum cardinality, then it is also maximal with respect to that property.

Each graph can be decomposed into its connected components. Figure 1.23 shows a graph with four connected components.

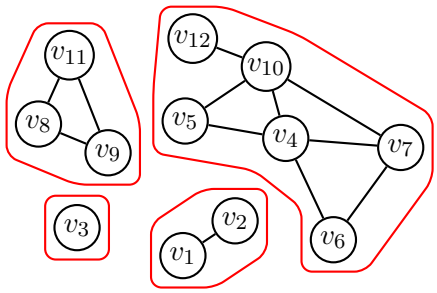


Figure 1.23: The above graph G has four connected components: $G[\{v_1, v_2\}]$, $G[\{v_3\}]$, $G[\{v_4, v_5, v_6, v_7, v_{10}, v_{12}\}]$, and $G[\{v_8, v_9, v_{11}\}]$.

Using BFS, one can also find the connected components of a graph. Starting from a starting vertex v_1, breadth-first search reaches exactly those vertices that are in the same connected component as v_1. Indeed, the vertices in the same connected component are precisely those that can be reached from v_1, i.e., those vertices v satisfying $d(v_1, v) < \infty$. So, if V_1 is the set of vertices reached by the breadth-first search, then $G[V_1]$ is a connected component of G. We can repeat this procedure for another starting vertex $v_2 \in V \setminus V_1$ to find a second connected component $G[V_2]$, and so on. A crude way to bound the running time of this procedure is by observing that we make $O(n)$ calls to BFS—because a graph has at most $n := |V|$ connected components—each of which uses $O(n + m)$ time by Theorem 1.14. This leads to an overall running time bound of $O(n(m + n))$.

Exercise 1.18: Running time improvement

Show that the connected components of an undirected graph $G = (V, E)$ can be found in running time $O(m + n)$. To achieve this, improve the running time analysis presented above and, if necessary, adapt BFS (Algorithm 1) for this setting.

We can analogously define connected components in directed graphs by disregarding arc directions. Again, for directed graphs, there is moreover a natural stronger notion of connected components.

Definition 1.19: Connected components in directed graphs

Let $G = (V, A)$ be a directed graph.

(i) The *connected components* of G are the connected components of the undirected graph G', which results from G by ignoring arc directions.

(ii) A *strongly connected component* of G is an induced directed subgraph $G[W]$ such that $G[W]$ is strongly connected and $W \subseteq V$ is maximal with respect to this property.

Figure 1.24 shows a directed graph with five strongly connected components.

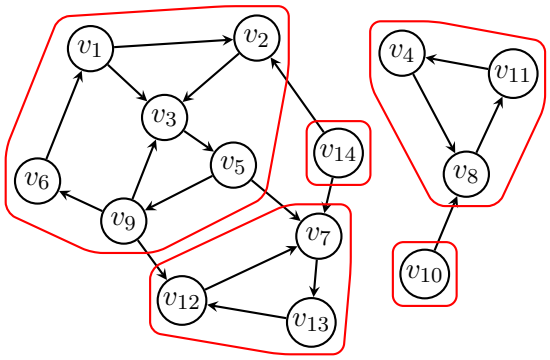


Figure 1.24: A graph G with two connected components and five strongly connected components (in red): $G[\{v_{10}\}]$, $G[\{v_1, v_2, v_3, v_5, v_6, v_9\}]$, $G[\{v_4, v_8, v_{11}\}]$, $G[\{v_7, v_{12}, v_{13}\}]$, and $G[\{v_{14}\}]$.

2 Some Classical Efficiently Solvable Graph Optimization Problems

In this chapter, we briefly describe some basic combinatorial optimization problems on graphs. All of these problems can be solved in polynomial time, i.e., they lie in the complexity class P. More precisely, these problems can even be solved in strongly polynomial time and also admit algorithms that are very fast in practice. Moreover, they often appear as building blocks in more sophisticated procedures.

To fix notation and terminology, we briefly introduce a few basic graph optimization problems in the following, without providing proofs of highlighted statements or going into further details. We refer the interested reader to standard textbooks on graph algorithms for more information.

2.1 Shortest paths

The shortest path problem is one of the most basic graph optimization problems with an abundance of applications. It is formally defined as follows.

Shortest path problem

Input: A directed graph $G = (V, A)$ with non-negative arc lengths $\ell\colon A \to \mathbb{R}_{\geq 0}$ and two vertices $s, t \in V$.
Task: Find an s-t path $P \subseteq A$ with smallest length $\ell(P) \coloneqq \sum_{a \in P} \ell(a)$.

We highlight that arc lengths are required to be non-negative. Indeed, negative arc lengths lead to an NP-hard problem class as this would allow for capturing the longest path problem.

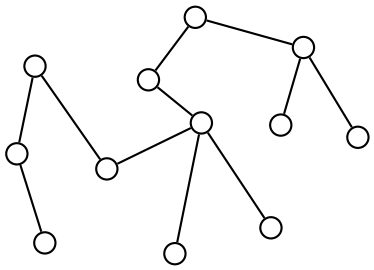
2.2 Minimum spanning trees

Spanning trees are edge-minimal graphs where all vertices are connected. Hence, they are minimal structures to connect given sites. We start with a formal definition and provide later several equivalent ways to characterize spanning trees.

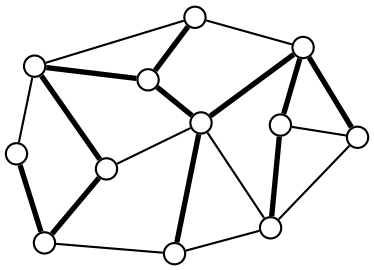
Definition 2.1: Spanning tree, Forest

(i) A *spanning tree* is an undirected graph $G = (V, E)$ in which there is a unique path between any pair of vertices.
(ii) A *forest* is a graph whose connected components are spanning trees.

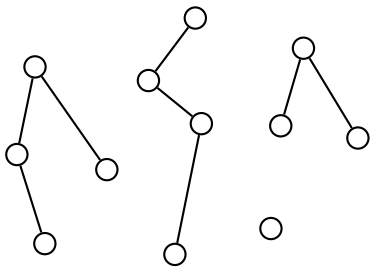
Hence, every spanning tree is also a forest, and every connected forest is a spanning tree. For convenience, we will often call an edge set $T \subseteq E$ of an undirected graph $G = (V, E)$ a *spanning tree* or *forest* in G, if the graph (V, T) is a spanning tree or forest, respectively. Figure 2.1 illustrates the notion of spanning tree and forest both as described in Definition 2.1 and also as edge sets of a graph.



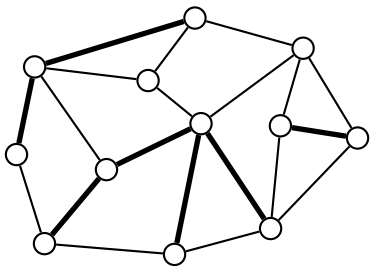
(a) A spanning tree.



(b) A spanning tree in a graph.



(c) A forest with four connected components, one of which being an isolated vertex, i.e., a vertex without any edges incident with it.



(d) A forest with five connected components in a graph. Two of the connected components are singleton vertices.

Figure 2.1: Examples for spanning trees and forests.

There are multiple ways to characterize spanning trees, which can easily shown to be equivalent. Apart from providing different viewpoints, which can often be helpful depending on the problem at hand, they also reveal some basic properties of spanning trees. Below, we highlight a few well-known equivalent ways to describe spanning trees.

Theorem 2.2: Characterizations of spanning trees

Let $G = (V, E)$ be an undirected graph with $n := |V|$ vertices and $m := |E|$ edges. The following statements are equivalent:

(i) G is a spanning tree, i.e., a graph with a unique path between any pair of vertices.
(ii) G has no cycles and is connected.
(iii) G is connected and it is edge-minimal with this property, i.e., removing any edge from G leads to a disconnected graph.
(iv) G is connected and $m = n - 1$.
(v) G has no cycles and $m = n - 1$.

(vi) G has no cycle and adding to G an edge between any two vertices creates a cycle.

To exemplify one implication of the above theorem, we can obtain an alternative characterization of forests: By (ii), a forest is simply a graph without cycles.

Moreover, the above theorem also shows that spanning trees are edge-minimal structures to connect vertices, as mentioned at the beginning. From this point of view, it is not surprising that they are often relevant in contexts where the task is to connect some given sites in a cheapest possible way. This naturally leads to the *minimum spanning tree problem*, which is formally defined below.

Minimum spanning tree problem

Input: An undirected graph $G = (V, E)$ with edge lengths $\ell \colon E \to \mathbb{R}$.

Task: Find a spanning tree $T \subseteq E$ of smallest length $\ell(T) \coloneqq \sum_{e \in T} \ell(e)$.

In the context of spanning trees and forests, vertices of degree one often play a distinguished role. To be able to refer to them more easily, they are called *leaves*.

Definition 2.3: Leaf

Let $G = (V, E)$ be a forest. A vertex $v \in V$ is a *leaf* if $\deg(v) = 1$.

2.3 Maximum flows and minimum cuts

Definition 2.4: s-t flow / flow

Let $G = (V, A)$ be a directed graph with non-negative arc capacities $u \colon A \to \mathbb{R}_{\geq 0}$ and let $s, t \in V$. An *s-t flow* in G is a function $f \colon A \to \mathbb{R}_{\geq 0}$ satisfying the following conditions.

(i) *Capacity constraints*: $f(a) \leq u(a)$ for all $a \in A$.

(ii) *Balance constraints*: For all $v \in V$,

$$f(\delta^+(v)) - f(\delta^-(v)) \begin{cases} = 0 & \text{if } v \in V \setminus \{s, t\} \ , \\ \geq 0 & \text{if } v = s \ , \\ \leq 0 & \text{if } v = t \ . \end{cases}$$

The *value* of a flow f is $\nu(f) \coloneqq f(\delta^+(s)) - f(\delta^-(s))$.

Figure 2.2 shows an example of an s-t flow.

The maximum flow problem, formally defined below, asks to find an s-t flow of maximum value.

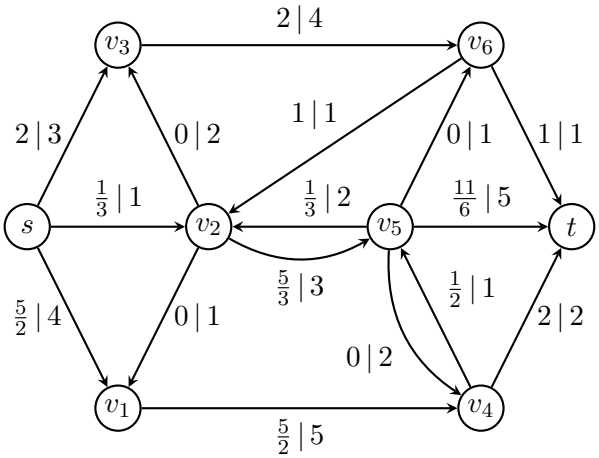


Figure 2.2: Example of an s-t flow. Next to each arc $a \in A$, one can find the value of the flow on a, i.e., $f(a)$, followed by its capacity $u(a)$, written as $f(a) \mid u(a)$. The value of the shown flow f is $\nu(f) = \frac{29}{6}$.

Maximum flow problem, or maximum s-t flow problem

Input: A directed graph $G = (V, A)$, arc capacities $u \colon A \to \mathbb{R}_{\geq 0}$, and $s, t \in V$, $s \neq t$.

Task: Find a maximum s-t flow in G, i.e., an s-t flow f that maximizes $\nu(f)$.

The input to a maximum s-t flow problem, i.e., a directed $G = (V, A)$ with non-negative arc capacities $u \colon A \to \mathbb{R}_{\geq 0}$ and vertices $s \in V$ and $t \in V$, is also called a *flow network*.

The values of s-t flows that one can achieve in a flow network is limited by bottlenecks in the network. Such bottlenecks can be described through s-t cuts. The value of an s-t cut, defined below, allows for bounding the largest possible throughput through the s-t cut, as we discuss later.

Definition 2.5: Value of an s-t cut, minimum s-t cuts

Let $G = (V, A)$ be a directed graph with arc capacities $u \colon A \to \mathbb{R}_{\geq 0}$. The *value* of an s-t cut C (with respect to u) is defined as $u(\delta^+(C))$. An s-t cut C is called *minimum* if it has minimum value among all s-t cuts.

First, we note that any s-t cut can be used to compute the value of any s-t flow.

Lemma 2.6: Value of a flow expressed via an s-t cut

Let $G = (V, A)$ be a directed graph with arc capacities $u\colon A \to \mathbb{R}_{\geq 0}$, let f be an s-t flow in G, and let $C \subseteq V$ be an s-t cut. Then

$$\nu(f) = f(\delta^+(C)) - f(\delta^-(C)) \ .$$

Moreover, one can easily verify that the value of any s-t cut upper bounds the value of any s-t flow. This important fact is known as the weak max-flow min-cut theorem, which is stated below.

Theorem 2.7: Weak max-flow min-cut theorem

Let $G = (V, A)$ be a directed graph with arc capacities $u\colon A \to \mathbb{R}_{\geq 0}$, let f be an s-t flow in G, and let $C \subseteq V$ be an s-t cut. Then

$$\nu(f) \leq u(\delta^+(C)) \ .$$

In other words, the value of a maximum s-t flow is upper bounded by the value of a minimum s-t cut.

Hence, the strongest possible upper bound that one can get on the value of s-t flows through Theorem 2.7 is obtained by an s-t cut C of minimum value. This leads to the well-known minimum s-t cut problem.

Minimum s-t cut problem

Input: A directed graph $G = (V, A)$, arc capacities $u\colon A \to \mathbb{R}_{\geq 0}$, and $s, t \in V$, $s \neq t$.

Task: Find a minimum s-t cut in G, i.e., an s-t cut $C \subseteq V$ that minimizes $u(\delta^+(C))$.

A celebrated stronger version of Theorem 2.7 is the so-called strong max-flow min-cut theorem, which states that the minimum s-t cut value matches the value of a maximum s-t flow. This is a formal justification of the above-mentioned intuition that s-t cuts can be used to describe bottlenecks in a flow network.

Theorem 2.8: Strong max-flow min-cut theorem

The value of a maximum s-t flow in a directed graph $G = (V, A)$ with arc capacities $u\colon A \to \mathbb{R}_{\geq 0}$ equals the value of a minimum s-t cut in G. In other words,

$$\max\{\nu(f)\colon f \text{ is an } s\text{-}t \text{ flow in } G\} = \min\left\{u(\delta^+(C))\colon C \text{ is an } s\text{-}t \text{ cut in } G\right\} \ .$$

Both the maximum s-t flow problem and the minimum s-t cut problem can be solved efficiently, actually even in strongly polynomial time. Moreover, they are used in an impressive number of applications because they can model a wide variety of interesting algorithmic prob-

lems, including many combinatorial optimization problems. At first sight, it may be surprising that a problem with a continuous solution space, like the maximum s-t flow problem, can be so useful to model problems with a discrete solution space as they are encountered in the field of Combinatorial Optimization. The main reason for this is the fact that any flow network with integer capacities always admits an integral flow, and there are algorithms to find such flows that run in polynomial time and are also very efficient in practice. Integral flows can often be used to model discrete problems. For example when the arc capacities are all 1, then an integral flow has a flow value of either 0 or 1 on each arc. Such a flow can then be interpreted as a subset of the arcs, namely those with flow value 1. This type of reasoning allows for modeling an impressively large range of combinatorial optimization problems. The theorem below formalizes the existence of integral maximum s-t flows and the fact that they can be computed in polynomial time.

Theorem 2.9: Integral maximum flows

Let $G = (V, A)$ be a directed graph with integer capacities $u\colon A \to \mathbb{Z}_{\geq 0}$, and let $s, t \in V$, $s \neq t$. Then there always exists a maximum s-t flow that is integral. Moreover, there are strongly polynomial time algorithms to compute an integral maximum s-t flow.

3 Matching Problems

Matching problems are about finding the best possible way to pair up vertices of a graph by using its edges. They enjoy a very wide set of applications and are a rich field of study in their own. One of the arguably most famous applications is the assignment problem, which can loosely be described as assigning jobs to machines, or tasks to workers. Classical matching problems are defined on undirected graphs. Hence, throughout this chapter, we focus on undirected graphs if not explicitly stated otherwise.

3.1 Basics on matchings and vertex covers

We start by formally defining matchings, which is the main notion of this chapter.

Definition 3.1: Matching

Let $G = (V, E)$ be an undirected graph. An edge set $M \subseteq E$ is called a *matching (in G)* if M does not contain any loops and every vertex is incident to at most one edge of M.

Given an undirected graph $G = (V, E)$ and a matching $M \subseteq E$, we say that a vertex $v \in V$ is *unmatched* or *exposed* (by M), if v is not incident to any edge of M; otherwise, we say that v is *covered* (by M). A matching is *perfect* if no vertex is exposed. Clearly, only graphs with an even number of vertices admit perfect matchings. The arguably most canonical matching problem is finding a matching of maximum cardinality.

Maximum cardinality matching problem

Input: An undirected graph $G = (V, E)$.

Task: Find a matching in G of maximum cardinality.

When dealing with a matching problem on a graph $G = (V, E)$, like the maximum cardinality matching problem defined above, we generally assume $|E| = \Omega(|V|)$ in running time analyses. This is not a very restrictive assumption, as it holds for any connected graph. Moreover, a matching problem on a graph trivially decomposes into independent problems on each of its connected components. Hence, if a graph is not connected, then one can typically perform a simple preprocessing step that breaks the problem down into independent problems on each connected component.

For any graph G, we denote by $\nu(G)$ the cardinality of a maximum cardinality matching in G. A problem that, as we will see, is very closely related to the maximum cardinality matching

problem, is the minimum cardinality vertex cover problem, which we formally define below after introducing the notion of a vertex cover.

Definition 3.2: Vertex cover

Let $G = (V, E)$ be an undirected graph. A set $S \subseteq V$ is a *vertex cover (of G)* if every edge of E is incident to at least one vertex in S.

Minimum cardinality vertex cover problem

Input: An undirected graph $G = (V, E)$.

Task: Find a vertex cover in G of minimum cardinality.

For any graph G, we denote by $\tau(G)$ the cardinality of a minimum cardinality vertex cover in G.

Consider a graph $G = (V, E)$, a matching $M \subseteq E$, and a vertex cover $S \subseteq V$. Because every matching edge has to be incident to at least one vertex in S, and, furthermore, any vertex in S can be incident to at most one edge in M, we obtain that the size of any matching cannot exceed the size of any vertex cover:

Observation 3.3

For any graph $G = (V, E)$,

$$\nu(G) \leq \tau(G) \ .$$

To construct matchings of large sizes, a typical approach is to start with some matching that is easy to find—this could even be the empty matching $M = \emptyset$—and to successively transform the current matching to a larger one. Such an augmentation from a smaller to a larger matching can be achieved by using (a particular type of) alternating paths, defined as follows.

Definition 3.4: M-alternating path

For a graph G and matching M of G, a path in G is called *M-alternating*, or simply *alternating* when M is clear from context, if its edges alternate between M and $E \setminus M$ when traversing the path.

Augmenting a matching M to a larger one can be achieved by an alternating path whose endpoints are both exposed. Such paths are called M-augmenting.

Definition 3.5: M-augmenting path

For a graph G and matching M of G, an M-alternating path is called *M-augmenting* if both its start and endpoint are exposed.

Indeed, one can easily check that for any matching $M \subseteq E$ in an undirected graph $G = (V, E)$ and any M-augmenting path $P \subseteq E$, we have that $M \triangle P := (M \setminus P) \cup (P \setminus M)$ is a matching with one edge more than M, i.e., $|M \triangle P| = |M| + 1$.[1] Going from a matching M to a larger matching $M \triangle P$ through an M-augmenting path P is also called *augmenting* the matching M through P. The resulting matching $M \triangle P$ is also called the *augmented matching* (when M and P are clear from context).

The above discussion clearly motivates the notion of M-augmenting paths as one way to augment a matching. Of course, this leaves open whether it may be possible to have a matching M that is not of maximal cardinality and neither admits an M-augmenting path. This, of course, would be a major issue when trying to find maximum cardinality matchings by successively augmenting matchings through augmenting paths, as it may then be possible to get stuck with a successive augmentation approach. As the theorem below shows, this situation is impossible. More precisely, M-augmenting paths always exist if M is not a matching of maximum cardinality.

Theorem 3.6

A matching M is of maximum cardinality if and only if there is no M-augmenting path.

Proof. Let $G = (V, E)$ denote the underlying graph. We prove the theorem by showing the contrapositive, namely that a matching M in G is not of maximum cardinality if and only if there is an M-augmenting path.

$\Rightarrow$) If there is an augmenting path $P \subseteq E$ with respect to M, then $M \Delta P$ is a matching with $|M \Delta P| = |M| + 1$. Hence, M was not of maximum cardinality.

$\Leftarrow$) Let M be a matching that is not of maximum cardinality. Hence, there is a matching $\overline{M} \subseteq E$ in G with $|\overline{M}| > |M|$. Consider $M \Delta \overline{M}$. Each vertex $v \in V$ is incident to at most one edge of M and at most one edge of $\overline{M}$. Thus, $M \Delta \overline{M}$ consist of paths and cycles that are alternating between M and $\overline{M}$. Because $|\overline{M}| > |M|$, there is at least one connected component of $(V, M \Delta \overline{M})$ whose edge set P contains (strictly) more edges from $\overline{M}$ than M. This implies that P is a path with both of its pending edges being contained in $\overline{M}$. Hence, P is an M-augmenting path. □

Consequently, by Theorem 3.6, to design an efficient algorithm to find a maximum cardinality matching, it suffices to find an efficient procedure that, given a matching M in a graph $G = (V, E)$, finds an M-augmenting path if there is one. Indeed, classical matching algorithms follow this plan. We start with bipartite graphs, for which significantly easier procedures to find augmenting paths are known than in general (not necessarily bipartite) graphs.

[1] We recall that $\triangle$ is the symmetric difference operator. More precisely, for any two sets A and B, the expression $A \triangle B$ is a shorthand for $(A \setminus B) \cup (B \setminus A)$.

3.2 Maximum cardinality bipartite matching

It turns out that many applications of matchings can naturally be described in so-called *bipartite* graphs $G = (V, E)$, which are defined as follows.

Definition 3.7: Bipartite graph

A graph $G = (V, E)$ is *bipartite* if its vertex set V admits a bipartition $V = X \dot\cup Y$ such that every edge $e \in E$ has one endpoint in X and the other one in Y.

Moreover, matching problems are typically significantly easier to study and solve on bipartite graphs than general graphs. We therefore often start with discussing the bipartite case.

For convenience, we use simplified terminology and notation to refer, for a bipartite graph $G = (V, E)$, to a bipartition $V = X \dot\cup Y$ of its vertex set that certifies bipartiteness, i.e., every edge has one endpoint in X and one in Y. First, to introduce a bipartite graph $G = (V, E)$ together with a bipartition $V = X \dot\cup Y$, we often simply say that $G = (X \dot\cup Y, E)$ *is a bipartite graph*, or we may also write $G = (V = X \dot\cup Y, E)$ to highlight that V denotes the set of all vertices. Sometimes we also talk about a *bipartite graph* $G = (V, E)$ *with bipartition* $V = X \dot\cup Y$. Note that the bipartition $X \dot\cup Y$ is not unique. In particular, one can exchange the roles of X and Y. More generally, if a bipartite graph has multiple connected components, then one can exchange the vertices contained in X and Y independently for each of the connected components to obtain different bipartitions of V certifying that G is bipartite.

In bipartite graphs, maximum cardinality matchings and minimum cardinality vertex covers always have the same cardinality. This famous result is known as König's Theorem, formally stated below.

Theorem 3.8: König's Theorem

For any bipartite graph G,

$$\nu(G) = \tau(G) \ .$$

König's Theorem is a so-called *min-max theorem*, as it shows equality, in terms of objective values, between a maximization problem (maximum cardinality matchings) and a minimization problem (minimum cardinality vertex cover). Apart from revealing a close relation between two ostensibly unrelated problems, such min-max theorems often have an important algorithmic relevance.

Indeed, notice that already Observation 3.3 shows that if we find an algorithm that, given a graph, computes both a matching and a vertex cover of the same cardinality, then the computed matching must be of maximum cardinality and the computed vertex cover must be of minimum cardinality. However, Observation 3.3 does not guarantee that one can always find a matching and vertex cover of identical cardinality in any graph. (And this indeed does not generally hold for non-bipartite graphs—a triangle being a concrete bad example.) However, for the case of bipartite graphs, König's Theorem shows that a matching/vertex cover pair of identical cardinality always exists. Hence, for bipartite graphs, there is hope to design an efficient procedure

that always returns a matching and vertex cover of same cardinality, thus immediately implying correctness of the procedure. Hence, in short, a *min-max theorem* typically leads to a canonical way to prove correctness of an algorithm.

Min-max theorems in combinatorial settings, like König's Theorem, can very often be seen to be a special case of one of the most fundamental min-max relations in Mathematical Optimization, namely strong duality of linear programming. However, we will not rely on linear programming duality in this chapter, and provide a proof of Königs Theorem algorithmically, through an efficient algorithm that finds a maximum cardinality matching in a graph, and allows for finding a vertex cover of identical cardinality. This combinatorial approach to prove König's Theorem is instructive in its own.

As already mentioned, we leverage Theorem 3.6 to find a maximum cardinality matching by successively finding augmenting paths. In bipartite graphs, augmenting paths are easy to find, as illustrated by the following algorithm, whose correctness is not hard to check.

Algorithm 2: Finding an augmenting path in a bipartite graph if possible.

Input : Bipartite graph $G = (V = X \dot\cup Y, E)$ and matching M in G.
Output: An augmenting path with respect to M if it exists.

1. Consider a directed version $\vec{G} = (V, A)$ of G obtained by orienting all edges in $E \setminus M$ from X to Y and all edges in M from Y to X.
2. Let $X_1 \subseteq X$ and $Y_1 \subseteq Y$ be the set of all exposed vertices in X and Y, respectively. Mark all vertices $L \subseteq X \cup Y$ reachable from X_1 (and remember predecessors while marking; hence, this can be done through BFS).
3. If $L \cap Y_1 \neq \emptyset$, then we **return** a path from X_1 to Y_1, which is an M-augmenting path. Otherwise there is no M-augmenting path.

By using Algorithm 2, an algorithm for maximum cardinality matching in bipartite graphs is readily obtained; see Algorithm 3. Notice that the running time of Algorithm 3 is bounded by

Algorithm 3: Finding a maximum cardinality matching in a bipartite graph.

Input : Bipartite graph $G = (X \dot\cup Y, E)$.
Output: A maximum cardinality bipartite matching M.

1. Initialize: $M = \emptyset$.
2. Find an M-augmenting path $P \subseteq E$ via Algorithm 2 if possible. If augmenting path was found, set $M \leftarrow M \Delta P$, and repeat this step. Otherwise, **return** M.

$O(nm)$, where $n := |V|$ and $m := |E|$. This follows by observing that step 2 takes $O(m)$ time and will be repeated at most $n/2$ times because $\nu(G) \leq n/2$.

The following statement immediately implies König's Theorem (Theorem 3.8), since it shows that there is a matching M and a vertex cover S with $|M| = |S|$, which together with Observation 3.3 implies

$$|M| \leq \tau(G) \leq \nu(G) \leq |S| \ .$$

Because $|M| = |S|$, we must have equality throughout the above inequality chain.

Theorem 3.9

Let $G = (X \dot\cup Y, E)$ be a bipartite graph, M be a maximum cardinality matching, and let $L \subseteq V$ be the set of marked vertices when applying Algorithm 2 to G and M. Then $S = (X \setminus L) \cup (Y \cap L)$ is a vertex cover of size $|S| = |M|$, which, due to Observation 3.3, is a minimum cardinality vertex cover in G.

Proof. We start by showing that S is a vertex cover. For this we have to prove that there is no edge $\{x, y\} \in E$ with $x \in X \cap L$ and $y \in Y \setminus L$. No such edge with $\{x, y\} \in E \setminus M$ can exist, as in that case we would have marked y to be in L. However, such an edge with $\{x, y\} \in M$ cannot exist either, because then the only way for x to have been marked to be part of L was over the edge $\{x, y\}$ because (i) x is not exposed as it is incident to $\{x, y\} \in M$, and (ii) in the directed graph $\vec{G}$ used for the marking as explained in Algorithm 2, (y, x) is the only arc entering x. Hence, S is indeed a vertex cover.

It remains to show that $|M| = |S|$. More precisely, it suffices to show $|M| \geq |S|$ because $|M| \leq |S|$ is implied by Observation 3.3. Notice that there is no edge $\{x, y\} \in M$ with $x \in X \setminus L$ and $y \in Y \cap L$ because such an edge would imply that x should have been marked as part of L, i.e., no edge in M has both endpoints in S. Moreover, observe that none of the vertices in S is exposed: No vertex in $Y \cap L$ can be exposed as this would imply that we found an augmenting path, and all exposed vertices in X were marked and are thus part of L. Hence, every vertex in S is incident to an edge in M, and no edge of M has both endpoints in S. Thus, $|M| \geq |S|$. □

König's Theorem allows for readily deriving another famous result about matchings in bipartite graphs, namely Hall's Theorem. Hall's Theorem provides a characterization of when a bipartite graph $G = (X \dot\cup Y, E)$ admits a matching touching all vertices of X by providing a condition on the neighborhood of vertices in X. Recall that for any graph $G = (V, E)$ and a vertex set $U \subseteq V$, the *neighbors* of U are $\{v \in V \setminus U \mid \exists u \in U \text{ with } \{u, v\} \in E\}$, and we denote this set by $N(U)$.

Theorem 3.10: Hall's Theorem

Let $G = (X \dot\cup Y, E)$ be a bipartite graph. Then there exists a matching touching all vertices of X if and only if $|N(U)| \geq |U|$ for all $U \subseteq X$.

Proof. $\Rightarrow$) Assume that there is a set $U \subseteq X$ with $|N(U)| < |U|$. Then $N(U) \cup (X \setminus U)$ is a vertex cover of size strictly less than $|X|$. Hence $\nu(G) = \tau(G) < |X|$ by König's Theorem.

$\Leftarrow$) Assume that $\nu(G) < |X|$. Hence, by König's Theorem, there is a vertex cover S of size $|S| = \nu(G) < |X|$. Because S is a vertex cover, there are no edges between $X \setminus S$ and $Y \setminus S$, and thus $N(X \setminus S) \subseteq Y \cap S$. However, this implies

$$|N(X \setminus S)| \leq |Y \cap S| = |S| - |X \cap S| < |X| - |X \cap S| = |X \setminus S| \ .$$ □

3.2.1 An application of König's Theorem

Let $G = (V, E)$ be a bipartite graph and let $k \in \mathbb{Z}_{\geq 0}$. Assume that you can remove up to k vertices from G and our goal is to do this in a way that minimizes the cardinality of the maximum cardinality matching in the resulting graph. Hence, formally speaking, the task is to find a vertex set $R \subseteq V$ with $|R| \leq k$ such that $\nu(G[V \setminus R])$ is as small as possible.

This problem can easily be solved by using König's Theorem, which implies that the problem is equivalent to removing a set $R \subseteq V$ with $|R| \leq k$ such that $\tau(G[V \setminus R])$ is minimized. Let S be a minimum cardinality vertex cover in G. We claim that the best strategy is to remove any k vertices $R \subseteq S$. Notice that if $k \geq |S|$, then removing S will remove all edges, and is therefore an optimal solution. Hence, assume $|S| \geq k$. First, notice that after removing any k vertices $R \subseteq S$, the set $S \setminus R$ is a vertex cover in $G[V \setminus R]$. Hence,

$$\nu(G[V \setminus R]) \leq \tau(G[V \setminus R]) \leq |S \setminus R| = \nu(G) - k \ , \tag{3.1}$$

where the first inequality follows by Observation 3.3. Conversely, let M be a maximum cardinality matching in G. We have

$$\nu(G[V \setminus U]) \geq |M| - k = \nu(G) - k \quad \forall\, U \subseteq V \text{ with } |U| \leq k \ ,$$

because there are at least $|M| - k$ edges of M not adjacent to any vertex of U. The above inequality together with (3.1) shows that our choice of R is indeed optimal.

3.3 Weighted bipartite matchings

We now consider weighted matching problems. In the previous chapters, we were interested in the size of a matching, i.e., the number of edges that the matching contains, so each edge contributed equally. In a weighted setting, edges can have non-uniform contributions, and the size of a matching is generalised to real-valued *weight*. More formally, we are given an undirected graph $G = (V, E)$ together with non-negative edge weights $w\colon E \to \mathbb{R}_{\geq 0}$. The weight of a matching M is given by $w(M) \coloneqq \sum_{e \in M} w(e)$. Below, we define three typical variants of weighted matching problems that are analogous in both bipartite and general (not necessarily bipartite) graphs.

Maximum weight (bipartite) matching problem

Input: An undirected (bipartite) graph $G = (V, E)$ with edge weights $w\colon E \to \mathbb{R}_{\geq 0}$.
Task: Find a matching $M \subseteq E$ of maximum weight $w(M)$.

Maximum weight (bipartite) perfect matching problem

Input: An undirected (bipartite) graph $G = (V, E)$ with edge weights $w\colon E \to \mathbb{R}_{\geq 0}$.
Task: Find a perfect matching $M \subseteq E$ of maximum weight $w(M)$.

Minimum weight (bipartite) perfect matching problem

Input: An undirected (bipartite) graph $G = (V, E)$ with edge weights $w\colon E \to \mathbb{R}_{\geq 0}$.
Task: Find a perfect matching $M \subseteq E$ of minimum weight $w(M)$.

In bipartite graphs, the problems of finding either a maximum or minimum weight perfect matching are also called *assignment problems*.

Note that even though in the above problem definitions we assumed that the edge weights are non-negative, this is not restrictive, but merely a common convention. For example, when dealing with a maximum weight matching problem including negative edge weights, we can simply delete all edges with negative weights, as they are never part of a maximum weight matching, because removing them from any matching leads to a matching with strictly larger weight. Moreover, for the problem of finding a maximum or minimum weight perfect matching, when given edge weights $w\colon E \to \mathbb{R}$ with potentially negative values, we can simply shift the weights by a sufficiently large constant to obtain non-negative edge weight. For example, one can use $w'\colon E \to \mathbb{R}_{\geq 0}$ defined by $w'(e) = w(e) - \min\{w(f)\colon f \in E\}$ for $e \in E$. This changes the weight of every perfect matching by the same amount of $\frac{|V|}{2} \cdot \min\{w(f)\colon f \in E\}$. Thus, a perfect matching of maximum/minimum w-weight is a matching of maximum/minimum w'-weight.

In what comes next, we first observe that these problems can also easily be reduced to each other. We then concentrate on an efficient algorithm for finding a minimum weight perfect matching in bipartite graphs, known as the *Hungarian Method*.

3.3.1 Relation between different weighted matching problems

We start by observing that the above problems—in both their bipartite as well as general (not necessarily bipartite) version—can be reduced to each other, in the sense that any polynomial algorithm for one problem can be used to obtain a polynomial algorithm for any of the other problems.

Lemma 3.11

Between the following problems (which can be either all restricted to bipartite graphs or not), there are polynomial reductions that change the size of the graph by at most a constant factor:

(i) maximum weight (bipartite) matching,
(ii) maximum weight (bipartite) perfect matching,
(iii) minimum weight (bipartite) perfect matching.

Proof. (i) $\rightarrow$ (ii): We start by showing how an algorithm for (i) can be used to solve (ii). Hence, assume we are given an algorithm to solve the maximum weight (bipartite) matching problem. Consider a problem instance for the maximum weight (bipartite) perfect matching problem, i.e., given is an undirected graph $G = (V, E)$ with edge weights $w\colon E \rightarrow \mathbb{R}_{\geq 0}$. To find a maximum weight perfect matching in G, we find a maximum weight matching $M \subseteq E$ in G with respect to the auxiliary edge weights $w'\colon E \rightarrow \mathbb{R}$ defined by

$$w'(e) \coloneqq w(e) + 1 + w(E) \quad \forall\, e \in E \ .$$

We claim that whenever G admits a perfect matching, then M is a maximum w-weight perfect matching in G. Thus, if M is perfect, we know that M is the maximum w-weight perfect matching in G; and if M is not perfect, we know that G does not admit a perfect matching.

To prove that the claim is true, first observe that if there is a perfect matching in G, then a maximum w'-weight matching must be a perfect matching because any perfect matching $\overline{M}$ in G fulfills $w'(\overline{M}) \geq \frac{|V|}{2}\,(1 + w(E))$ and every matching that is not perfect has w'-weight strictly less than that. In short, no matching that is not perfect can make up for the extra $1 + w(E)$ units of added w'-weight that a matching with larger cardinality gets. Moreover, if M is a perfect matching, then its w'-weight is $w'(M) = \frac{|V|}{2}\,(1 + w(E)) + w(M)$. Hence, a perfect matching in G has largest w'-weight if it has largest w-weight, showing that M is a maximum w-weight perfect matching in G, as desired.

(ii) $\leftrightarrow$ (iii): These problems can easily be interchanged by replacing the weights $w(e)$ by $w'(e) = \max\{w(f) \mid f \in E\} - w(e)$ for all $e \in E$.

(ii) $\rightarrow$ (i): This reduction can be done by adding additional auxiliary vertices and edges (with weight zero) to G to obtain an auxiliary graph H, such that any matching in G can be completed

to a perfect matching in H of same weight. One way to do this such that the size of H is at most a constant factor larger than the size of G is as follows. Let $G' = (V', E')$ be an independent copy of the graph G; hence for any $v \in V$ there is a vertex $v' \in V'$, and $E' = \{\{u', v'\} \mid \{u, v\} \in E\}$. Consider the graph $H = (V \cup V', E \cup E' \cup F)$, where $F = \{\{v, v'\} \mid v \in V\}$. The edge weights w_H of H are chosen to be an extension of the weights w, by assigning a weight of zero to all edges not in E. Notice that the weight of any perfect matching M_H in H is thus simply the weight of the submatching $M_H \cap E$, which is a matching in G. Conversely, any matching M in G can be extended to a perfect matching M_H in H as follows

$$M_H = M \cup \{\{u', v'\} \colon \{u, v\} \in M\} \cup \{\{v, v'\} \colon v \text{ exposed wrt } M\} \ .$$

Hence, a maximum weight matching in G can be obtained by determining a maximum perfect matching M_H in H and returning $M_H \cap E$.

One can observe that all reductions presented above preserve bipartiteness and do not increase the problem size by more than a constant factor. □

3.3.2 The Hungarian Method

We now discuss the Hungarian Method, which is an algorithm to solve the minimum weight perfect matching problem in bipartite graphs $G = (V = X \dot\cup Y, E)$ with edge weights $w \colon E \to \mathbb{R}_{\geq 0}$. The Hungarian Method is based on a very simple yet very useful observation. If we add the same value $\Delta \in \mathbb{R}$ to the weight of each edge incident to some vertex $v \in V$, then we obtain an equivalent minimum weight perfect matching problem. More formally, consider the edge weights w' given by

$$w'(e) = \begin{cases} w(e) + \Delta & \text{if } e \in \delta(v), \\ w(e) & \text{otherwise.} \end{cases}$$

Because every perfect matching M in G contains precisely one edge of $\delta(v)$, we have $w'(M) = w(M) + \Delta$. Hence, the weight of any perfect matching simply shifts by Δ, and minimum w'-weight perfect matchings are therefore the same as minimum w-weight perfect matchings. We only consider such edge weight modifications such that the resulting weights w' remain non-negative. To easily describe a sequence of such changes, applied to different vertices, we introduce the following notion of a feasible node potential, which describes how much we added to the edges incident to each vertex.

Definition 3.12: Feasible node potential

Let $G = (V, E)$ be a bipartite graph with non-negative edge weights $w \colon E \to \mathbb{R}_{\geq 0}$. A function $\eta \colon V \to \mathbb{R}$ is a *feasible node potential* (with respect to G and w) if

$$w(\{u, v\}) + \eta(u) + \eta(v) \geq 0 \quad \forall \{u, v\} \in E \ .$$

For a feasible node potential η, we define the corresponding modified weight function $w_\eta \colon E \to \mathbb{R}_{\geq 0}$ by $w_\eta(\{u, v\}) \coloneqq w(\{u, v\}) + \eta(u) + \eta(v)$.

The Hungarian Method iteratively changes the current weights by updating a feasible node potential. The goal is to end up with some non-negative weights $w_\eta \colon E \to \mathbb{R}_{\geq 0}$ such that there is a perfect matching M in G with $w_\eta(M) = 0$. Such a matching M is clearly of minimum w_η-weight (and thus also of minimum w-weight) because any matching in G has non-negative weight with respect to w_η. We summarize this in the following lemma.

Lemma 3.13

Let $G = (V, E)$ be a bipartite graph with non-negative edge weights $w \colon E \to \mathbb{R}_{\geq 0}$ and let M be a perfect matching in G. If there is a feasible node potential $\eta \colon V \to \mathbb{R}$ with $w_\eta(M) := \sum_{e \in M} w_\eta(e) = 0$, then M is a minimum w-weight perfect matching in G.

Proof. By our discussion on node potentials, showing that M is a minimum w-weight perfect matching is equivalent to showing that M is a minimum w_η-weight perfect matching. However, this immediately follows from the fact that the modified edge weights w_η are non-negative. Indeed, every perfect matching in G has non-negative w_η-weight due to non-negativity of w_η, and because M satisfies $w_\eta(M) = 0$, it must have smallest w_η-weight among all matchings. □

Interestingly, if G admits a perfect matching, we can always find a feasible node potential η and a matching M that satisfy the conditions of Lemma 3.13. This is the central result that underlies the Hungarian Method. It is summarized below, and we will prove it as a byproduct of the correctness of the Hungarian Method.

Theorem 3.14

Let $G = (V, E)$ be a bipartite graph with non-negative edge weights $w \colon E \to \mathbb{R}_{\geq 0}$ that admits a perfect matching. Then there exists a feasible node potential η and a perfect matching $M \subseteq E$ such that

$$w_\eta(e) = 0 \quad \forall\, e \in M \ .$$

Note that this implies that M is a minimum weight perfect matching due to Lemma 3.13.

3.3.2.1 The Hungarian Method by example

In this section we present a version of the Hungarian Method on an example. We will later formally state and analyze a slight variation of the version of the Hungarian Method that we use for the example, on which we impose more conditions on how some particular steps are performed. The goal of the example to be presented next is to build up intuition for the Hungarian Method. Furthermore, the approach we present on the example allows for a nice way to see why Theorem 3.14 holds.

The example we consider is a complete bipartite graph on 8 vertices. Notice that we can assume without loss of generality that the bipartite graph is complete; for otherwise, one can add the missing edges with a sufficiently high edge weight like the sum of the four highest edge weights plus one. We represent the example graph $G = (X \dot\cup Y, E)$ with $X = \{x_1, x_2, x_3, x_4\}$,

$Y = \{y_1, y_2, y_3, y_4\}$, and weights $w\colon E \to \mathbb{R}_{\geq 0}$ by a matrix $W \in \mathbb{R}_{\geq 0}^{X \times Y}$, i.e., each row corresponds to a vertex in X and each column to a vertex in Y, and $W_{x,y} = w(\{x, y\})$ for $x \in X$ and $y \in Y$. The matrix below, with rows and columns labeled by the vertices in X and Y, respectively, describes our example instance.

	y_1	y_2	y_3	y_4
x_1	2	6	7	4
x_2	0	7	10	7
x_3	2	8	9	7
x_4	0	1	7	0

As mentioned, our goal is to obtain modified weights that admit a perfect matching of modified weight zero. To check whether such a perfect matching only using edges of weight 0 exists with respect to the original weights w, we solve a maximum cardinality bipartite matching problem on the subgraph consisting only of edges of weight zero. In this case, the edges of zero weight are $E_0 \coloneqq \{\{x_2, y_1\}, \{x_4, y_1\}, \{x_4, y_4\}\}$, and the unique maximum cardinality matching in (V, E_0) is $M_0 \coloneqq \{\{x_2, y_1\}, \{x_4, y_4\}\}$. As this matching has only cardinality 2, instead of the desired size of 4 of a perfect matching, we modify the weights. To this end, we compute a minimum cardinality vertex cover $S \subseteq V$ in (V, E_0), which, by Königs Theorem, has the same cardinality as M_0, namely 2. Such a vertex cover S can, for example, be computed from M_0 using Theorem 3.9, which leads to $S = \{x_2, x_4\}$. The matrix below highlights the matching M_0 by surrounding the weights corresponding to entries in M_0 with rectangles, and the vertex cover S is highlighted by using red stars to indicate the rows and columns corresponding to vertices in S.

	2	6	7	4
⋆	$\boxed{0}$	7	10	7
	2	8	9	7
⋆	0	1	7	$\boxed{0}$

We now define a feasible node potential as follows. For some parameter $\Delta > 0$, to be fixed later, we will choose the node potential $\eta\colon V \to \mathbb{R}$ defined by

$$\eta(v) = \begin{cases} \Delta & \text{if } v \in S \ , \\ -\Delta & \text{if } v \in V \setminus S \ . \end{cases}$$

We choose Δ to be the largest possible value such that η is a feasible node potential, i.e., w_η is non-negative. One can observe that this value is equal to

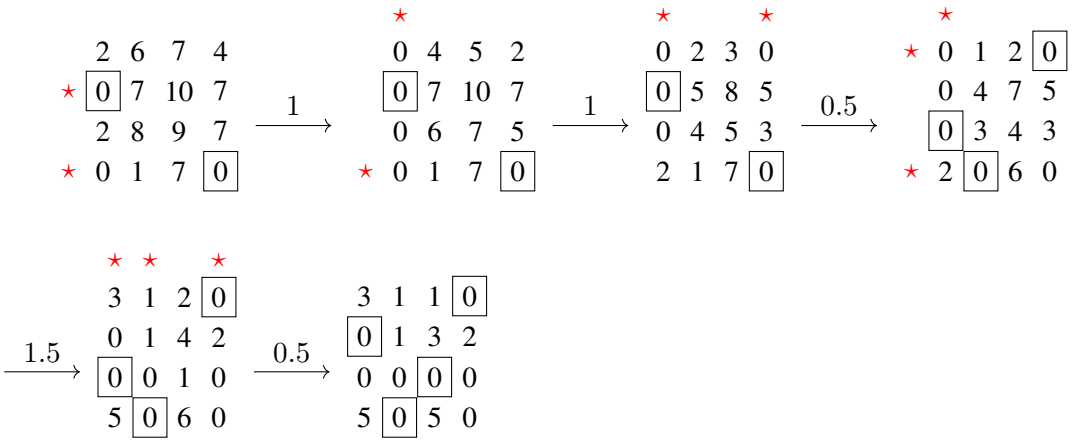
$$\Delta = \frac{1}{2} \min \{W_{x,y} : x \in X \setminus S, y \in Y \setminus S\} \ .$$

In our example, this leads to a value of $\Delta = 1$. Notice that the above definition of Δ must lead to a strictly positive value, because $W_{x,y} > 0$ for all $x \in X \setminus S$ and $y \in Y \setminus S$ as S is a vertex

cover for the edges of weight 0. The resulting modified weights are shown in the weight matrix below.

$$\begin{matrix} 0 & 4 & 5 & 2 \\ 0 & 7 & 10 & 7 \\ 0 & 6 & 7 & 5 \\ 0 & 1 & 7 & 0 \end{matrix}$$

We keep iterating the steps above, which leads to a sequence of modified weights as shown below. As above, the rectangles highlight a maximum cardinality matching among all edges of modified weight zero, and the red stars indicate the rows and columns corresponding to a minimum cardinality vertex cover when only edges of weight zero are considered. The values on the arrow indicate the value of Δ used to modify the weights.



Hence, the last modified weights we obtained admit a perfect matching only using edges of modified weight equal to zero. By Theorem 3.14, this matching is a minimum w-weight perfect matching, as desired. The above procedure is one realization of the *Hungarian Method*. Note that the way we introduced it here, we left it open how to compute minimum cardinality vertex covers S to update the node potentials. This leaves room for different ways to realize the Hungarian Method. Before returning to this topic, we first discuss some properties that are independent of the procedure used to compute minimum cardinality vertex covers.

We provide a first reasoning showing that, when starting with integer edge weights w, this procedure will always find a node potential η so that the edges E_η of modified weight zero, i.e.,

$$E_\eta := \{e \in E \colon w_\eta(e) = 0\} \quad ,$$

admit a perfect matching. To this end, first observe that at each iteration we change the node potential of each node by either $+\Delta$ or $-\Delta$. Hence, for each edge, its modified weight either (i) stays the same, (ii) increases by 2Δ, or (iii) decreases by 2Δ. In particular, this implies that if w is integral, then also all modified weights during the procedure remain integral. Indeed, if the current modified weights are integral, then Δ will be half-integral, i.e., $\Delta \in \frac{1}{2}\mathbb{Z}_{\geq 0}$, which will

change the weights by an integral amount because each weight changes by either -2Δ, 0, or 2Δ. Moreover, as long as E_η does not admit a perfect matching, the minimum cardinality vertex cover has size strictly less then the number of rows (or columns), and there are strictly more vertices whose node potentials get decreased by Δ than there are vertices whose node potentials get increased by Δ. Hence, the sum of all modified weights strictly decreases at each iteration. Because modified weights are always non-negative and integer, this procedure must stop, which implies that we will end up with node potentials η such that E_η admits a perfect matching.

The same argument works for rational weights w by first multiplying them with a large integer to make them integral. Hence, this is a way of showing Theorem 3.14 for rational edge weights w. The argument easily extends to non-negative real entries, for example by considering the feasible node potential that minimizes the sum of all entries. Such a node potential must exist by compactness arguments. This node potential must lead to a matrix where the zero-entries admit a perfect matching; for otherwise one could apply one step of the update procedure, which changes the entries by some value $\Delta > 0$, thus reaching a feasible node potential whose corresponding matrix has a smaller sum of entries.

The above discussion nicely highlights the core ideas behind why the Hungarian Method works, and shows how one can prove Theorem 3.14. However, it leaves one important point open, namely how quickly the procedure will terminate. Interestingly, this depends on the choice of the minimum cardinality vertex cover in each iteration. Even for integer starting weights w, a bad choice can lead to an algorithm that needs exponentially many steps in the input size to terminate. We will see that by choosing the minimum cardinality vertex cover as described in Theorem 3.9 (which is based on Algorithm 2 to label vertices), the number of iterations—and therefore the whole algorithm—has a running time with a strongly polynomial bound. For simplicity, we will call this specific realization of the algorithm the *Hungarian Method*. We introduce it formally in what follows and provide a proof of a strongly polynomial running time bound.

3.3.2.2 Formal definition and analysis of the Hungarian Method

Algorithm 4 gives a formal description of the Hungarian Method, where we use Theorem 3.9 to obtain minimum cardinality vertex covers with respect to the edges of modified weight zero.

Note that in step 3 of the above algorithm, Δ is the maximum value that leads to non-negative weights $w_\eta \geq 0$ when performing the update in (3.2). For the analysis, we first observe that Algorithm 4 maintains a feasible node potential throughout the full execution of the algorithm, and that all updates in step 3 are non-trivial, i.e., with a parameter $\Delta > 0$.

Lemma 3.15

Whenever Δ is determined in step 3 of Algorithm 4, we have $\Delta > 0$. Furthermore, at any step during Algorithm 4, η is a feasible node potential.

Proof. The initial node potential η, which is all-zero, is clearly feasible. Hence, consider a run through step 3. To prove the result, we show the following. Assuming that the node potential η when entering step 3 is feasible, we show that $\Delta > 0$ and the node potential at the end of step 3,

Algorithm 4: Hungarian Method

Input : Bipartite graph $G = (X \dot\cup Y, E)$ with edge weights $w\colon E \to \mathbb{R}_{\geq 0}$ and $|X| = |Y|$.
Output: A minimum weight perfect matching in G if G admits a perfect matching.

1. Initialize: Set $\eta(v) = 0$ for all $v \in V$. Let $M = \emptyset$.
2. If $|M| = |X|$, i.e., M is a perfect matching, **return** M.
   Otherwise, let $G_0 \coloneqq (V, E_\eta = \{e \in E \mid w_\eta(e) = 0\})$. We try to augment the matching M in G_0 using Algorithm 2, i.e., we mark all vertices $L \subseteq V$ reachable from exposed vertices in X in the oriented version of G_0, where matching edges are oriented from Y to X and non-matching edges from X to Y. If there is an augmenting path, augment M and go back to start of step 2. Otherwise, continue with step 3.
3. Let
   $$U \coloneqq \{\{x, y\} \in E\colon x \in X \cap L, y \in Y \setminus L\} \ .$$
   If $U = \emptyset$, then $(X \setminus L) \cup (Y \cap L)$ is a vertex cover of G of size strictly less than $|X|$, thus proving that G does not contain a perfect matching $\to$ STOP.
   Else
   $$\Delta \coloneqq \frac{1}{2} \min \{w_\eta(e)\colon e \in U\} \ .$$
   Update the potential as follows:
   $$\eta(v) \leftarrow \begin{cases} \eta(v) - \Delta & \text{if } v \in (X \cap L) \cup (Y \setminus L) \ , \\ \eta(v) + \Delta & \text{if } v \in (X \setminus L) \cup (Y \cap L) \ . \end{cases} \tag{3.2}$$
   Go back to step 2.

which we denote by η', is feasible.

We start by proving $\Delta > 0$, i.e, $w_\eta(e) > 0$ for $e \in U$. By Theorem 3.9, $S = (X \setminus L) \cup (Y \cap L)$ is a vertex cover in $G_0 \coloneqq (V, E_\eta)$. Hence, there is no edge $e \in E$ with $w_\eta(e) = 0$ with both endpoints outside of S, i.e., there is no $e \in U$ with $w_\eta(e) = 0$. Because η is a feasible node potential, we have $w_\eta(e) \geq 0$ for all $e \in E$ and thus $w_\eta(e) > 0$ for all $e \in U$, as desired.

It remains to show $w_{\eta'}(e) \geq 0$ for all $e \in E$. By the way how node potentials are updated, the edges $e \in E$ for which $w_{\eta'}(e) < w_\eta(e)$ are precisely the edges in U. Thus for all edges $e \in E \setminus U$ we have $w_{\eta'}(e) \geq w_\eta(e) \geq 0$. Furthermore, for $e \in U$ we have

$$w_{\eta'}(e) = w_\eta(e) - 2\Delta \geq w_\eta(e) - w_\eta(e) = 0 \ . \qquad \square$$

Lemma 3.16

Whenever Algorithm 4 is at step 2, we have $M \subseteq E_\eta \coloneqq \{e \in E\colon w_\eta(e) = 0\}$.

Proof. The only possibility how the condition may become violated is that there was an update of the node potential in step 3 from some node potential η that satisfies $M \subseteq E_\eta$ to some node potential η' for which $M \not\subseteq E_{\eta'}$. Assume by the sake of deriving a contradiction that this happens. Notice that M is a maximum cardinality matching in $G_0 = (V, E_\eta)$, because step 2 was repeated until no augmenting path existed anymore, which implies that M is of maximum

cardinality in G_0 by Theorem 3.6. Notice that an edge $e \in E$ is in G_0 but not in $G'_0 := (V, E_{\eta'})$ if and only if $w_\eta(e) = 0$ and $w_{\eta'}(e) > 0$. Observe that the edges $e \in E$ with $w_\eta(e) < w_{\eta'}(e)$ are precisely the edges with both endpoints in $S = (X \setminus L) \cup (Y \cap L)$, which is a vertex cover in G_0 of size $|S| = |M|$ by Theorem 3.9. We assumed that $M \not\subseteq E_{\eta'}$, hence there is an edge $\{x, y\} \in M$ with $x \in X \setminus L$ and $y \in Y \cap L$. However, then $\overline{S} = S \setminus \{x, y\}$ is a vertex cover in the graph $\overline{G}_0 = G_0[V \setminus \{x, y\}]$—the graph obtained from G_0 by removing x and y—and $\overline{M} = M \setminus \{x, y\}$ is a matching in $\overline{G}$, and we have

$$|\overline{M}| = |M| - 1 = |S| - 1 = |\overline{S}| + 1 \ .$$

This would imply that $\overline{G}_0$ has a vertex cover of smaller size than some matching in $\overline{G}_0$, which contradicts Observation 3.3. □

Notice that there are two ways how Algorithm 4 can finish: either in step 2 by returning a perfect matching, or in step 3 by claiming that there is no perfect matching in G due to a vertex cover of size strictly less than $|X|$. If a perfect matching M is returned, then this matching must be of minimum weight due to Lemma 3.13, which can be invoked with the final node potential η. We continue by analyzing the other stopping criterion of the algorithm, i.e., when the algorithm stops in step 3 by claiming that there is no perfect matching in G.

Proposition 3.17

If Algorithm 4 finishes in step 3, then the set $S = (X \setminus L) \cup (Y \cap L)$ is a vertex cover in G of size strictly less than $|X|$.

Proof. Notice that $U = \emptyset$ implies that S is a vertex cover in G, because edges not covered by S would need to have one endpoint in $X \cap L$ and the other one in $Y \setminus L$. However, edges with this property are precisely the edges in U.

Furthermore, we have $|S| < |X|$ due to

$$|S| = |M| < |X| \ ,$$

where the equality follows by applying König's Theorem to the graph G_0 and using the fact that S is a minimum cardinality vertex cover in G_0 due to Theorem 3.9; moreover, the inequality holds because M is not a perfect matching whenever we are in step 3. □

Hence, to show that Algorithm 4 indeed returns a minimum weight perfect matching if one exists, it suffices to prove that the algorithm terminates. One (crude) way to measure progress of the algorithm is through the size of M. We call a *phase* of Algorithm 4 the steps performed between two successive increases of the size of M. Hence, during a phase, the algorithm repeatedly goes through steps 2 and 3 until $|M|$ increases. Because the size of a perfect matching is $|X|$, we have at most $O(|X|)$ many phases. Thus, it suffices to bound the number of times that the algorithm iterates between steps 2 and 3 within the same phase. The following theorem shows that whenever M is not increased, then L is increased.

Theorem 3.18

Consider the state of Algorithm 4 between two consecutive times that the algorithm is at the start of step 2. Between these states in the algorithm, either $|M|$ or $|L|$ increased.

Proof. Assume that M is not increased. Hence between the two consecutive times we consider where the algorithm has been at the beginning of step 2, the node potential has been updated from some potential η to η'. Let $G_0 = (V, E_\eta)$ and $G'_0 = (V, E_{\eta'})$, and we denote by $L \subseteq V$ and $L' \subseteq V$ the labeled vertices with respect to G_0 and G'_0, respectively. We want to show $|L'| > |L|$. Notice that $w_\eta(e) = w_{\eta'}(e)$ for any edge $e = \{x, y\} \in E$ with both endpoints in L, because $\eta'(x) = \eta(x) - \Delta$ and $\eta'(y) = \eta(y) + \Delta$. Hence, $G_0[L] = G'_0[L]$. Thus, $L \subseteq L'$, as any vertex that was labeled in G_0 can also be labeled in G'_0 because only edges in $G_0[L] = G'_0[L]$ were used to label the vertices L in G_0. Furthermore, consider an edge $\{x, y\}$ with $x \in X$ and $y \in Y$ that achieves the minimum in the definition of $\Delta = \frac{1}{2}\min\{w_\eta(e) \colon e \in U\}$, which, because U is the set of all edges with an endpoint in $X \cap L$ and one in $Y \setminus L$, satisfies $x \in X \cap L$ and $y \in Y \setminus L$. This edge $\{x, y\}$ cannot be part of M because $w_\eta(\{x, y\}) > 0$ whereas $w_\eta(e) = 0$ for all $e \in M$. Hence, $y \in L'$ because Y can be marked in G'_0 via $\{x, y\}$. Thus $L \cup \{y\} \subseteq L'$ and therefore $|L'| > |L|$ as desired. □

The above theorem now easily implies that Algorithm 4 terminates in strongly polynomial time.

Theorem 3.19

Algorithm 4 runs in $O(n^2 m)$ time, where n and m are the number of vertices and edges of the input graph, respectively.

Proof. By Theorem 3.18, the number of times Algorithm 4 goes through step 2 without augmenting M is bounded by n, because the size of L increases every time and $|L| \leq n$. Because the number of phases is bounded by $O(n)$, the algorithm goes through step 2 at most $O(n^2)$ times without augmenting M. Furthermore, M is augmented at most $n/2$ times when going through step 2, because $n/2$ is the size of a perfect matching. Hence, step 2 is executed at most $O(n^2)$ times. Thus, also step 3 is executed at most $O(n^2)$ times. It remains to observe that step 2 and step 3 can both be performed in $O(m)$ time, thus leading to the claimed overall running time bound of $O(n^2 m)$. □

3.3.3 The Directed Chinese Postman Problem

The weighted bipartite matching problem, and in particular the assignment problem, has many natural applications where things have to be assigned or matched up.

A very classical setting is to assign a set of jobs to a set of machines of the same cardinality. The cost of processing a job depends on the machine on which the job is run. Each machine has to process exactly one job and the goal is to process all the jobs at minimum cost.

Similarly, imagine a company that has to perform a set of tasks with the available workforce, where the quality how well a task is completed depends on the assigned worker. Each worker

needs a day of work to complete any task. However, some workers tend to perform some tasks better than others, e.g., one might want to assign a very socially skilled person to handling the hotline with the goal to maximize customer satisfaction. The goal is to assign jobs to the workforce such that the quality of service is maximized.

The above examples are rather straightforward applications of the assignment problem, and similar applications can be found in a wide variety of settings. Due to its various applications, the assignment problem is one of the most important elementary combinatorial optimization problems. Furthermore, there are many further problems whose description does not immediately reveal any straightforward link to the assignment problem, even though they can be reduced to it. In the following, we discuss such an example, known as the Directed Chinese Postman Problem.

3.3.3.1 Problem setting

In the Directed Chinese Postman Problem we are given a strongly connected directed graph $G = (V, A)$ with non-negative arc lengths $\ell \colon A \to \mathbb{R}_{\geq 0}$. One classical version is to find a closed walk of minimum length that uses each arc at least once. Hence, a solution can be described by a sequence of arcs $(a_1, \ldots, a_k)$ for some $k \in \mathbb{Z}_{\geq 0}$ with $a_i \in A$ for $i \in [k]$, $\text{tail}(a_i) = \text{head}(a_{i+1})$ for $i \in [k-1]$ and $\text{tail}(a_k) = \text{head}(a_1)$, and each arc of A should appear at least once in the sequence.

We look at a slightly more general version, where, additionally to G and its arc lengths, we are given a set $U \subseteq A$ of weakly connected arcs, i.e., when looking at G as an undirected graph, then U is a connected set of arcs. The goal is to find a minimum length closed walk in G that contains each arc of U at least once. We call such a closed walk also simply a *postman tour*. This version is indeed more general as it captures the first-introduced version by setting $U = A$. Figure 3.1 shows an example instance of (this generalized version of) the Directed Chinese Postman Problem.

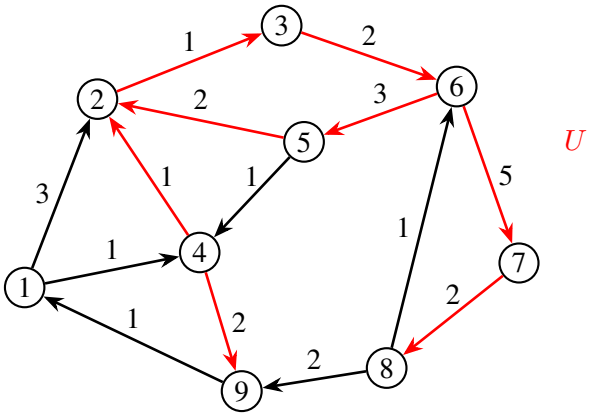


Figure 3.1: Example instance of the Directed Chinese Postman Problem, where the edges in U are marked red.

Interestingly, the condition that U has to be connected is crucial for computational reasons,

because without this condition one can show that the problem is NP-hard by a reduction from the Hamiltonian path problem. Figure 3.1 shows an example instance of the Directed Chinese Postman Problem.

One approach to design an algorithm for the directed Chinese Postman Problem is to first decide how many times to use each arc, and then to form a postman tour with the selected arcs. Of course, the chosen arcs (counted with the multiplicity with which they were chosen) must be such that one can form a postman tour out of them, and furthermore, they must contain U. There is a nice characterization, due to Euler, of when a set of arcs can be put in a sequence such that it is a closed walk. This is why graphs with this property are called *Eulerian*.

Definition 3.20

A directed graph $G = (V, A)$ is called *Eulerian* if one can order its arcs A such that they form a closed walk. Hence, each arc appears precisely once in the tour. Such a closed walk is called an *Euler walk*.

The theorem below states Euler's characterization of when a graph is Eulerian.

Theorem 3.21

A directed graph $G = (V, A)$ is Eulerian if and only if the following two conditions hold:

(i) $(V(A), A)$ is weakly connected, where $V(A) \subseteq V$ are all vertices that are an endpoint of some arc in A.

(ii) $\deg^+(v) = \deg^-(v)$ for all $v \in V$.

Furthermore, if G fulfills these conditions, an Euler walk can be found in $O(|A| + |V|)$ time.

Proof. The necessity of the conditions is obvious. Clearly, no Euler walk can be found if $(V(A), A)$ contains several connected components. Furthermore, if a vertex is visited k times by a Euler walk, then the walk must enter the vertex k times and leave it k times, hence, indegree must be the same as outdegree for every vertex.

Now assume that $G = (V, A)$ is a graph that fulfills both conditions. To construct an Euler walk in G we use the following procedure. Let $F \subseteq A$ be some subset of arcs for which we already have an Euler walk of (V, F). In particular, at the start we set $F = \emptyset$. If $F = A$, we are done. Otherwise, we show how to transform F into a larger closed walk as follows. Because G is connected and $A \neq F$, there must be a vertex v that is visited by F and not all of whose incident arcs are contained in F, i.e., $\delta(v) \not\subseteq F$. We then find a closed walk in $(V, A \setminus F)$ that contains v by starting at v and adding step by step arcs of $A \setminus F$ to the walk that were not added previously, starting with any arc in $\delta^+(v) \setminus F$. This procedure must stop at some point, and this can only happen when it reaches a vertex w such that all arcs $\delta^+(w) \setminus F$ have already been added to the walk. However, the only vertex w for which this can happen is $w = v$: If the algorithm got stuck at any other vertex $w \neq v$, then this would imply $|\delta^-(w) \setminus F| < |\delta^+(w) \setminus F|$ because before adding the last arc to the walk that reached w, the vertex w has been entered and left the same number of times. Hence $w = v$, and we thus found a closed walk using some arc set

$Q \subseteq A \setminus F$ that contains v. This walk can now be used to extend the closed walk we have on F, by gluing these two walks together, i.e., a new closed walk is obtained over $F \cup Q$ by first traversing the closed walk on F starting at v and then traversing the walk on Q. This procedure can then be repeated until all arcs in A are contained in the closed walk.

Finally, we give a brief explanation how the above procedure can be implemented to run in $O(|A| + |V|)$ time. As usual, we assume that the graph is given through an incidence list. We slightly enhance the incidence list representation by making sure that for each vertex v, we have two lists, one for all arcs going out of v, and one for all arcs entering v. One can observe that such an enhanced incidence list representation with two lists per vertex can be built in linear time from one that has a single incidence list per vertex.

When constructing a new closed walk in some graph $(V, A \setminus F)$ to extend the current closed walk, we always choose the first available outgoing arc of the current vertex. To make this efficient, we keep a pointer for each vertex, pointing to the first outgoing arc not used so far. Furthermore, to find a vertex v on which to extend the current closed walk, we keep a list of all vertices touched by the closed walk constructed so far that still have outgoing arcs not used so far. One can check that this way, the algorithm can be implemented in linear time. □

Using the characterization given by Theorem 3.21, the Directed Chinese Postman Problem reduces to finding a minimum cost multiset of arcs to be added to U such that an Eulerian graph is obtained. To formalize things, let us denote, for $a \in A$, by $x(a) \in \mathbb{Z}_{\geq 0}$ the number of times we will traverse a in our postman tour. Hence, we want to find values $x(a)$ such that $x(a) \geq 1$ for $a \in U$ because we have to traverse all arcs in U. Furthermore, the condition that the indegree of the chosen arcs (considered with multiplicities $x(a)$) is equal to the outdegree translates into

$$\sum_{a \in \delta^+(v)} x(a) = \sum_{a \in \delta^-(v)} x(a) \quad \forall v \in V \ .$$

Our objective is to minimize the total length, i.e.,

$$\min \sum_{a \in A} \ell(a) x(a) \ .$$

According to Theorem 3.21, we would also have to make sure that the arcs $\{a \in A \colon x(a) \geq 1\}$ to traverse are connected. However, it turns out that we can ignore this condition due to the following. Consider some vector $x \colon A \to \mathbb{Z}_{\geq 0}$ that fulfills the above-mentioned conditions, i.e., $x(a) \geq 1$ for $a \in U$ and the indegrees equal the outdegrees with respect to x. If $\{a \in A \colon x(a) \geq 1\}$ consists of several connected components, then it suffices to consider the connected component containing U. Because U is connected, all of U must be in the same connected component of $\{a \in A \colon x(a) \geq 1\}$. By only considering the arcs $W \subseteq \{a \in A \colon x(a) \geq 1\}$ in the component containing U, we obtain a solution $y \colon A \to \mathbb{Z}_{\geq 0}$ given by

$$y(a) = \begin{cases} x(a) & \text{if } a \in W, \\ 0 & \text{otherwise.} \end{cases}$$

The vector y indeed corresponds to an Eulerian graph containing all arcs of U. Moreover, y is no more expensive than x, because all arc lengths are non-negative. Hence, the Directed Chinese

Postman Problem reduces to the question of how to complete the graph (V, U) in a minimum length way by adding arcs (maybe multiple times) to equalize the indegree and outdegree of each vertex. For $v \in V$, denote by $\alpha(v)$ the excess indegree when just considering the arcs in U, i.e.,

$$\alpha(v) := |\delta^-(v) \cap U| - |\delta^+(v) \cap U| \ .$$

Figure 3.2 shows the values of α for the example instance depicted in Figure 3.1.

Formalizing the above, the Directed Chinese Postman Problem reduces to the following.

Problem 3.22

Given is a directed graph $G = (V, A)$ with non-negative arc lengths $\ell \colon A \to \mathbb{R}_{\geq 0}$ and a function $\alpha \colon V \to \mathbb{Z}$ with $\alpha(V) := \sum_{v \in V} \alpha(v) = 0$. The goal is to find a multiset of arcs[a], i.e., a function $y \colon A \to \mathbb{Z}_{\geq 0}$, that satisfies

$$\sum_{a \in \delta^+(v)} y(a) - \sum_{a \in \delta^-(v)} y(a) = \alpha(v) \quad \forall\, v \in V \tag{3.3}$$

and has minimum length $\sum_{a \in A} \ell(a) y(a)$ among all such multisets.

[a] Here, we represent a multisubset of arcs from A as a function $y \colon A \to \mathbb{Z}_{\geq 0}$, where the interpretation is that for every $a \in A$, $y(A)$ is the number of times a is contained in the multisubset.

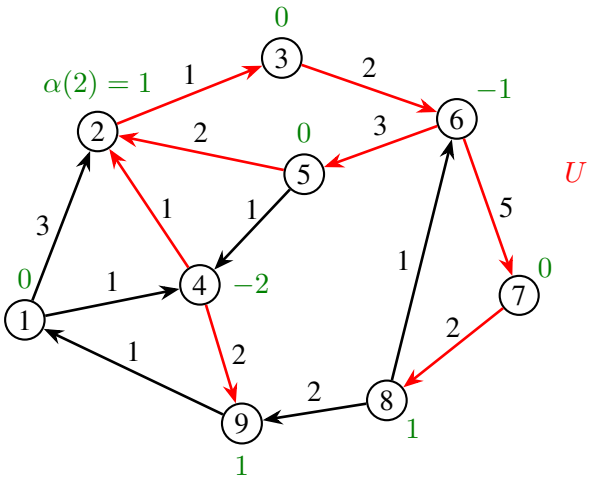


Figure 3.2: Problem instance for Problem 3.22 that corresponds to the example in Figure 3.1. Notice that the set U of red arcs is not part of the input of Problem 3.22. The input of Problem 3.22 requires the green α-values instead, which can be obtained from the set U.

Problem 3.22 is a minimum cost flow problem with demands given by α and costs by ℓ. Hence, this problem can be solved in polynomial time by a minimum cost flow algorithm. Minimum cost flow problems can be seen to be a generalization of assignment problems. In the

following, we show that Problem 3.22 can even be reduced to an assignment problem. This allows us to solve the problem through the Hungarian Method, or any other minimum weight perfect matching algorithm, and we do not need to rely on minimum cost flow algorithms, which need to be able to tackle a larger class of problems.

To transform Problem 3.22, we start by the following result that allows us to interpret a solution satisfying (3.3) as a collection of paths and cycles. This is a classical result from the theory of network flows, also known as the *flow decomposition theorem*.

Lemma 3.23

Let $G = (V, A)$ be a directed graph, and let $\alpha\colon V \to \mathbb{Z}$ with $\alpha(V) = 0$. Then a multiset represented by $y\colon A \to \mathbb{Z}_{\geq 0}$ satisfies (3.3) if and only if its arcs can be partitioned into a set of paths $P_1, \dots, P_p \subseteq A$ and cycles $C_1, \dots, C_q \subseteq A$ satisfying the following.

(i) Every path P_i starts at some vertex u with $\alpha(u) > 0$ and ends at some vertex $v \in V$ with $\alpha(v) < 0$.

(ii) For $v \in V$, if $\alpha(v) > 0$, then there are precisely $\alpha(v)$ paths starting at v, and if $\alpha(v) < 0$, then there are precisely $-\alpha(v)$ paths ending at v.

To clarify, we call a set of paths $P_1, \dots, P_p \subseteq A$ and cycles $C_1, \dots, C_q \subseteq A$ a partitioning of y if

$$|\{i \in [p]\colon a \in P_i\}| + |\{j \in [q]\colon a \in C_j\}| = y(a) \quad \forall a \in A \ .$$

Proof. The *if*-direction of the lemma is easy to see. Namely, if there is a partitioning of y into paths and cycles satisfying the condition of the lemma, then y satisfies (3.3).

For the *only if*-direction, consider a multiset given by $y\colon A \to \mathbb{Z}_{\geq 0}$ that satisfies (3.3). We construct paths and cycles iteratively as follows. We start at any vertex $v \in V$ with $\alpha(v) > 0$, if such a vertex exist. Starting from v, we build a walk as in the proof of Theorem 3.21, i.e., we follow unused arcs in y in an arbitrary way. As soon as we visit any vertex twice, we found a cycle C_1. We decrease y by one unit on all arcs on C_1 and continue. If no vertex is visited twice and the procedure gets stuck, then it must end at a vertex w with $\alpha(w) < 0$ and we found a path P_1. Again we can iterate the procedure on the multiset obtained by removing one unit from $y(a)$ for each $a \in P_1$. As soon as there is no excess vertex left, we continue the procedure at any vertex. One can observe that this will partition the remaining arcs into cycles. □

By Lemma 3.23, Problem 3.22 is equivalent to finding a minimum length collection of cycles and paths satisfying the conditions of Lemma 3.23. Since lengths are non-negative, there is no need to include cycles in the collection. Hence, we concentrate on finding a minimum length solution to Problem 3.22 by finding a minimum length collection of paths $P_1, \dots, P_p$, where we have $\alpha(v)$ paths starting at v for any vertex $v \in V$ with $\alpha(v) > 0$ and $-\alpha(v)$ paths ending at $v \in V$ if $\alpha(v) < 0$. Thus, the total number p of paths will be

$$p = \sum_{\substack{v \in V\colon \\ \alpha(v) > 0}} \alpha(v) \ .$$

Furthermore, a minimum length collection of paths $P_1, \dots, P_p$ will clearly only contain shortest paths, i.e., if P_1 is a path from u to v, then it is a shortest path from u to v. Let $d_\ell(u, v)$ be the distance between u and v with respect to the lengths ℓ.

Hence, to determine an optimal solution to Problem 3.22, it remains to determine an optimal way to define the start and endpoints of the p paths $P_1, \dots, P_p$, or more precisely, to determine which start points are *matched* to which endpoints by an optimal set of paths $P_1, \dots, P_p$. To formalize this viewpoint, we use that every path P_i can be represented concisely by a pair of vertices (s_i, t_i), containing its start point $s_i \in V$ and end endpoint $t_i \in V$.

To summarize, the Directed Chinese Postman Problem reduces to the following problem.

Problem 3.24

Let $G = (V, A)$ be a directed graph with arc lengths $\ell\colon E \to \mathbb{R}_{\geq 0}$, and let $\alpha\colon V \to \mathbb{Z}$ with $\alpha(V) = 0$. Find a collection of $p := \sum_{v \in V : \alpha(v) > 0} \alpha(v)$ pairs $(s_1, t_1), \dots, (s_p, t_p) \in V \times V$ that satisfy

$$\begin{aligned} |\{i \in [p] \colon s_i = v\}| &= \alpha(v) \quad \text{for all } v \in V \text{ with } \alpha(v) > 0 \ , \\ |\{i \in [p] \colon t_i = v\}| &= -\alpha(v) \quad \text{for all } v \in V \text{ with } \alpha(v) < 0 \ , \end{aligned}$$

and minimizes

$$\sum_{i=1}^{p} d_\ell(s_i, t_i)$$

under these constraints.

Notice that Problem 3.24 can be modeled as a generalized version of an assignment problem as follows. We construct an undirected edge-weighted bipartite (multi-)graph $H = (V_1 \dot\cup V_2, F)$ with

$$\begin{aligned} V_1 &= \{v \in V \colon \alpha(v) > 0\} \ , \\ V_2 &= \{v \in V \colon \alpha(v) < 0\} \ , \end{aligned}$$

and for each $v_1 \in V_1, v_2 \in V_2$, the edge set F contains $\min\{\alpha(v_1), -\alpha(v_2)\}$ many parallel edges between v_1 and v_2, all having weight $d_\ell(v_1, v_2)$.

The left-hand side picture in Figure 3.3 shows the graph H for our example Directed Chinese Postman Problem highlighted in Figure 3.1, which leads to α-values as shown in Figure 3.2.

One can observe that Problem 3.24 translates into finding a minimum weight set of edges $D \subseteq F$ in H such that the degree of each vertex $v \in V_1 \cup V_2$ with respect to D is equal to $|\alpha(v)|$. Hence, this can be seen as a generalized minimum weight perfect bipartite matching problem, with the difference that instead of needing precisely one matching edge incident to each vertex v, there have to be precisely $|\alpha(v)|$ edges incident to v. However, this problem can easily be cast as a classical minimum weight matching problem in a bipartite graph by replacing each vertex $v \in V_1 \cup V_2$ by $|\alpha(v)|$ many copies. More precisely, we construct a complete bipartite graph

Figure 3.3: Reduction of the Directed Chinese Postman Problem shown in Figure 3.1 to a generalized matching problem on the graph H as described after Problem 3.24 (left), and then to a minimum weight assignment problem on the graph H' (right). The minimum weight assignment is highlighted in blue.

$H' = (V_1' \dot\cup V_2', F')$ (without any parallel edges), where

$$V_1' \coloneqq \bigcup_{v_1 \in V_1} \left\{ v_1^1, \dots, v_1^{\alpha(v_1)} \right\} ,$$
$$V_2' \coloneqq \bigcup_{v_2 \in V_2} \left\{ v_2^1, \dots, v_2^{-\alpha(v_2)} \right\} ,$$

and the weight of an edge $\{v_1^i, v_2^j\}$ for $v_1^i \in V_1'$ and $v_2^j \in V_2'$ is set to $d_\ell(v_1, v_2)$. A minimum weight perfect matching M in H' can thus be mapped back to a matching on H by mapping any edge $\{v_1^i, v_2^j\} \in M$ to one of the parallel edges in H between v_1 and v_2 (while avoiding to map two edges of M to the same edge in H). The right-hand side picture in Figure 3.3 shows the graph H' for the instance depicted in Figure 3.1.

To summarize, the Directed Chinese Postman Problem can thus be solved by first determining a minimum weight perfect matching M in H'. Then, each $\{v_1^i, v_2^j\} \in M$ is replaced by a shortest path from v_1 to v_2. This leads to a collection of shortest paths $P_1, \dots, P_p$. Finally, the multiset obtained by putting together all paths $P_1, \dots, P_p$ and U is an Eulerian graph. It remains to determine any Euler walk in this graph, which can be done efficiently by Theorem 3.21. By our discussion above, such a tour is an optimal Chinese Postman tour. Figure 3.4 shows a minimum cost set of arcs that corresponds to an optimal Chinese Postman tour for the problem depicted in Figure 3.1. The arc set in Figure 3.4 corresponds to the minimum weight bipartite matching shown in Figure 3.3.

To make sure that the above algorithm is indeed efficient, we have to ensure that the auxiliary minimum weight perfect matching problem, as well as the graph on which we have to find the Euler walk at the end, are polynomially bounded. The number of vertices of H' is equal to $\sum_{v \in V} |\alpha(v)| \leq 2|U| \leq 2|A|$. Hence, this graph is indeed polynomially bounded in the input. The graph on which we have to find an Euler walk consists of the multi-union of U and up to $|A|$ shortest paths in G. This leads to a total number of arcs bounded by $O(|V||A|)$. Hence, this

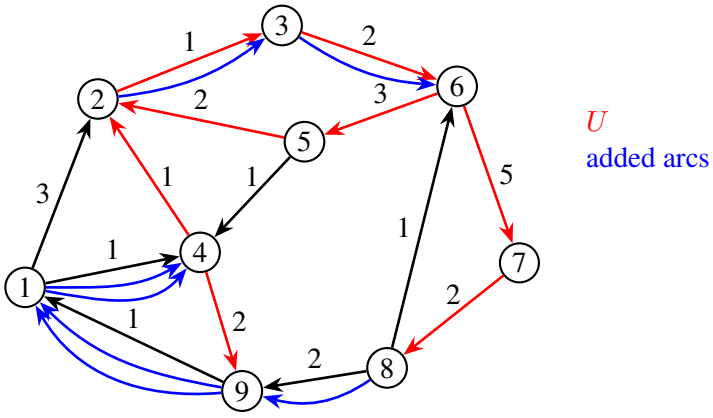


Figure 3.4: An optimal completion (blue arcs) of U (red arcs) to a multiset of arcs that can be traversed as a tour, leading to an optimal Chinese Postman tour for the example shown in Figure 3.1.

step can also be performed in polynomial time.

3.4 Maximum cardinality matchings (in non-bipartite graphs)

We now consider matchings in general, i.e., not necessarily bipartite, graphs. We focus on the case of unweighted matchings and present Edmonds' algorithm, which is an efficient algorithm to find a maximum cardinality matching. Without loss of generality, we assume that all graphs in this section are simple, i.e., they contain neither parallel edges nor loops.

Before discussing Edmonds' matching algorithm, we briefly discuss an optimality certificate for maximum cardinality matchings. First, this allows us to draw parallels to results of similar spirit in the case of bipartite graphs, notably König's Theorem. This, in turn, allows for highlighting some crucial differences faced when dealing with non-bipartite matchings. Moreover, like in the bipartite case, we show the correctness of algorithmic results—in this case Edmonds' algorithm—by showing that the algorithm reveals an optimality certificate as a byproduct.

3.4.1 A certificate of optimality for maximum cardinality matchings

König's Theorem provided a min-max relation in bipartite graphs between maximum cardinality matchings and minimum cardinality vertex covers. In non-bipartite graphs, it is not true anymore that a maximum cardinality matching has the same size as a minimum cardinality vertex cover, as can easily be verified by considering a graph G consisting of a single triangle. Here, $\nu(G) = 1$ but $\tau(G) = 2$. For a general graph G, only $\tau(G) \leq 2\nu(G)$ holds, which can be checked by observing that, for any maximum cardinality matching M, the set of all non-exposed vertices S forms a vertex cover in G with $|S| = 2|M| = 2\nu(G)$.

Hence, a natural question is whether there is a more general certificate that can help us to prove that some matching in a general graph is of maximum cardinality. It turns out that König's Theorem can be extended to non-bipartite graphs through the so-called Tutte-Berge formula. The Tutte-Berge formula is stated in the theorem below, where for a graph $G = (V, E)$ and a vertex set $S \subseteq V$, we denote by $q_G(S)$ the number of odd connected components of $G[V \setminus S]$.

Theorem 3.25: Tutte-Berge formula

For any undirected graph $G = (V, E)$,

$$\nu(G) = \frac{1}{2}\left(|V| - \max_{S \subseteq V}\left(q_G(S) - |S|\right)\right) \quad .$$

We will later prove Theorem 3.25 algorithmically through Edmonds' maximum cardinality matching algorithm. As a sanity check, let us first observe that the right-hand side of the above equality is integral.

Lemma 3.26

For any graph $G = (V, E)$ and $S \subseteq V$,

$$q_G(S) - |S| \equiv |V| \pmod 2 \quad .$$

Proof. Notice that the statement is equivalent to

$$q_G(S) \equiv |V| - |S| \pmod 2 \ .$$

The above statement holds because if $|V| - |S|$ is odd, then $G[V \setminus S]$ must have an odd number of odd components. Similarly, if $|V| - |S|$ is even, then $G[V \setminus S]$ must have an even number of odd components. □

We now prove that the right-hand side of Theorem 3.25 is an upper bound on $\nu(G)$, the cardinality of a maximum cardinality matching in G. This already shows that if we are able to find a matching M and a vertex set $S \subseteq V$ that fulfill the equation in Theorem 3.25, then M must be a maximum cardinality matching in G. Moreover, the proof of this consequence of the Tutte-Berge formula allows for building up important intuition about the formula.

Lemma 3.27

$$\nu(G) \leq \frac{1}{2}\left(|V| - \max_{S \subseteq V}(q_G(S) - |S|)\right) \ .$$

The above lemma turns out to be not hard to verify. Interestingly, when dealing with such characterizations that allow for certifying the optimality of some solution (here matchings), one of the two inequalities is typically easy to check. Remember that this was the same for König's Theorem, where $\nu(G) \leq \tau(G)$ was easy to check. This is no coincidence because one of the inequalities can typically be seen as a special case of a much more general result in linear programming known as *weak duality*, which is not hard to show even in the very general context of linear programming.

Proof of Lemma 3.27. Let M be a matching in G and let $S \subseteq V$. We have to show

$$|M| \leq \frac{1}{2}\left(|V| - q_G(S) + |S|\right) \ .$$

Let us partition M into the set of edges $M_1 \subseteq M$ with both endpoints in $V \setminus S$, and the set of edges $M_2 \subseteq M$ with at least one endpoint in S. Clearly

$$|M_2| \leq |S| \ . \tag{3.4}$$

We will now derive an upper bound on $|M_1|$. Note that each odd connected component in $G[V \setminus S]$ must have at least one M_1-exposed vertex. Hence,

$$|M_1| \leq \frac{1}{2}\left(|V \setminus S| - q_G(S)\right) = \frac{1}{2}\left(|V| - |S| - q_G(S)\right) \ .$$

Combining the above inequality with (3.4) we obtain, as desired,

$$\begin{aligned} |M| = |M_1| + |M_2| &\leq |S| + \frac{1}{2}\left(|V| - |S| - q_G(S)\right) \\ &= \frac{1}{2}\left(|V| - q_G(S) + |S|\right) \ . \end{aligned}$$

□

As mentioned, analogous to König's Theorem for bipartite graphs, Lemma 3.27 gives us a means to certify that a matching M is of maximum cardinality, namely by finding a set $S \subseteq V$ with $|M| = \frac{1}{2}(|V| - q_G(S) + |S|)$. Theorem 3.25 then guarantees that such a certificate can always be found.

We later present Edmonds' algorithm to efficiently find a maximum cardinality matching. As a byproduct, this algorithm will give us—apart from a maximum cardinality matching M—also a set $S \subseteq V$ satisfying $|M| = \frac{1}{2}(|V| - q_G(S) + |S|)$, thus proving Theorem 3.25.

Moreover, we highlight that König's Theorem can elegantly be derived from the Tutte-Berge formula, showing that the Tutte-Berge formula can indeed be interpreted as a generalization of König's Theorem to general (not necessarily bipartite) graphs. We leave the proof of this connection as an exercise.

3.4.2 Edmonds' maximum cardinality matching algorithm

Even in non-bipartite graphs, it is possible to find a maximum cardinality matching, and even a maximum weight matching, in polynomial time. The first and still most prominent algorithm to find efficiently a maximum cardinality matching is due to Edmonds. Analogous to the maximum cardinality matching algorithm that we have seen for bipartite graphs, Edmonds' algorithm is motivated by Theorem 3.6, i.e., a matching is of maximum cardinality if and only if there is no augmenting path. To find augmenting paths with respect to some matching M, Edmonds' algorithm starts at any exposed vertex v and tries to determine which vertices can be reached from v along some alternating path.

However, finding augmenting paths in a non-bipartite graph faces several hurdles that do not exist in the bipartite case. Before introducing Edmonds' algorithm, we start by highlighting some of the issues faced when trying to find augmenting paths in non-bipartite graphs. This helps to better understand the problem and the design of Edmonds' maximum cardinality matching algorithm.

3.4.2.1 Challenges in finding an augmenting path in non-bipartite graphs

Finding augmenting paths in non-bipartite graphs can be nontrivial. In Figures 3.5 and 3.6, we show a few examples that highlight some of these difficulties that did not exist in the bipartite case.

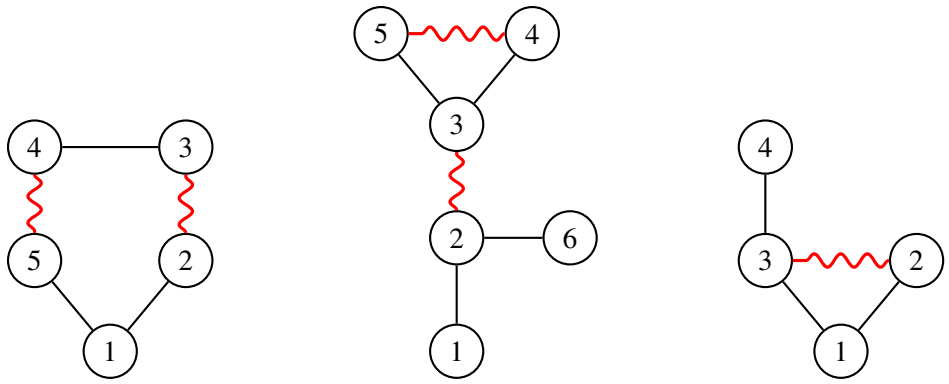


Figure 3.5: Three example graphs where simple exploration algorithms to find an augmenting path risk to fail. The current matching M is always indicated as red wiggly edges, and we denote by X the set of exposed vertices. The first example highlights that there may be a danger to confuse M-alternating X-X walks with same start and endpoint (like the walk along the vertices $1 \to 2 \to 3 \to 4 \to 5 \to 1$) as M-augmenting paths. The second example shows that even M-alternating X-X walks with different start and endpoint ($1 \to 2 \to 3 \to 4 \to 5 \to 3 \to 2 \to 6$) may not be M-augmenting. In the third example, there is an M-augmenting path from vertex 1 to 4. However, there are two ways to reach vertex 3 from vertex 1 through an M-alternating path, and only one of them can be extended to an M-augmenting path from vertex 1 to 4.

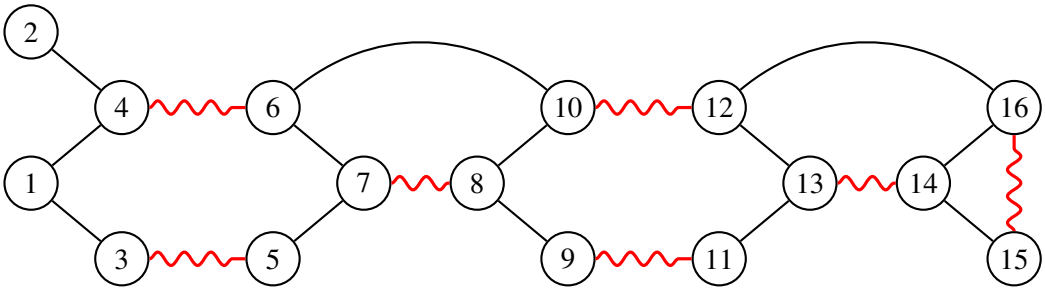


Figure 3.6: This example admits an M-augmenting path along the vertices $1 \to 3 \to 5 \to 7 \to 8 \to 9 \to 11 \to 13 \to 14 \to 15 \to 16 \to 12 \to 10 \to 6 \to 4 \to 2$. However, finding this path efficiently seems nontrivial, because when we start to explore from the exposed vertex 1, then it is important to reach at some point vertex 13 by an alternating path using the edges $\{3, 5\}$ and $\{9, 11\}$. If $\{4, 6\}$ or $\{10, 12\}$ is used on an alternating path to reach 13, then it is impossible to complete this alternating path to an M-augmenting path. This nicely illustrates some dangers with simple labeling algorithms, and one has to be careful when remembering the past of the labeling algorithm, because there may be exponentially many different options to reach a certain vertex through M-alternating paths.

3.4.2.2 Edmonds' algorithm

As we saw in the previous section, finding augmenting paths in non-bipartite graphs seems significantly more difficult than in bipartite ones. A key difference when constructing augmenting paths through a simple procedure is that the same vertex may be used twice, for example once by arriving at the vertex via a matching edge and a second time via a non-matching edge. In Edmonds' algorithm, we allow for this to happen, but then, instead of augmenting the current matching, we will modify the graph. We therefore work with alternating walks instead of alternating paths. Alternating walks are defined analogously to alternating paths with the only difference that a vertex can be visited more than once, which is implied by the definition of a walk.

Definition 3.28: Alternating walk

For a graph G and a matching M of G, a walk in G is called *M-alternating*, or simply *alternating* when M is clear from context, if its edges alternate between M and $E \setminus M$ when traversing the walk.

Let $X \subseteq V$ be the set of all exposed vertices. A key operation in Edmonds' algorithm is to compute a shortest M-alternating X-X walk of positive length, i.e., an M-alternating walk of positive length starting at some vertex in X and ending at some vertex in X. Such a walk can be computed efficiently, for example as follows. Consider the directed graph $D = (V, A)$ with

$$A = \{(u, v) \colon \exists x \in V \text{ with } \{u, x\} \in E \setminus M \text{ and } \{x, v\} \in M\} \ .$$

We compute an M-alternating walk by computing a path in D. One can think of an arc $(u, v) \in A$ as a placeholder for the pair of edges $\{u, x\} \in E \setminus M$ and $\{x, v\} \in M$. Now, a shortest M-alternating X-X walk in G of positive length can be obtained by first computing a shortest X-$N(X)$ path P_D in D, where $N(X) = \{u \in V \setminus X \colon \exists x \in X \text{ with } \{u, x\} \in E\}$ denotes, as usual, the set of all neighbors of X in G. This path P_D corresponds to an M-alternating walk in G starting at some vertex in X and ending at a vertex in $N(X)$. Hence, because the endpoint of the walk is in $N(X)$, this M-alternating walk can be extended to an M-alternating X-X walk P by appending an edge that connects its endpoint to a vertex in X. Hence, $|P| = 2|P_D| + 1$ and it is not hard to observe that P must be a shortest M-alternating X-X walk because P_D is a shortest X-$N(X)$ walk in D.

However, as we observed earlier, even if we obtain a shortest M-alternating X-X walk of positive length, this does not imply that this walk can be used to augment the current matching. It can even be that there is an M-alternating X-X walk of positive length even though M is of maximum cardinality already. (The first and second graph in Figure 3.5 exemplify this.)

As we will argue later, whenever any M-alternating X-X walk of positive length that is not a path contains a particular structure, known as an *M-flower*, and which can be exploited algorithmically. For simplicity and convenience, we represent in the following a (M-alternating) walk as the sequence of the vertices that it traverses. For example, $W = (v_1, v_2, v_3)$ represents the walk going from vertex v_1 over vertex v_2 to vertex v_3, which is its endpoint.

Definition 3.29: M-flower, M-blossom

For an undirected graph $G = (V, E)$ and a matching $M \subseteq E$, an M-alternating walk $(v_0, v_1, \ldots, v_t)$ is an *M-flower* if all of the following hold:

(i) t is odd,
(ii) $v_0, \ldots, v_{t-1}$ are distinct,
(iii) $v_0 \in X$, where $X \subseteq V$ is the set of M-exposed vertices in G, and
(iv) $v_t = v_i$ for some even $i < t$.

The cycle $(v_i, \ldots, v_t)$ is called an *M-blossom* (associated with the M-flower), and the path $(v_0, \ldots, v_i)$ is called the *stem* (of the M-flower).

See Figure 3.7 for an illustration of the above definitions.

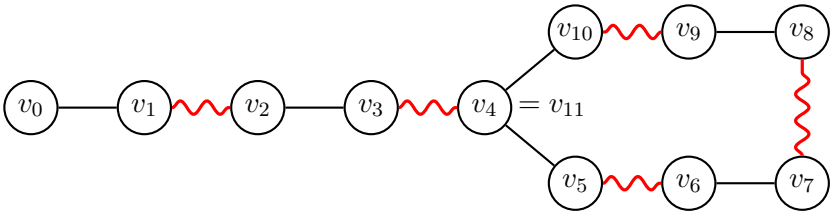


Figure 3.7: Example of an M-flower $(v_0, \ldots, v_{11})$ with M-blossom $(v_4, \ldots, v_{11})$ with respect to the matching M consisting of the red wiggly edges. In this example, the indices t and i as described in Definition 3.29 are equal to $t = 11$ and $i = 4$.

The core idea of Edmonds' algorithm is that, whenever there is a flower, then one can reduce the problem of finding an augmenting path to a smaller graph obtained by shrinking the blossom.

Let us first discuss what we mean by shrinking. Let $G = (V, E)$ be a graph and $B \subseteq V$. Then the graph obtained from G by shrinking (or contracting) B is denoted by G/B and is defined as follows. The vertex set of G/B is $(V \setminus B) \cup \{B\}$, and for each edge $e \in E$ there is an edge in G/B obtained by replacing any endpoint of e that lies in B by B, and by ignoring thus created loops. The new edge in G/B is called the *image* of the original edge e. For simplicity, we denote both edges by e. Furthermore, for any matching M, let M/B be the image of all edges in M that do not have both endpoints in B.

The key structural result that justifies the contracting of blossoms is the following.

Theorem 3.30

Let B be an M-blossom in G. Then M is a maximum cardinality matching in G if and only if M/B is a maximum cardinality matching in G/B.

Proof. Let $B = (v_i, \ldots, v_t)$ be an M-blossom and let $(v_0, \ldots, v_t)$ be an M-flower whose associated M-blossom is B.

$\Rightarrow$) We show this direction by proving its contrapositive, i.e., we assume that M/B is not a maximum cardinality matching in G/B and show that this implies that M is not a maximum

cardinality matching in G. If M/B is not of maximum cardinality, then there is an M/B-augmenting path P in G/B. If P does not traverse the vertex B of G/B, then P is also an M-augmenting path in G; hence, we obtain that M is not of maximum cardinality as desired. Otherwise, we may assume that P enters B with some edge $\{u, B\}$ not in M/B (otherwise, traverse the path P along the opposite direction), and, consequently, either B is an endpoint of P or P leaves B using the only matching edge incident with it. Notice that if there is a matching edge incident with B, then it must be $\{v_i, v_{i-1}\}$. (Note that it could also be that there is no vertex v_{i-1}, which happens if $v_0 = v_i$, i.e., the stem consists of a single vertex. In this case, no matching edge is incident with B.) Let $\{u, v_j\} \in E \setminus M$, where $j \in \{i, i+1, \dots, t\}$, be the edge with which P is entering B. We modify P as follows to obtain an M-augmenting path in G:

$$\begin{aligned} &\text{If } j \text{ is odd, replace vertex } B \text{ in } P \text{ by } v_j, v_{j+1}, \dots, v_t. \\ &\text{If } j \text{ is even, replace vertex } B \text{ in } P \text{ by } v_j, v_{j-1}, \dots, v_i. \end{aligned} \tag{3.5}$$

In both cases we obtain an M-augmenting path in G, implying that M is not a maximum cardinality matching in G.

$\Leftarrow$) We can assume without loss of generality that $v_i = v_0$, i.e., v_i is exposed, by replacing M with $M\Delta\{\{v_0, v_1\}, \dots, \{v_{i-1}, v_i\}\}$. This does not change the theorem. We again show the contrapositive. Hence, assume that M is not of maximum cardinality and let $P = (u_0, \dots, u_s)$ be an M-augmenting path in G. If P does not contain any vertices in B, then P is also an M/B-augmenting path in G/B, showing as desired that M/B is not a maximum cardinality matching in G/B. Otherwise, if P contains at least one vertex of B, we assume without loss of generality $u_0 \notin B$, which can be achieved by reversing the direction of P if necessary. Let u_j be the first vertex of P in B. Then $(u_0, \dots, u_{j-1}, B)$ is an M/B-augmenting path in G/B. Hence, M/B is not a maximum cardinality matching in G/B. □

Theorem 3.31

Let $P = (v_0, \dots, v_t)$ be a shortest M-alternating X-X walk of positive length. Then either P is an M-augmenting path or $(v_0, \dots, v_j)$ is an M-flower for some $j \leq t$.

Proof. If P is a path, then P is M-augmenting. Hence, assume that at least one vertex is visited twice in P. Choose two indices $i, j \in \{0, \dots, t\}$ with $i < j$, $v_i = v_j$, and such that j is as small as possible. We claim that $(v_0, \dots, v_j)$ is a flower with blossom $(v_i, \dots, v_j)$.

We observe first that such indices satisfy that $j - i$ is odd. Indeed, if $j - i$ was even, then P would not have been a shortest M-alternating X-X walk of positive length, because a shorter M-alternating X-X walk of positive length is obtained by removing the cycle $(v_i, \dots, v_j)$ from P, which leads to the walk $(v_0, \dots, v_{i-1}, v_i, v_{j+1}, v_{j+2}, \dots v_t)$. Hence, $j - i$ must be odd. For $(v_0, \dots, v_i)$ to be a flower, it remains to show that i is even, which corresponds to property (iv) in Definition 3.29. To this end, notice that because $j - i$ is odd and P is M-alternating, we have that $\{v_i, v_{i+1}\}$ and $\{v_{j-1}, v_j\}$ are either both in M or both in $E \setminus M$. As $v_i = v_j$ and M is a matching, we have $\{v_i, v_{i+1}\}, \{v_{j-1}, v_j\} \notin M$. Finally, because P is alternating and v_0 is exposed, this implies that i must be even, as desired. □

Hence, Theorem 3.31 leads to a natural strategy to compute an M-augmenting path, namely

by determining a shortest M-alternating X-X walk of positive length. This walk is either an M-augmenting path, in which case the current matching M can be augmented, or, if not, then it contains a flower and we can shrink its blossom and recurse. Algorithm 5 formalizes this procedure.

Algorithm 5: Edmonds' algorithm to find an augmenting path.

Input : Undirected graph $G = (V, E)$ and matching M in G.
Output: An M-augmenting path in G if there is one.

(i) Determine a shortest M-alternating X-X walk $P = (v_0, \dots, v_t)$ of positive length. If no such walk exists, then **stop** (i.e., return that there is no augmenting path).

(ii) If P is a path $\rightarrow$ **return** P.
Otherwise, let $j \in [t]$ be smallest index such that $(v_0, \dots, v_j)$ is a flower with M-blossom B. Apply Algorithm 5 recursively to G/B and M/B, giving an M/B-augmenting path P in G/B. Expand P to an M-augmenting path in G (see (3.5)).

Figure 3.8 shows on an example how Algorithm 5 finds an augmenting path.

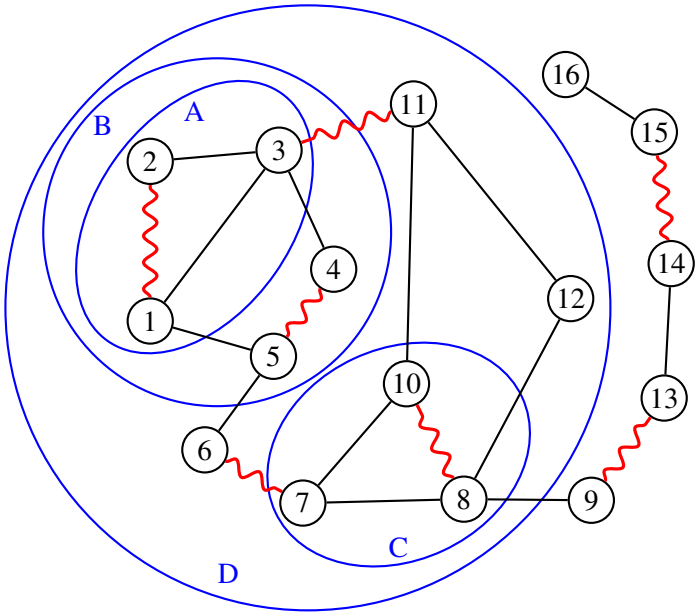


Figure 3.8: Illustration of how Algorithm 5 finds an M-augmenting path. The first shortest M-alternating X-X walk traverses the vertices $12 \rightarrow 11 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow 11 \rightarrow 12$ and finds the flower $(12, 11, 3, 2, 1, 3)$ whose blossom is highlighted by the blue ellipse A. After shrinking A and continuing the algorithm we find successively blossom B, then C, and finally D, all discovered through a shortest X-X walk, which all start and end in vertex 12 in this example. After having shrunk D, the $(((M/A)/B)/C)/D$-augmenting path $(D, 9, 13, 14, 15, 16)$ is found in $(((G/A)/B)/C)/D$, which, after expansion, leads to the M-augmenting path $(12, 11, 3, 4, 5, 6, 7, 10, 8, 9, 13, 14, 15, 16)$.

Theorem 3.32

Given an undirected graph $G = (V, E)$ and a matching M in G, Algorithm 5 finds an M-augmenting path if there is one. Furthermore, its running time is bounded by $O(|V||E|)$.

Proof. Theorem 3.31 guarantees that step (ii) is well defined, i.e., one indeed either finds an M-augmenting path or an M-flower. Hence, the algorithm makes progress in the sense that whenever step (ii) is performed without finding an augmenting path, the recursive call is applied to a graph with a strictly smaller size than $|V|$. Hence, Algorithm 5 does terminate. Furthermore, Theorem 3.30 guarantees that if G admits an augmenting path, then so does any graph used in a recursive call. Hence, in this case the algorithm cannot stop at step (i) and will thus return an M-augmenting path in G, as promised.

To prove that the running time is bounded by $O(|V||E|)$ it suffices to observe that one execution of step (i) or step (ii) can be performed in $O(|E|)$ time. (We leave the details of this observation as an exercise.) Furthermore, there cannot be more than $O(|V|)$ recursive calls, because every time a blossom is contracted, the number of vertices in the graph reduces by at least 2. Hence, the total running time is bounded by $O(|V||E|)$, as claimed. □

To complete Edmonds' algorithm, it now remains to repeatedly use Algorithm 5 to find M-augmenting paths and to augment the current matching through these paths. Algorithm 6 provides pseudocode that summarizes the procedure. Its correctness follows from Theorem 3.6, like in the bipartite case.

Algorithm 6: Edmonds' algorithm to find a maximum cardinality matching.

Input : Graph $G = (V, E)$.
Output: A maximum cardinality matching M in G.

(i) Initialize: $M = \emptyset$.
(ii) Use Algorithm 5 to find an M-augmenting path P.
If no M-augmenting path is found $\rightarrow$ **return** M.
Otherwise, $M \leftarrow M \Delta P$, and repeat step (ii).

Theorem 3.33

The running time of Algorithm 6, i.e., Edmonds' Algorithm, is bounded by $O(|V|^2|E|)$.

Proof. There are $O(|V|)$ augmentation steps, each taking $O(|V||E|)$ time by Theorem 3.32. □

Edmonds' algorithm can be further sped up to obtain a running time of $|V|^3$, by warm-starting the sub procedure described by Algorithm 5. This can be achieved by using a so-called M-*alternating forest*, which is a structure to represent the parts of a graph that have already been explored in the search of an augmenting path. We do not give the details here of how to reduce the running time to $O(|V|^3)$. However, we nevertheless introduce the notion of M-alternating forest in the next section, not with the focus of speeding up the algorithm, but rather to prove Theorem 3.25 constructively.

3.4.3 Proof of Tutte-Berge formula

We now provide a proof of the Tutte-Berge formula, i.e., Theorem 3.25. Again, let $G = (V, E)$ be an undirected graph, M be a matching in G, and $X \subseteq V$ be the exposed vertices.

Definition 3.34: M-alternating forest

A subgraph $F = (V_F, E_F)$ of G is an M-alternating forest if it is a forest and satisfies the following three properties:

(i) $X \subseteq V_F$ and each connected component in F contains precisely one exposed vertex. This vertex is called the *root* of the connected component.

(ii) For any $v \in V_F$, the unique path $P_F(v)$ from the root of the connected component containing v to v is M-alternating.

(iii) All edges of F that are incident to a leaf of F are matching edges, except edges incident with the root.

A vertex $v \in V_F$ is called an *inner (outer) vertex* if its distance to the root in the connected component containing v is odd (even). In particular, all roots are outer vertices.

Assume we ran Edmonds' algorithm on the graph $G = (V, E)$ to obtain a maximum cardinality matching M. At the end of Edmonds' algorithm, we have a (blossom-)graph $G' = (V', E')$, where vertices in V' may correspond to contracted blossoms, and a contracted blossom may contain even several nested contracted sub-blossoms. Let $M' \subseteq M$ be the matching edges that are present in G', i.e., have not been contracted, and let X' be all exposed vertices in G'. We represent a vertex $v' \in V'$ simply by the set of all vertices of V contained in v', i.e., $v' \subseteq V$.

Because Edmonds' Algorithm stopped, there is no M'-alternating X'-X' walk in G' of positive length. Let us consider a maximal M'-alternating forest $F' = (V'_{F'}, E'_{F'})$ in G'. Such a forest can easily be constructed in $O(|E'|) = O(|E|)$ time by starting with the trivial alternating forest $(X', \emptyset)$ consisting of $|X'|$ single roots, and by growing this forest step by step. See Figure 3.9 for an example.

The following theorem shows that, given the forest F', we can readily deduce from it a set $S \subseteq V$ that allows for obtaining the equality shown in Theorem 3.25.

Theorem 3.35

Let $S \subseteq V$ be the subset of all vertices that correspond to inner vertices in F', i.e.,

$$S = \bigcup_{v' \text{ inner vertex in } F'} v' \ .$$

Then

$$|M| = \frac{1}{2}\left(|V| - q_G(S) + |S|\right) \ .$$

Clearly, the above theorem immediately implies Theorem 3.25. To show Theorem 3.35, we want to better understand the odd connected components of $G[V \setminus S]$. For this we first derive

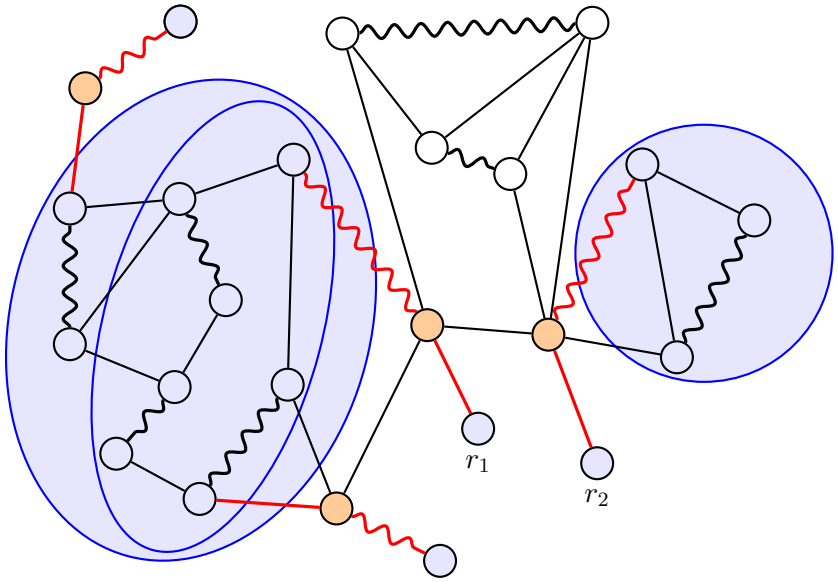


Figure 3.9: Example of a maximal forest F' (in red) constructed in an example graph at the end of Edmonds' algorithm, where blossoms (blue ellipses) are contracted. Blue vertices are outer vertices or belong to a shrunk blossom, which is also outer. Orange vertices are inner vertices. The forest F' has two connected components, one for each exposed vertex, which are denoted by r_1 and r_2, respectively, and these vertices form the roots of F'.

some basic properties of F'.

Lemma 3.36

The forest F' satisfies the following properties:

(i) $|v'| = 1$ for each inner vertex $v' \in V'$.

(ii) $|v'|$ is odd for any $v' \in V'$.

(iii) There is no edge in G' between any two outer vertices, or between any outer vertex and any vertex $v' \in V' \setminus V'_{F'}$.

Assuming Lemma 3.36, Theorem 3.35 is obtained as follows.

Proof of Theorem 3.35. Let $S \subseteq V$ be defined as in Theorem 3.35, i.e., these are all vertices in V contained in inner vertices of F'. By Lemma 3.36 (i), each inner vertex corresponds to a single vertex of V. Hence, $|S|$ is the number of inner vertices. Consider the graph $G[V \setminus S]$. Each outer vertex of F' corresponds to a connected component of $G[V \setminus S]$ by Lemma 3.36 (iii). Furthermore, by Lemma 3.36 (ii), each outer vertex of F' contains an odd number of vertices of V. Hence,

$$q_G(S) \geq \#\text{outer vertices in } F' \ .$$

Now observe that each connected component of F' has one more outer vertex than inner

vertices. One can see this by observing that in each component the matching edges M' connect always an outer and an inner vertex, and the root is the only non-matched vertex and is outer. Hence

$$q_G(S) - |S| \geq \#\text{roots in } F' = |X'| = |X| = |V| - 2|M| \ ,$$

which, after reordering terms, leads to

$$|M| \geq \frac{1}{2}\left(|V| - q_G(S) + |S|\right) \ .$$

Finally, Lemma 3.27 implies that the above inequality must hold with equality, as desired. □

Hence, it remains to prove Lemma 3.36. It turns out that the most difficult part of Lemma 3.36 is point (i). To prove this statement, we want to make sure that shrunk blossoms never become part of an inner vertex. This is clear at the moment a blossom is created. However, we have to show that also later shrinking operations do not destroy that property. The following Lemma is key in proving this property as it establishes an invariant that does not change through shrinking operations.

Lemma 3.37

Let $G = (V, E)$ be an undirected graph, M a matching in G, and let B be a blossom. Let $v \in V$ be any non-exposed vertex that is not part of the blossom. Then if there is an even M-alternating path in G from an exposed vertex to v, then there is an even M/B-alternating path in G/B from an exposed vertex to v.

Proof. Let $(v_0, \ldots, v_t)$ be an M-flower that corresponds to the blossom $B = (v_i, \ldots, v_t)$. Let $P = (u_0, \ldots, u_q)$ be an even M-alternating path in G from an exposed vertex u_0 to $u_q = v$. Notice that we may have $u_0 = v_0$.

Let $V_p \subseteq V$ and $E_p \subseteq E$ be the vertices and edges, respectively, on the path P, i.e.,

$$\begin{aligned} V_p &\coloneqq \{u_0, \ldots, u_q\} \ , \\ E_p &\coloneqq \{\{u_{j-1}, u_j\} \colon j \in [q]\} \ . \end{aligned}$$

Similarly, we denote by $V_f \subseteq V$ and $E_f \subseteq V$ the vertices and edges, respectively, of the flower $(v_0, \ldots, v_t)$, i.e.,

$$\begin{aligned} V_f &\coloneqq \{v_0, \ldots, v_t\} \ , \\ E_f &\coloneqq \{\{v_{j-1}, v_j\} \colon j \in [t]\} \ . \end{aligned}$$

We will derive the statement by applying Theorem 3.30 to an auxiliary graph H. To this end, let w be a new vertex, i.e., one not already contained in V. We consider two cases, depending on whether $v_0 = u_0$.

If $v_0 = u_0$, the we define the auxiliary graph to be

$$H \coloneqq (V_p \cup V_f \cup \{w\}, E_p \cup E_f \cup \{u_q, w\}) \ .$$

Otherwise, if $v_0 \neq u_0$, our auxiliary graph H is the one obtained from the above one, i.e., from $(V_p \cup V_f \cup \{w\}, E_p \cup E_f \cup \{u_q, w\})$, by contracting the vertices $\{v_0, u_0\}$. In both cases, we consider the matching $M_H := M \cap (E_p \cup E_f)$. Notice that H contains only two exposed vertices, which we denote by a and b. One is $b = w$ and the other one is $a = v_0 = u_0$ in the first case, and the contracted vertex $a = \{v_0, u_0\}$ in the case $v_0 \neq u_0$. Moreover, in both cases, H admits an M_H-augmenting a-b path, obtained by appending the vertex w to P. Hence, by Theorem 3.30, there is an M_H/B-augmenting path in H/B between the only two exposed vertices a and b. By removing the last edge of this path, we obtain an even M/B-alternating path in G/B from an exposed vertex to v, as desired. □

We are now ready to prove Lemma 3.36.

Proof of Lemma 3.36. (i) Consider a vertex $v' \in V'$ with $|v'| > 1$, and assume by sake of deriving a contradiction that v' is inner. Because $|v'| > 1$, vertex v' was created through the contraction of a blossom B at some step during Emonds' algorithm. Just after that step, there was an even M-alternating path from some exposed vertex to v', given by the edges in the stem of the flower that corresponded to B. By Lemma 3.37, this property is preserved when shrinking blossoms. Hence, there must be an even M'-alternating path from an exposed vertex to v' in G'. Because we assumed that v' is inner, there is an odd M'-alternating path Q in F' from some exposed vertex to v'. By concatenating Q with the reverse of P we obtain an odd M'-alternating X'-X' walk, i.e., a walk between two exposed vertices in G' (the two exposed endpoints of the walk do not have to be distinct). However, this is not possible as we assumed that G' is the graph obtained when Edmonds' algorithm stops, which does not happen if there still is an M'-alternating X'-X' walk.

(ii) If v' corresponds to a single vertex in V, then $|v'| = 1$ is odd. Otherwise the result follows from the fact that blossoms have an odd number of vertices. Hence, if before contracting a blossom B, all supernodes in B contain an odd number of vertices of V, then the contracted supernode B also contains an odd number of vertices of V. Therefore, the parity is preserved through contractions, implying that $|v'|$ must be odd.

(iii) Notice that no outer vertex u' can be connected to any vertex $v' \in V' \setminus V'_{F'}$ outside the forest, as this would violate maximality of F' because we could have added the edge $\{u', v'\}$ to F and the matching edge adjacent to v'. Note that such a matching edge must exist, for otherwise there would be an M'-augmenting path obtained by concatenating the path $P_F(u')$ from the root of the component containing u' to u' and the edge $\{u', v'\}$.

Now consider the case of an edge between two outer vertices $u', v' \in V'$. Again, let $P_F(u')$ be the M'-alternating path in F' starting at the root of the connected component of F' that contains u' and ending at u'. Analogously, let $P_F(v')$ be the M'-alternating path in F' starting at the root of the connected component of F' that contains v' and ending at v'. If there is an edge between u' and v', then the walk obtained by concatenating $P(u')$, $\{u', v'\}$, and the reverse of $P(v')$, is an M'-alternating X'-X' walk, which again contradicts the fact that Edmonds' algorithm stopped. □

4 A Brief Introduction to Linear Programming

4.1 Basic notions and examples

Linear programming captures one of the most canonical and influential constrained optimization problems. More precisely, it asks to maximize or minimize a linear objective under linear inequality and equality constraints. Below, we give a concrete example of a linear programming problem:

$$\begin{array}{rrcrcrcr} \max & 6x_1 & + & 5x_2 & + & 5.5x_3 & & \\ \text{subject to} & 10x_1 & - & x_2 & - & 2.5x_3 & \geq & 11.5 \\ & -21x_1 & + & x_2 & - & 6x_3 & \geq & -104 \\ & 4.25x_1 & + & 2.75x_2 & - & x_3 & \leq & 24 \\ & & & x_2 & & & \geq & 0 \\ & 10x_1 & - & x_2 & + & 35x_3 & \geq & 49 \\ & x_1 & & & + & 2x_3 & = & 12\ . \end{array}$$

Here, the variables x_1, x_2, x_3 all take real values. Such a problem is often called a *linear program* or simply *LP*. As highlighted in the above example, the objective depends linearly on the variables, and each constraint imposes either a lower bound, an upper bound, or an equality condition on a linear form of the variables. Finally, in a linear program one can either ask to maximize or minimize the objective. Hence, formally, a general linear program is of the following form:

$$\begin{array}{rrcl} \max / \min & c^\top x & & \\ & Ax & \leq & e \\ & Bx & \geq & f \\ & Cx & = & g\ , \end{array} \qquad \text{(general LP)}$$

where A, B, and C are real matrices, and c, e, f, and g are real column vectors of the appropriate dimensions.

Linear programs can be reformulated in various equivalent forms. When talking about structural results or algorithms, it is often convenient to fix one particular form of writing a linear program, like the *canonical form*, which looks as follows:

$$\begin{array}{rrcl} \max & c^\top x & & \\ & Ax & \leq & b \\ & x & \geq & 0\ , \end{array} \qquad \text{(LP in canonical form)}$$

where, if $n \in \mathbb{Z}_{\geq 0}$ is the number of variables and $m \in \mathbb{Z}_{\geq 0}$ is the number of constraints in the system $Ax \leq b$, we have $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, and the variable x take values in $\mathbb{R}^n$. Indeed, any linear program can be transformed into canonical form as follows:

(i) Any constraint of type $a^\top x \geq \beta$ can be rewritten equivalently as $(-a)^\top x \leq -\beta$.
(ii) Any constraint of type $a^\top x = \beta$ can be replaced by a pair of equivalent constraints $a^\top x \leq \beta$ and $(-a)^\top x \leq -\beta$.
(iii) Any variable x_i that is not bound by a non-negativity constraint $x_i \geq 0$ can be replaced as follows. Introduce two new variables x_i^+ and x_i^- with non-negativity constraints $x_i^+ \geq 0$ and $x_i^- \geq 0$, and replace all occurrences of x_i by $x_i^+ - x_i^-$.
(iv) If the problem is a minimization problem, then replace the objective $\min c^\top x$ by the objective $\max -c^\top x$. This will flip the sign of all solution values. However, maximizers of the new problem are minimizers of the original problem and vice versa.

Notice that linear programming can also be equivalently described as the task to maximize a linear function over a polyhedron, which is the intersection of finitely many half-spaces in finite-dimensional Euclidean space.

4.1.1 Different types of LPs and goal of LP algorithms

Linear programs are often divided into the following three types, depending on their optimal value:

LP with finite optimum An LP with a finite optimal value. The optimum is either attained at a unique point or the LP may have multiple optimal solutions.

Unbounded LP An LP with feasible solutions of arbitrarily large value in case of a maximization problem, or of arbitrarily small value in case of a minimization problem.

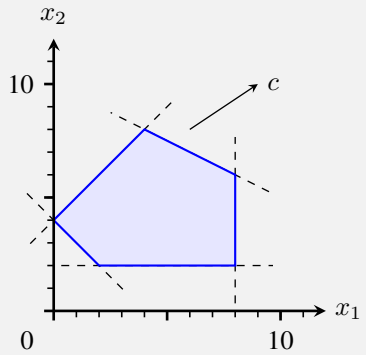
Infeasible LP An LP without any feasible solutions. If it is a maximization problem, its optimal value is often set to $-\infty$ by convention, and to $+\infty$ for minimization problems.

In the three examples below, we exemplify the above notions on LPs in dimension 2.

Example 4.1: LP with finite optimum

The LP below has a unique optimum, attained at the point $(x_1, x_2) = (8, 6)$.

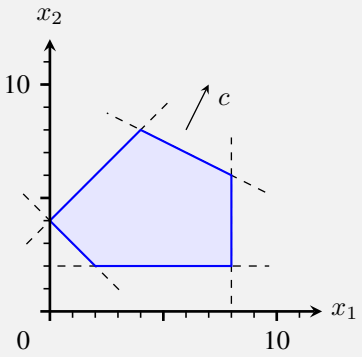
$$\begin{array}{lrcrcl} \max & 3x_1 & + & 2x_2 & & \\ & -x_1 & + & x_2 & \leq & 4 \\ & x_1 & + & 2x_2 & \leq & 20 \\ & x_1 & & & \leq & 8 \\ & & & x_2 & \geq & 2 \\ & x_1 & + & x_2 & \geq & 4 \end{array}$$



Taking the same example as above with a different objective function that is perpendicular to one of the sides of the polygon describing the feasible region, an LP with multiple optimal

solutions is obtained.

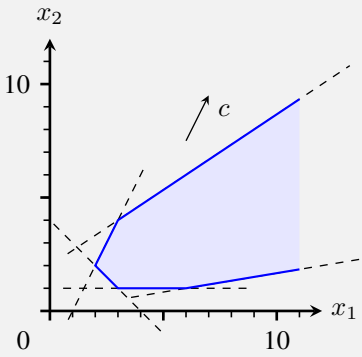
$$
\begin{array}{lrcrcl}
\max & x_1 & + & 2x_2 & & \\
 & -x_1 & + & x_2 & \leq & 4 \\
 & x_1 & + & 2x_2 & \leq & 20 \\
 & x_1 & & & \leq & 8 \\
 & & & x_2 & \geq & 2 \\
 & x_1 & + & x_2 & \geq & 4
\end{array}
$$



Example 4.2: Unbounded LP

The LP below is unbounded.

$$
\begin{array}{lrcrcl}
\max & x_1 & + & 2x_2 & & \\
 & x_1 & - & 6x_2 & \leq & 0 \\
 & & & x_2 & \geq & 1 \\
 & x_1 & + & x_2 & \geq & 4 \\
 & 2x_1 & - & x_2 & \geq & 2 \\
 & -2x_1 & + & 3x_2 & \leq & 6
\end{array}
$$

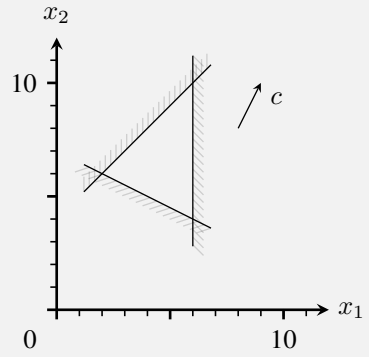


Indeed, one can observe that for any $\lambda \geq 0$, the point $(5 + 3\lambda, 2 + \lambda)$ is feasible, and its objective is $9 + 5\lambda$, which can get arbitrarily large for large enough values of λ.

Example 4.3: Infeasible LP

The LP below is an example of an infeasible LP.

$$\begin{array}{lrcrcr} \max & x_1 & + & 2x_2 & & \\ & -x_1 & + & x_2 & \geq & 4 \\ & x_1 & + & 2x_2 & \leq & 14 \\ & x_1 & & & \geq & 6 \end{array}$$



Indeed, assuming that there is a feasible point (x_1, x_2), we can use the constraints to obtain

$$0 = 2 \cdot \underbrace{(-x_1 + x_2)}_{\geq 4} - \underbrace{(x_1 + 2x_2)}_{\leq 14} + 3 \cdot \underbrace{x_1}_{\geq 6} \geq 10 \ ,$$

which is a contradiction.

When talking about an algorithm that solves linear programs, we expect the algorithm to provide answers to the following questions. First, the algorithm should detect the type of LP we are dealing with, i.e., whether it is an LP with finite optimum, an unbounded LP, or an infeasible one. Normally, in case of an LP with finite optimum, one does not require an LP algorithm to distinguish between whether there is a unique optimal solution or multiple ones. Second, the algorithm should return the following.

- If the LP has a finite optimum, the algorithm should return an optimal solution.
- If the LP is unbounded, the algorithm should return a half-line pointing in an improving unbounded direction. More precisely, assuming that we want to maximize the objective c, this consists of a feasible point $y \in \mathbb{R}^n$ and a non-zero vector $v \in \mathbb{R}^n$ such that
  (i) $y + \lambda v$ is feasible for any $\lambda \in \mathbb{R}_{\geq 0}$, and
  (ii) $c^\top v > 0$.

Hence, in the first problem of Example 4.1, an LP algorithm must return the unique optimal solution $\binom{8}{6}$, whereas in the second LP of the same example, any point on the segment connecting $\binom{8}{6}$ and $\binom{4}{8}$ can be returned. For the unbounded LP shown in Example 4.2, an LP algorithm needs to return a strictly improving half-line, for example $\binom{5}{2} + \lambda \cdot \binom{3}{1}$ for $\lambda \in \mathbb{R}_{\geq 0}$.

Whereas the above highlights minimal requirements that we expect LP algorithms to fulfill, sometimes, additional information is desired, as for example:

- In case of an LP with finite optimum such that there is a corner of the feasible region at which the optimum is attained, we may want that the LP algorithm returns an optimal solution that is a corner of the feasible region.
- In case of an LP with finite optimum, we may want a certificate of optimality.
- In case of an infeasible LP, we may want to obtain a certificate of infeasibiliy.

Informally speaking, a certificate of optimality or infeasibility is a small piece of information with which one can check quickly that a certain solution is optimal or a certain LP is infeasible,

respectively. Such certificates can be obtained through the theory of linear duality, which we discuss later on.

There are efficient algorithms to solve linear programs and also answer all of the above-highlighted questions.

4.1.2 Example applications

Before we talk about structural and algorithmic results related to linear programming, we present some of its numerous, and sometimes surprising, applications.

4.1.2.1 Production planning

Linear programs, and variations and generalizations thereof, are often used in Operations Research problems coming from industrial applications. In this example, we consider an extreme simplification of such a real-world problem, to provide a glimpse of why linear programs can be relevant in such contexts, and to introduce some terminology that is prevalent in Operations Research when dealing with linear programs.

To this end, consider the following heavily simplified production planning example focusing on a recycling facility that recycles a raw material into a purified final form, which we simply call the *recycled material*. We are interested in determining the maximum number of tons of recycled material that the facility can produce per day. There are two different types of raw materials, type A and type B, both being recycled to the same final recycled material. These raw materials have different types and levels of impurity.

At most 20 t of total raw material of both types together can be transported to the recycling facility per day. For every ton of raw material of type A, 720 kg of recycled material can be produced, whereas for every ton of raw material of type B, 600 kg of recycled material can be obtained. Moreover, if a mix of raw material A and B is used, then the total amount of recycled material (in kg) that can be produced behaves linearly with respect to the above values.

The recycling process can handle 16 t/d (tons per day) of raw material A, if only raw material A is used, and 24 t/d of raw material B, if only raw material B is used. We assume that the number of tons per day of a mix of raw materials A and B that can be processed behaves linearly with respect to the above values, e.g., the recycling process can simultaneously handle 8 t/d of raw material A and 12 t/d of raw material B.

The purity of the recycled material depends on the proportions of raw material A and B that are being used. This is measured though an *impurity coefficient* which should be no more than 30 for the recycled material. Only using raw material A will lead to an impurity coefficient of 18 for the recycled material, and the obtained impurity coefficient when only using B is 38. A mix of both raw materials leads to an impurity coefficient that behaves linearly with respect to the above values, e.g., when using the same amount of raw material A and B, the recycled material has impurity coefficient 28.

Finally, the recycling process leads to smoke emissions, and the recycling facility wants to adhere to strict emission limits by not emitting more than 12 kg of smoke per day. For every ton of raw material A that is used, 0.5 kg of smoke are emitted, and for every ton of raw material B, the smoke emission is 1 kg.

To determine the maximum number of tons of recycled material that the recycling facility can produced per day subject to the above conditions, we set up a linear program.

Decision variables The key parameters to be determined are the total number of tons per day that are used of raw material A and B, respectively. Because these are the parameters that can be controlled in this decision problem, the following two variables, which correspond to these "free" parameters, are called *decision variables*:

- $x_1 \in \mathbb{R}_{\geq 0}$: amount of raw material A used per day (in t/d), and
- $x_2 \in \mathbb{R}_{\geq 0}$: amount of raw material B used per day (in t/d).

Objective function The total number of tons of recycled material per day depends on the number of tons of raw material A and B that are recycled daily. Because each ton of raw material A and B lead to 0.72 t and 0.6 t of recycled material, respectively, and, as described, mixtures behave linearly, the objective is to maximize the following linear form:

$$\underbrace{0.72\, x_1 + 0.6\, x_2}_{\text{objective function}} \;=\; \underbrace{z}_{\substack{\text{value of the}\\ \text{objective function}}} \;.$$

Hence, in the two-dimensional (x_1, x_2)-space, the level curve for every fixed value of z is a line.

Constraints In addition to the non-negativity requirements, the variables are also constrained by further conditions that can be of various types, like physical, economic, or legal conditions.

Smoke emission constraints. The maximum smoke emission must be no more than 12 kg/d. We know that 0.5 kg of smoke are emitted per ton of raw material A and 1 kg of smoke per ton of raw material B. When x_1 tons of A and x_2 tons of B are being used per day, we have

$$0.5\, x_1 \;+\; 1\, x_2$$

kilograms of smoke emitted per day. The following constraint makes sure that this value does not exceed the emission limit of 12 kg/d:

$$\begin{array}{lcll} 0.5x_1 & + & 1x_2 & \leq \quad 12 \;. \\ \uparrow & & \uparrow & \uparrow \\ \text{coefficients of the "left-hand side (lhs)"} & & & \text{"right-hand side (rhs)"} \end{array}$$

Transport capacities. The transport capacity of 20 t of total raw material per day is captured by the following constraint:

$$x_1 + x_2 \leq 20 \;.$$

Recycling performance. The facility can recycle 16 t/d of raw material A and 24 t/d of raw material B. In other words, $1/16$ d is needed to recycle one ton of A and $1/24$ d to recycle one ton

of B. Because the processing of a mix of A and B behaves linearly, processing x_1 tons of raw material A and x_2 tons of raw material B takes

$$\frac{1}{16}x_1 + \frac{1}{24}x_2 \quad \text{days.}$$

Thus we get the following inequality for the recycling performance constraint:

$$\frac{1}{16}x_1 + \frac{1}{24}x_2 \leq 1 \ .$$

Expressing the maximum recycling performance in time units per ton, we were able to link them with the time constraints.

Purity constraint. We recall that the required impurity coefficient of the recycled material must not exceed 30 and depends on the proportions of the raw materials used. Raw material A alone leads to recycled material of impurity coefficient 18, whereas raw material B alone leads to an impurity coefficient of 38. The proportion of raw material A used among all the used raw material is $\frac{x_1}{x_1+x_2}$ and, accordingly, the proportion of raw material B is $\frac{x_2}{x_1+x_2}$. The final impurity coefficient is the weighted average of the impurity obtained when only using A or B, respectively, which leads to the following purity constraint:

$$18 \cdot \frac{x_1}{x_1 + x_2} + 38 \cdot \frac{x_2}{x_1 + x_2} \leq 30 \ .$$

Even though this constraint is not linear, it can be linearized by multiplying both sides by x_1+x_2. This leads to the linear constraint

$$12x_1 - 8x_2 \geq 0 \ ,$$

which ensures that the required purity level is achieved for the recycled material.

Mathematical formulation Summarizing the above discussion, we obtain the following linear program, which is a mathematical model of the described problem.

$$\begin{array}{rrrcrl}
\max & 0.72x_1 & + & 0.6x_2 & & \text{(amount of recycled material)} \\
& \frac{1}{2}x_1 & + & x_2 & \leq \ 12 & \text{(smoke emission)} \\
& x_1 & + & x_2 & \leq \ 20 & \text{(transport capacity)} \\
& \frac{1}{16}x_1 & + & \frac{1}{24}x_2 & \leq \ 1 & \text{(recycling performance)} \\
& 12x_1 & - & 8x_2 & \geq \ 0 & \text{(purity)} \\
& x_1 & & & \geq \ 0 & \text{(non-negativity)} \\
& & & x_2 & \geq \ 0 & \text{(non-negativity),}
\end{array}$$

or, in matrix notation in canonical form, $\max \ c^\top x$ subject to $Ax \leq b$ and $x \geq 0$, where

$$c = \begin{pmatrix} 24 \\ 20 \end{pmatrix}, \qquad A = \begin{pmatrix} \frac{1}{2} & 1 \\ 1 & 1 \\ \frac{1}{16} & \frac{1}{24} \\ -12 & 8 \end{pmatrix}, \qquad x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \quad \text{and} \quad b = \begin{pmatrix} 12 \\ 20 \\ 1 \\ 0 \end{pmatrix} \ .$$

Note that, to obtain canonical form, we multiplied the purity constraint by -1 to convert the inequality type from "$\geq$" into "$\leq$".

4.1.2.2 ℓ_1 regression

In statistics, regression analysis is used to estimate the relationships between different variables. In this section, we consider linear ℓ_1 regression. Suppose we have a set of input points $x_1, \dots, x_k \in \mathbb{R}^n$, and for each input point x_i we have a response variable $y_i \in \mathbb{R}$ that we observed. In linear regression, we consider a model where we assume that the response variables are an affine transformation of the input points plus some random error. This leads to the question of finding an affine function $f(x) \coloneqq a^\top x + \beta$, where $a \in \mathbb{R}^n$ and $\beta \in \mathbb{R}$ such that $f(x_1), \dots, f(x_k)$ is a good approximation of $y_1, \dots y_k$. To quantify how good the approximation is, ℓ_1 regression uses the ℓ_1 norm (contrary to the more common square error, which uses the ℓ_2 norm), leading to the following optimization problem

$$\min \left\{ \sum_{i=1}^{k} |y_i - a^\top x_i - \beta| \colon a \in \mathbb{R}^n, \beta \in \mathbb{R} \right\} \ . \tag{4.1}$$

Notice that because the response variables are scalars, the ℓ_1 norm of the error $y_i - a^\top x_i - \beta$ is simply its absolute value. More generally, we could have response variables in a higher-dimensional space $\mathbb{R}^m$; the discussion here extends to this generalization in a straightforward way.

Problem (4.1) is not a linear program in its current form because the absolute value is not an affine function. However, it can be converted into one by introducing, for $i \in [k]$, a variable z_i that captures the ℓ_1 error $|y_i - a^\top x_i - \beta|$:

$$\begin{aligned}
\min \quad & \sum_{i=1}^{k} z_i & & & & & & & \\
& y_i & - & a^\top x_i & - & \beta & \leq & z_i & \forall i \in [k] \\
& -y_i & + & a^\top x_i & + & \beta & \leq & z_i & \forall i \in [k] \\
& & & & & a & \in & \mathbb{R}^n & \\
& & & & & \beta & \in & \mathbb{R} & \\
& & & & & z & \in & \mathbb{R}^k \ . &
\end{aligned} \tag{4.2}$$

Notice that the constraints imposed on the variables z_i only require $z_i \geq |y_i - a^\top x_i - \beta|$ and not the ostensibly more natural $z_i = |y_i - a^\top x_i - \beta|$. The reason is that requiring $z_i = |y_i - a^\top x_i - \beta|$ would lead to a non-convex set of feasible solutions; however, linear programs always optimize over a polyhedron, which is convex. Nevertheless, because we minimize the sum of the z_i, the fact that the z_i will be at least as large as the absolute value is sufficient. They will automatically satisfy $z_i = |y_i - a^\top x_i - \beta|$ in an optimal solution to the LP, because setting them to a larger value would just lead to a strictly worse objective value. Hence, finding the optimal coefficients in linear ℓ_1 regression can be reduced to solving a linear program.

Figure 4.1 shows an example of linear ℓ_1 regression with unidimensional input points, i.e., $n = 1$, applied to a small random data set.

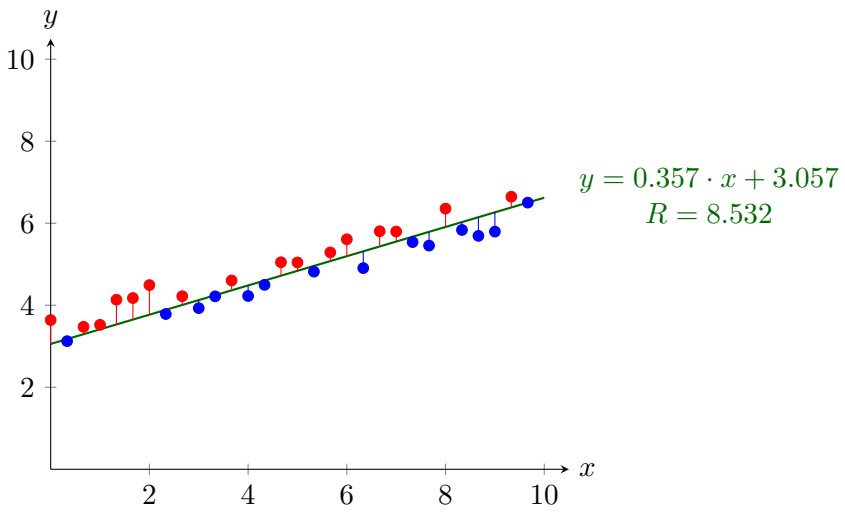


Figure 4.1: Example with 30 pairs $(x_i, y_i) \in \mathbb{R}^2$. The green line is the optimal linear ℓ_1 regression computed by solving the corresponding linear program (4.2). The sum of the vertical distances between the points and the line is the total error R that got minimized.

4.1.2.3 Resource allocation

Resource allocation is one of the classical problem settings in Operations Research. The goal is to distribute scarce resources among different activities, commonly with the objective of maximizing profit.

To exemplify this problem class, consider a fictitious Italian coffee shop that wants to edge the competition by brewing its own special coffee types. They are doing this by combining different sorts of coffee beans into a single type. Moreover, they know how many beans of which sort they need to produce one liter of a certain coffee type. Now they need to figure out how much to produce of each coffee type (resource allocation) to maximize profit. Assume that they have fixed amounts of coffee beans of each sort (scarce resource) to be used for the next brewing cycle. Moreover, the coffee types need very different compositions of beans and each type gives a different profit per liter. This is shown in Table 4.1, where we assume that there are four bean sorts (Liberica, Excelsa, Arabica, and Robusta), and three coffee types (Prego, Barzo, Pronto).

To approach this problem with linear programming, we first introduce a variable for each coffee type, capturing the amount of coffee to be brewed of the corresponding type:

$$c_j \coloneqq \text{production quantity of coffee type } j \text{ (in liters)} \qquad \forall j \in \{1, 2, 3\} \ ,$$

where $j = 1$ corresponds to coffee type Prego, $j = 2$ to Barzo, and $j = 3$ to Pronto.

Now we introduce one constraint per bean type to ensure that the total number of beans needed

Bean sort	Available amount	Prego	Barzo	Pronto
Liberica	300 000	100	100	300
Excelsa	350 000	200	300	100
Arabica	250 000	100	100	250
Robusta	500 000	300	200	200
profit/liter	-	10	10	20

Table 4.1: Beans needed per liter of each coffee type.

of each type is not exceeded by the production plan.

$$\begin{array}{lrcrcrcr} \text{Liberica:} & 100c_1 & + & 100c_2 & + & 300c_3 & \leq & 300\,000 \\ \text{Excelsa:} & 200c_1 & + & 300c_2 & + & 100c_3 & \leq & 350\,000 \\ \text{Arabica:} & 100c_1 & + & 100c_2 & + & 250c_3 & \leq & 250\,000 \\ \text{Robusta:} & 300c_1 & + & 200c_2 & + & 200c_3 & \leq & 500\,000 \end{array}$$

Finally, we have non-negativity constraints:

$$c_j \geq 0 \quad \forall j \in \{1, 2, 3\} \ .$$

The objective is to maximize the profit:

$$\max \ 10c_1 + 10c_2 + 20c_3 \ .$$

Hence, this completes the description of the problem in form of a linear program. Notice that to keep things as simple as possible, we assumed that unused beans do not have any value. However, we could also assume that there is a salvage value per bean type for unused beans, and this could still be modeled as a linear program.

4.1.2.4 Project scheduling

Consider a project comprising of a finite set of n tasks, where task i needs t_i time for completion. Additionally, some tasks depend on others and cannot be started before all the prerequisite tasks are finished. Our goal is to minimize the makespan, i.e., the total time required to finish all tasks and, thus, the whole project. Here, we assume that any number of tasks can be processed simultaneously without compromising performance or quality.

Figure 4.2 shows an example project with 9 tasks and the dependencies between the tasks. For example, the arrow from task 3 to task 4 indicates that task 4 can only be started once task 3 has been completed.

We now show how this project scheduling question can be modeled by a linear program. Assuming that work on the projects starts at time 0, denote by s_v the starting time of task v. Moreover, let z be a variable we introduce to denote the completion time of the whole project,

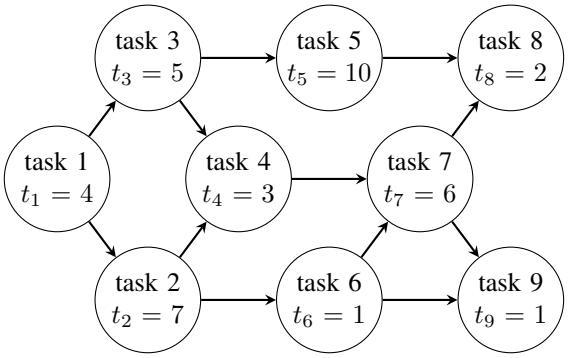


Figure 4.2: Example project scheduling problem.

which is the completion time of the last task. We set up the following linear program.

$$\begin{array}{lrcrcll} \min & z & & & & & \\ & z & \geq & s_v & + & t_v & \text{for every task } v \\ & s_v & \geq & s_u & + & t_u & \text{for every dependency } u \text{ of } v \\ & s_v & \geq & 0 & & & \text{for every task } v \\ & z & \geq & 0 & & & \end{array}$$

Let us check that solving (4.1.2.4) indeed yields an optimal task schedule (minimizing the makespan). On the one hand, the optimal solution of the LP gives us starting times for a scheduling of the tasks satisfying the precedence constraints, so the optimal value is greater than or equal to the minimal makespan. On the other hand, starting times of every optimal schedule satisfy the constraints of the LP, hence they are feasible solution, and thus the optimal objective value is less than or equal to the minimal makespan. Therefore, the optimal solution of the LP is precisely the minimal makespan of the whole project. Note that by backtracking from the end of the project and following tight constraints, one can construct the "bottleneck" chain of tasks that determine the project completion time (see Figure 4.3). This reasoning leads to the following interesting property: The minimum makespan is the same as the length of a longest path in the task graph, as shown in Figure 4.2, where the length of a path is the sum of the processing times of the tasks on the path.

4.1.2.5 Shortest s-t path

The problem of finding a shortest path from one site to another frequently occurs in various contexts, such as planning the quickest route from one physical location to another (from point A to point B), finding the shortest chain of social connections between two people to test the six degrees of separation hypothesis, or solving the Rubik's Cube in the least number of moves. In this section, we show that the shortest path problem can be reformulated in terms of an LP.

We recall the problem setting of the shortest path problem. We are given a directed graph $G = (V, A)$ with non-negative arc length $\ell\colon A \to \mathbb{R}_{\geq 0}$, together with a designated start node s and end node t (see Figure 4.4 for an example). The goal is to find a shortest s-t path.

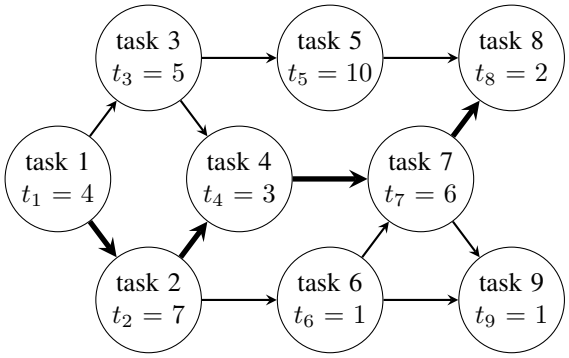


Figure 4.3: The highlighted path shows a bottleneck chain of tasks. This is a longest path in the above graph, where the length of the path is measured by summing up the processing times of the tasks on the path. In the shown example, the highlighted path has total length 22, which is the optimal makespan.

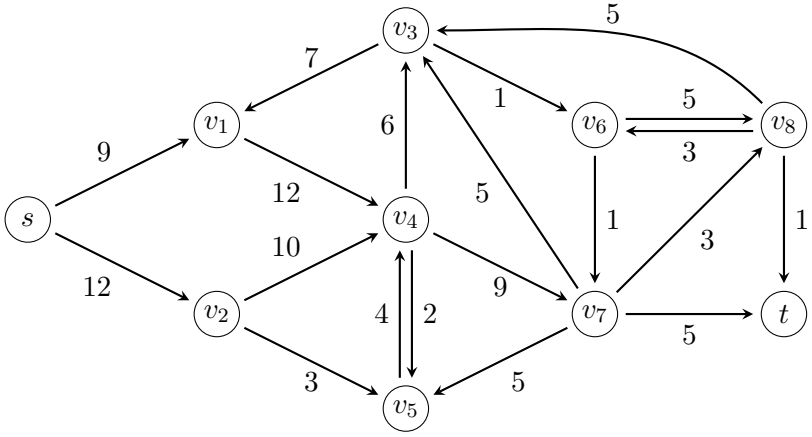


Figure 4.4: Example of finding a shortest s-t path.

To reformulate the shortest s-t path problem as an LP, we introduce a non-negative variable d_v for each node v with which we want to capture the distance from s to v. In particular, the distance from s to itself is zero, i.e., we can set $d_s = 0$. For the shown problem, another example distance is the distance from s to v_5, which is 15, because the shortest path from s to v_5 is $s \to v_2 \to v_5$ of length 15. Notice that the following is a valid constraint for the distances: for any arc a from u to v, the distance from s to v is no greater than the distance from s to u plus the length of the arc a. Indeed, to reach u from s, one option is to first visit v and then go from v to u over the arc a. Due to this, we will impose the constraints

$$d_v \leq d_u + \ell\,(a) \quad \text{for every arc } a \text{ from } u \text{ to } v,$$

on the variables d_v. In turns out that these constraints ensure that any d_v is never strictly larger than the distance from s to v. However, the discussed constraints allow for choosing distances that are significantly shorter. For example, $d_v = 0$ for every node v is a feasible solution. For this

reason, our linear program will actually not minimize d_t, but maximize it, which might sound counter-intuitive at first sight. In summary, the LP below is the formulation we are using.

$$\begin{array}{lllll} \max & d_t & & & \\ & d_s & = & 0 & \\ & d_v & \leq & d_u + \ell(a) & \text{for every arc } a \text{ from } u \text{ to } v \\ & d_v & \geq & 0 & \text{for every node } v \end{array}$$

Let us justify why the optimal value of the above LP is indeed the shortest path distance from s to t. First, for any path P, we can sum up all the constraints corresponding to arcs on the path to obtain the inequality $d_t \leq d_s + \ell(P) = \ell(P)$. Thus the maximum value of d_t we obtain through the LP is not greater than the shortest path length. Moreover, by setting d_v, for each node v, to the actual distances from s to v, all of our constraints are fulfilled. Hence, d_t is not smaller than the shortest path length from s to t, implying that the optimal LP value is indeed the distance from s to t.

Additionally, a shortest s-t path can be inferred from an optimal solution d of the LP. More precisely, one can observe that an s-t path is shortest if and only if it only uses arcs whose respective constraint in the LP is tight for d. Such a path can be obtained through BFS on the subgraph only consisting of arcs corresponding to d-tight LP constraints.

Figure 4.5 indicates the values d_v of an optimal LP solution for the example graph from Figure 4.4 in green next to each node v. Notice there are further optimal solutions; for example one can exchange $d_{v_1} = 9$ by $d_{v_1} = 8$. In particular, this shows that there may be vertices $v \in V \setminus \{s, t\}$ for which, even in an optimal LP solution, d_v is strictly smaller than the s-v distance.

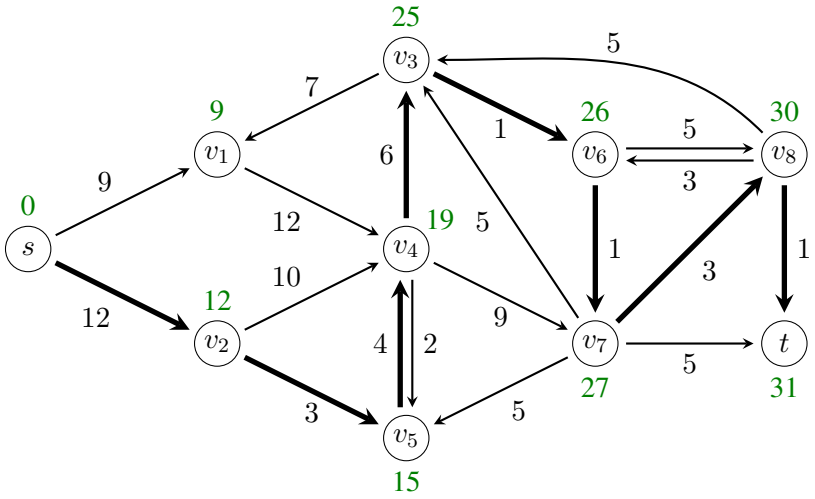


Figure 4.5: The green numbers show the d_v-values of one optimal LP solution. A shortest s-t path can be found via BFS by computing an s-t path only using arcs whose corresponding LP constraints are tight.

4.2 Polyhedra and basic convex geometry

As discussed, linear programming is about maximizing or minimizing a linear (or affine) function over a polyhedron. Not surprisingly, the study of polyhedra is key in understanding and solving linear programs. Moreover, as we will see later, polyhedra also play a central role in Combinatorial Optimization.

4.2.1 Basic notions

We start with some basic notions and terminology.

Definition 4.4: Half-space & hyperplane

A *half-space* in $\mathbb{R}^n$ is a set of the form $\{x \in \mathbb{R}^n : a^\top x \leq \beta\}$ for $a \in \mathbb{R}^n \setminus \{0\}$ and $\beta \in \mathbb{R}$. Moreover, $\{x \in \mathbb{R}^n : a^\top x = \beta\}$ is called a *hyperplane*.

Definition 4.5: Polyhedron & polytope

A *polyhedron* $P \subseteq \mathbb{R}^n$ is a finite intersection of half-spaces. Moreover, a bounded polyhedron is called a *polytope*.

An immediate consequence is that a polyhedron can be described by a finite set of linear inequalities, which is generally known as an *inequality description*. In other words, for a polyhedron $P \subseteq \mathbb{R}^n$, we can always write $P = \{x \in \mathbb{R}^n : Ax \leq b\}$ for some $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$ with $m \in \mathbb{Z}_{\geq 0}$. Even though, as we already discussed in the context of linear programs, we can always describe P in terms of less-than-or-equal inequalities, we allow an inequality description of a polyhedron to have inequalities of both types and also equality constraints. Clearly, there are many equivalent inequality descriptions for the same polyhedron.

Notice that polyhedra are convex sets.[1] This follows immediately from the fact that half-spaces are convex and the intersection of any family of convex sets is convex.

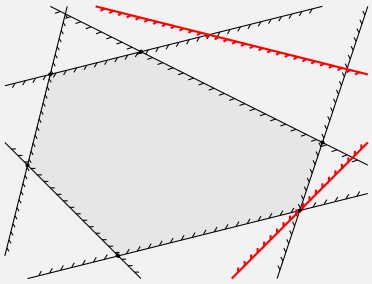
Definition 4.6: Redundancy

A linear inequality or equality of an inequality description of a polyhedron is called *redundant* if removing it from the description does not change the polyhedron.

Example 4.7: Redundant constraints

The picture below shows a polytope with two redundant constraints, highlighted in red.

[1] We recall that a set $X \subseteq \mathbb{R}^n$ is convex if, for any $x_1, x_2 \in X$ and $\lambda \in [0, 1]$, we have $\lambda x_1 + (1 - \lambda)x_2 \in X$.



Definition 4.8: Dimension of a polyhedron

The dimension $\dim(P)$ of a polyhedron $P \subseteq \mathbb{R}^n$ is the dimension of a smallest-dimensional affine subspace containing P, i.e.,

$$\dim(P) := \min\{k \in \mathbb{Z}_{\geq 0} \colon \exists A \in \mathbb{R}^{n\times n} \text{ with } \operatorname{rank}(A) = n - k \;\&\; Ax = Ay \;\forall x, y \in P\} \ .$$

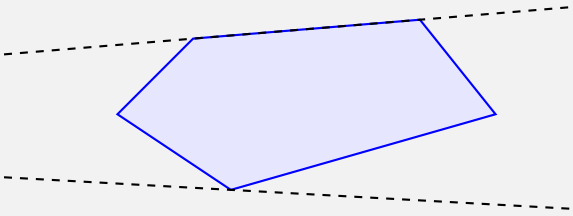
In particular, P is called *full-dimensional* if $\dim(P) = n$.

Definition 4.9: Supporting hyperplane

Let $P \subseteq \mathbb{R}^n$ be a polyhedron. A hyperplane $H = \{x \in \mathbb{R}^n \colon a^\top x = \beta\}$ is called *P-supporting*—or simply *supporting*, if P is clear from context—if $P \cap H \neq \emptyset$ and P is contained in one of the two half-spaces defined by H, i.e., either $P \subseteq \{x \in \mathbb{R}^n \colon a^\top x \leq \beta\}$ or $P \subseteq \{x \in \mathbb{R}^n \colon a^\top x \geq \beta\}$.

Example 4.10

The figure below shows a 2-dimensional polytope with two supporting hyperplanes.



Definition 4.11: Face, vertex, edge, and facet

Let $P \subseteq \mathbb{R}^n$ be a non-empty polyhedron.

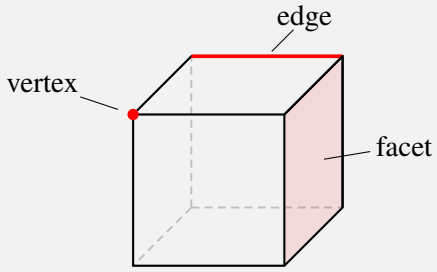
(i) A *face* of P is either P itself or the intersection of P with a supporting hyperplane.
(ii) A *vertex* of P is a 0-dimensional face of P.
(iii) An *edge* of P is a 1-dimensional face of P.
(iv) A *facet* of P is a $(\dim(P) - 1)$-dimensional face of P.

The empty polyhedron has only one face, which is the empty set. We denote by $\text{vertices}(P)$ the set of all vertices of P.

Notice that a face of a polyhedron P is also a polyhedron, because it is the intersection of P with a hyperplane; indeed, this follows because a hyperplane is the intersection of two half-spaces and a polyhedron is by definition a finite intersection of half-spaces. Thus, when talking about the dimension of a face, we refer to the notion of dimension that we introduced for polyhedra.

Example 4.12

The cube below is a polytope, and three of its faces are highlighted in red: a vertex, an edge, and a facet.



Note that vertices are the smallest (inclusion-wise and in terms of dimension) possible faces of a non-empty polyhedron P, and edges are the next-smallest faces P can have. On the other end, P is its largest face, and facets are the next-largest faces of P. A non-empty polyhedron does not need to have any vertices or edges; for example, a hyperplane in $\mathbb{R}^n$ for $n \geq 3$ is a polyhedron without vertices or edges. Moreover, the empty polyhedron, the whole space $\mathbb{R}^n$, as well as polytopes consisting of a single point, do not have any facets. However, all other polyhedra do have facets.

Proposition 4.13

Let $P = \{x \in \mathbb{R}^n : Ax \leq b\} \subseteq \mathbb{R}^n$ be a non-empty polyhedron with $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$, and let $F \subseteq P$. Then the following statements are equivalent.

(i) F is a face of P.
(ii) $\exists c \in \mathbb{R}^n$ such that $\delta := \max\{c^\top x : x \in P\}$ is finite and $F = \{x \in P : c^\top x = \delta\}$.
(iii) $F = \{x \in P : \overline{A}x = \overline{b}\} \neq \emptyset$ for a subsystem $\overline{A}x \leq \overline{b}$ of $Ax \leq b$.

Proposition 4.13 reveals a basic property of faces, namely that the face relationship is transitive.

Corollary 4.14

Let P be a polyhedron. Then a face of a face of P is itself a face of P.

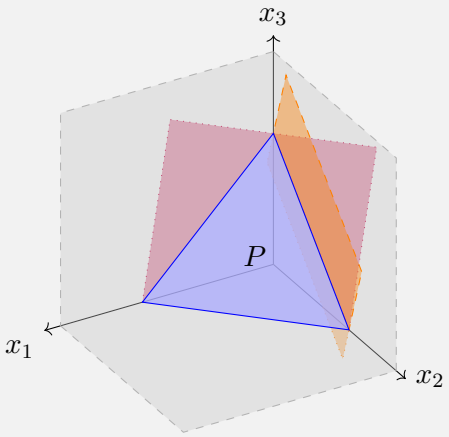
Proof. This follows immediately from point (iii) of Proposition 4.13. □

Definition 4.15: Facet-defining inequality

Let $P = \{x \in \mathbb{R}^n : Ax \leq b\}$ be a polyhedron. An inequality $a^\top x \leq \beta$ of the system $Ax \leq b$ is *facet-defining* if $F := P \cap \{x \in \mathbb{R}^n : a^\top x = \beta\}$ is a facet of P. We also say that $a^\top x \leq \beta$ is a constraint *defining* the facet F.

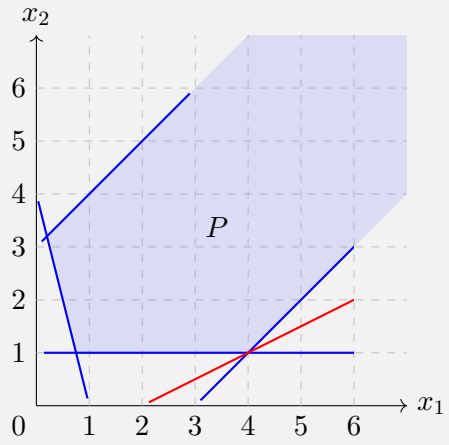
Example 4.16: Facet-defining inequalities

The example below shows a polytope $P \subseteq \mathbb{R}^3$ with $\dim(P) = 2$. The two red constraints imply an equality constraint and are responsible for P not being full-dimensional. Notice that neither of these two constraints is facet-defining even though they are both not redundant. Moreover, the orange constraint and the non-negativity constraint $x_1 \geq 0$ (highlighted in gray) both define the same facet of P.



$$P = \left\{ \begin{array}{rcrcrcl} x_1 & + & x_2 & + & x_3 & \leq & 2 \\ x_1 & + & x_2 & + & x_3 & \geq & 2 \\ x_1 & & & & & \geq & 0 \\ & & x_2 & & & \geq & 0 \\ & & & & x_3 & \geq & 0 \\ -4x_1 & + & x_2 & + & x_3 & \leq & 2 \end{array} \right\}$$

In the 2-dimensional example shown below, all constraints except for the red one are facet-defining. Whenever P is full-dimensional, every non-facet-defining constraint is redundant. However, as highlighted by the example above, this does not hold for lower-dimensional polyhedra.



$$P = \left\{ \begin{array}{rcrcr} x_1 & - & x_2 & \geq & -3 \\ 4x_1 & + & x_2 & \geq & 4 \\ & & x_2 & \geq & 1 \\ x_1 & - & 2x_2 & \leq & 2 \\ x_1 & - & x_2 & \leq & 3 \end{array} \right\}$$

Proposition 4.17

Let $P = \{x \in \mathbb{R}^n \colon Ax \leq b\}$ be a polyhedron. For any facet $F \subseteq P$ of P, there is at least one inequality $a^\top x \leq \beta$ of $Ax \leq b$ that defines F.

Lemma 4.18

Let $P = \{x \in \mathbb{R}^n \colon Ax \leq b\}$ be a full-dimensional polyhedron, then each inequality $a^\top x \leq \beta$ of $Ax \leq b$ that is not facet-defining for P is redundant.

Corollary 4.19

Let $P \subseteq \mathbb{R}^n$ be a polyhedron, and let f be the number of facets of P. Then, every inequality description of P requires at least f inequalities. Moreover, if P is full-dimensional, then an inequality description of P with f inequalities exists.

Definition 4.20: Extreme point

Let P be a polyhedron. A point $y \in P$ is an *extreme point* of P if it is not the midpoint of two distinct points of P, i.e., there are no $p_1, p_2 \in P$ with $p_1 \neq p_2$ and $y = \frac{1}{2}(p_1 + p_2)$.

Note that one can equivalently define an extreme point $y \in P$ of a polyhedron P by requiring that y is not a *non-trivial convex combination* of two distinct points $p_1, p_2 \in P$, i.e., we cannot write $y = \lambda p_1 + (1 - \lambda)p_2$ with $\lambda \in (0, 1)$. This follows by observing that if a point in a

convex set C is a non-trivial convex combination of two distinct points in C, then it is also the midpoint of two distinct points in C. In short, this modified definition as well as the one stated in Definition 4.20 state that for y to be an extreme point in P, it should not be in the interior of a segment that lies in P.

Proposition 4.21

Let $P = \{x \in \mathbb{R}^n : Ax \leq b\}$ be a polyhedron and $y \in P$. Then the following statements are equivalent.

(i) y is a vertex of P.

(ii) y is the unique solution to $\overline{A}x = \overline{b}$, where $\overline{A}x \leq \overline{b}$ is the subsystem of $Ax \leq b$ containing all constraints that are tight at y (often simply referred to as the *y-tight constraints*).

(iii) y is an extreme point of P.

Even though we focus on polyhedra here, we want to highlight that some natural results like Proposition 4.21 may not hold anymore when extending the involved notions to closed convex sets. For example, when defining extreme points and vertices for closed convex sets analogously to the polyhedral case, then an extreme point of a closed convex set does not need to be a vertex of it. See Figure 4.6 for an illustration.

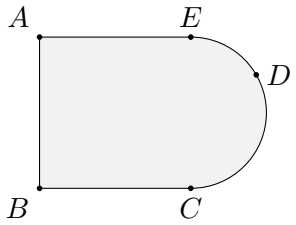


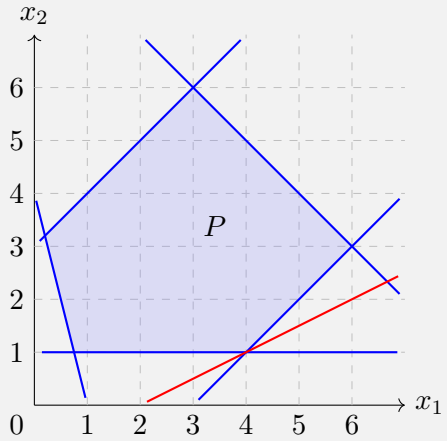
Figure 4.6: A closed and bounded convex set with an infinite number of extreme points and vertices, due to the curved boundary on the right of the object, which forms a half-circle. All of the five named points are extreme points. However, among these points, only A, B, and D are vertices. The points C and E are extreme points but not vertices because there is no supporting hyperplane that touches the set only at C or E, respectively.

Definition 4.22: Degeneracy with respect to linear inequality descriptions

Consider a linear inequality description of a polyhedron $P \subseteq \mathbb{R}^n$. A vertex $y \in \text{vertices}(P)$ is called *degenerate* (w.r.t. the linear inequality description) if the number of y-tight constraints in the linear inequality description is strictly larger than n. The inequality description is called *degenerate* if there is a degenerate vertex with respect to it.

Example 4.23: Degenerate vertex & degenerate inequality description

The inequality description of the 2-dimensional polytope below is degenerate, because the vertex $\binom{4}{1}$ is degenerate with respect to the given inequality description.



$$P = \left\{ \begin{array}{rcrcr} x_1 & - & x_2 & \geq & -3 \\ 4x_1 & + & x_2 & \geq & 4 \\ & & x_2 & \geq & 1 \\ x_1 & - & 2x_2 & \leq & 2 \\ x_1 & - & x_2 & \leq & 3 \\ x_1 & + & x_2 & \leq & 8 \end{array} \right\}$$

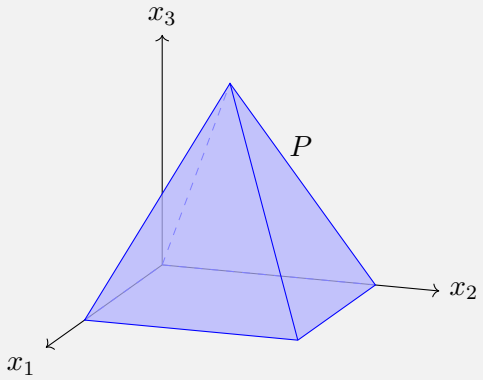
Definition 4.24: Degeneracy of a polyhedron

A polyhedron $P \subseteq \mathbb{R}^n$ is called *degenerate* if it has a vertex $y \in P$ contained in strictly more than $\dim(P)$ many facets.

Notice that even though the inequality description of the polytope in Example 4.23 is degenerate, the polytope itself is not degenerate.

Example 4.25: Degenerate polyhedron

The polytope below is an example of a degenerate 3-dimensional polytope, namely a pyramid with a square base. It is degenerate because the apex of the pyramid is a vertex contained in 4 facets.



$$P = \left\{ \begin{array}{rcrcr} & & & x_3 & \geq & 0 \\ 2x_1 & & - & x_3 & \geq & 0 \\ 2x_1 & & + & x_3 & \leq & 2 \\ & 2x_2 & - & x_3 & \geq & 0 \\ & 2x_2 & + & x_3 & \leq & 2 \end{array} \right\}$$

In particular, if a polyhedron $P \subseteq \mathbb{R}^n$ is degenerate, then any inequality description for it is

degenerate, too. This is a consequence of Proposition 4.17, which implies that for each facet, there must be a facet-defining inequality.

Definition 4.26: Dominant

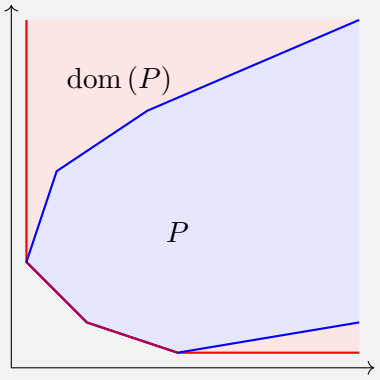
For a set $X \subseteq \mathbb{R}^n$, its dominant $\mathrm{dom}(X) \subseteq \mathbb{R}^n$ is defined by

$$\mathrm{dom}(X) := X + \mathbb{R}^n_{\geq 0} = \{x + y \colon x \in X, y \in \mathbb{R}^n_{\geq 0}\} \ .$$

For $X, Y \subseteq \mathbb{R}^n$, the set addition $X + Y := \{x + y \colon x \in X, y \in Y\}$, which we used above in the definition of the dominant, is called the *Minkowski sum*.

Example 4.27: Dominant

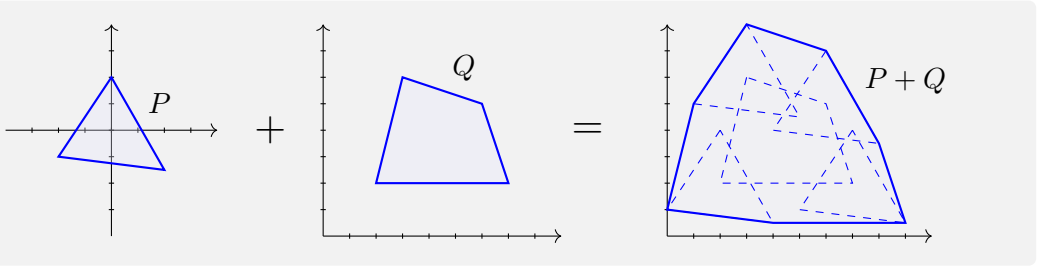
The figure below shows an unbounded polyhedron $P \subseteq \mathbb{R}^2$ (in blue) together with its corresponding dominant $\mathrm{dom}(P)$ (in red).



Before moving over to the representation of polyhedra, we show an example of a Minkowski sum, because this simple yet useful operation will be relevant also later on. Hence, it is helpful to develop some intuition for it.

Example 4.28: Minowski sum

The figure below shows the Minkowski sum $P + Q$ of two 2-dimensional polytopes P and Q. The third picture illustrates that $P + Q$ can be obtained by "shifting" one of the polytopes, say P, to each vertex of the other one—such that the origin in the figure showing P lies at the vertices of Q—and then taking the convex hull of the shifted versions of P. In particular, each vertex of $P + Q$ is the sum of a vertex of P and one of Q. However, not all sums of vertices in P and Q lead to a vertex of $P + Q$.



4.2.2 Representation of polyhedra

As mentioned, polyhedra are a basic family of convex sets. Not surprisingly, convexity is exploited in a multitude of results and algorithms linked to polyhedra. At the heart of convexity, there is the notion of a convex combination.

Definition 4.29: Convex combination

A *convex combination* of $x_1, \dots, x_k \in \mathbb{R}^n$ is a point described by $\sum_{i=1}^k \lambda_i x_i$, where $\lambda \in \mathbb{R}^k_{\geq 0}$ and $\sum_{i=1}^k \lambda_i = 1$.

In this course, convex combinations are always with respect to a finite number of elements. The fact that linear combinations with the above properties are called convex combinations is motivated by the observation that a set $X \subseteq \mathbb{R}^n$ is convex if and only if any convex combination of finitely many points in X lies in X. The convex combinations of an arbitrary point set are known as its convex hull.

Definition 4.30: Convex hull

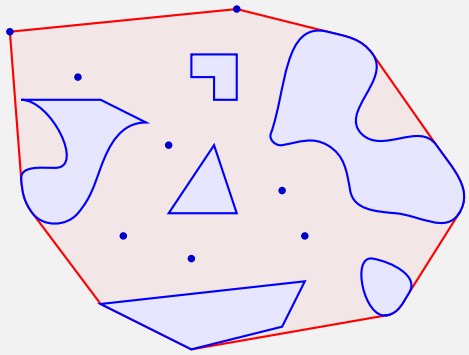
Let $X \subseteq \mathbb{R}^n$. The *convex hull* $\operatorname{conv}(X) \subseteq \mathbb{R}^n$ of X are all convex combinations of finitely many points in X, i.e.,

$$\operatorname{conv}(X) := \left\{ \sum_{i=1}^k \lambda_i x_i \;\middle|\; \begin{array}{l} k \in \mathbb{Z}_{\geq 1},\ x_1, \dots, x_k \in X, \text{ and} \\ \lambda_1, \dots, \lambda_k \in \mathbb{R}_{\geq 0} \text{ with } \sum_{i=1}^k \lambda_i = 1 \end{array} \right\}.$$

Hence, a set $X \subseteq \mathbb{R}^n$ is convex if and only if $\operatorname{conv}(X) = X$. This, together with the fact that the convex hull operator is monotone, i.e., $\operatorname{conv}(X) \subseteq \operatorname{conv}(Y)$ for any $X \subseteq Y \subseteq \mathbb{R}^n$, implies that $\operatorname{conv}(X)$ is the smallest convex set containing X.

Example 4.31: Convex hull

Example of the convex hull in 2 dimensions. The red area is the convex hull of the blue objects.



The following proposition highlights how polytopes are described by their vertices.

Proposition 4.32

A polytope is the convex hull of its vertices.

Moreover, as stated below, the convex hull of any finite point set is a polytope.

Proposition 4.33

Let $X \subseteq \mathbb{R}^n$ be a finite set. Then $\text{conv}(X)$ is a polytope.

To move from polytopes to polyhedra, we have to consider unbounded sets. One of the most basic building blocks when dealing with unbounded sets are cones, which we define below. They allow for obtaining a nice characterization of polyhedra that we mention later.

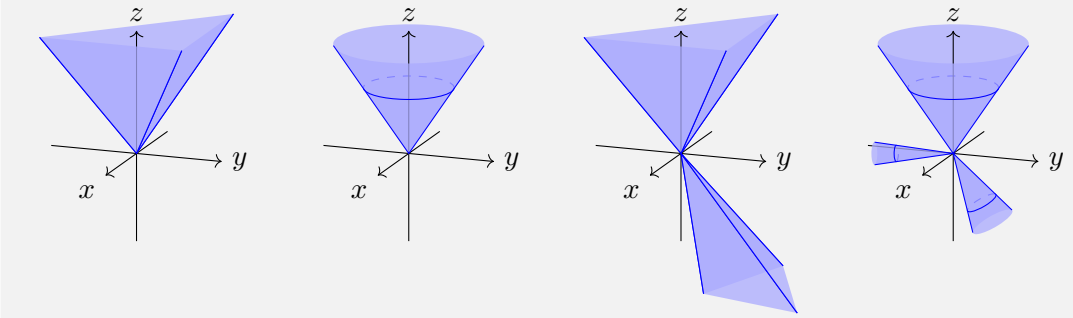
Definition 4.34: (Polyhedral) cone

A *cone* is a set $C \subseteq \mathbb{R}^n$ such that for any $x \in C$ and $\lambda \in \mathbb{R}_{\geq 0}$ we have $\lambda \cdot x \in C$. A cone that is a polyhedron is called a *polyhedral cone*.

We highlight that a cone does not need to be convex. Nevertheless, we will mostly be interested in convex cones. These can be characterized as the sets that are invariant under *conic combinations*, which are linear combinations with exclusively non-negative coefficients.

Example 4.35

Below are four examples of cones in $\mathbb{R}^3$. The first two of them are convex cones whereas the last two are not. Moreover, only the first one is a polyhedral cone. Notice that the third is not a polyhedral cone because it is not a polyhedron as it is not even convex.



Proposition 4.36

If $C \subseteq \mathbb{R}^n$ is a non-empty polyhedral cone, then

$$C = \{x \in \mathbb{R}^n \colon Ax \leq 0\} \tag{4.3}$$

for some matrix $A \in \mathbb{R}^{m \times n}$, where $m \in \mathbb{Z}_{\geq 0}$. Vice-versa, any set C with a description as in (4.3) is a polyhedral cone.

Another way to characterize polyhedral cones is by the fact that they have a finite set of generators, as formalized in the statement below.

Proposition 4.37

If $C \subseteq \mathbb{R}^n$ is a polyhedral cone, then there exists a finite set of points $x_1, \ldots, x_k \in \mathbb{R}^n$ such that

$$C = \left\{ \sum_{i=1}^{k} \lambda_i x_i \colon \lambda_i \geq 0 \; \forall i \in [k] \right\} \; . \tag{4.4}$$

The points $x_1, \ldots, x_k$ are called a *set of generators* of C. Vice-versa, any set C as described in (4.4) is a polyhedral cone.

Polyhedral cones can be thought of as the most basic unbounded polyhedra. The statement below underlines this intuition by showing that any polyhedron can be written as a Minkowski sum of a polytope and a polyhedral cone.

Proposition 4.38

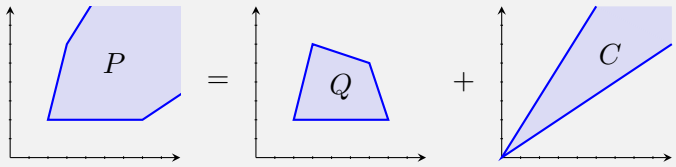
Let $P \subseteq \mathbb{R}^n$ be a polyhedron. Then

$$P = Q + C \; ,$$

where $Q \subseteq \mathbb{R}^n$ is a polytope and $C \subseteq \mathbb{R}^n$ is a polyhedral cone. Vice-versa, the Minkowski sum of a polytope and a polyhedral cone is always a polyhedron.

Example 4.39

The figure below shows an unbounded 2-dimensional polyhedron P and how it can be written as the Minkowski sum of a polytope Q and a polyhedral cone C.



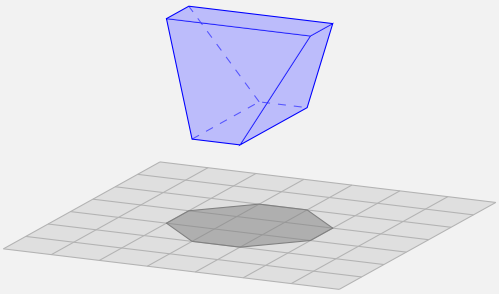
The above characterizations of polytopes and polyhedra often allow for elegantly deriving further important statements. One such statement, shown below, states that the family of polyhedra is invariant under affine transformations. This property is at the heart of various techniques in Combinatorial Optimization, in particular when a complex polytope, say one with a very large number of facets, is described as the affine projection of a much simpler one.

Proposition 4.40

An affine image of a polyhedron is a polyhedron, i.e., for any polyhedron $P \subseteq \mathbb{R}^n$ and any affine function $\varphi\colon \mathbb{R}^n \to \mathbb{R}^m$, the set $\varphi(P) := \{\varphi(x)\colon x \in P\}$ is a polyhedron.

Example 4.41: Projection of polytope

The illustration below shows an axis-parallel projection of a 3-dimensional polytope leading to a 2-dimensional octagon. First, this exemplifies that the projection of a polytope is a polytope.



Moreover, we highlight that the number of facets of the projection, which is 8, is strictly larger than the number of facets of the original polytope, which is only 6. The fact that polyhedra with many facets can sometimes be represented as projections of polyhedra with significantly fewer facets is a crucial property that can sometimes be exploited to obtain compact representations of polyhedra with exponentially many facets in terms of the dimension. More precisely, there are polytopes $P \subseteq \mathbb{R}^n$ whose number of facets is $2^{\Omega(n)}$, which can be obtained as a projection of a polytope $Q \subseteq \mathbb{R}^m$ for some $m \geq n$, where the number of facets of Q is bounded by a polynomial in n.

Moving to a higher-dimensional space and then projecting is crucial to obtain a compact representation. Indeed, due to Corollary 4.19, any inequality description of a polyhedron with exponentially many facets needs exponentially many constraints; actually, at least one per facet.

Moreover, Proposition 4.38 can be leveraged to show that certain sets are polyhedra by showing that they are affine projections of polyhedra. This can in particular be used to obtain the following basic fact on dominants.

Proposition 4.42

The dominant of a polyhedron is a polyhedron.

4.2.3 Convex separation theorems

Convex separation theorems are very natural statements about non-overlapping convex sets that, despite their simplicity, allow for deriving many of the classical results in linear programming and beyond, like the Strong Duality Theorem. In general, they are helpful to derive certificates because they are often stated in a way that guarantees that either two convex sets intersect or there is a hyperplane (strictly) separating them. Such either/or statements can often be used by designing convex sets whose intersection corresponds to one property of interest, like the possibility to improve some given feasible LP solution, in which case the separating hyperplane corresponds to a certificate that no improvement is possible.

Definition 4.43: (Strictly) separating hyperplanes

Let $Y, Z \subseteq \mathbb{R}^n$ be two sets. A hyperplane $H = \{x \in \mathbb{R}^n \colon a^\top x = \beta\}$ is called a (Y, Z)-*separating hyperplane*, or simply *separating hyperplane*, if Y is contained in one of the half-spaces defined by H and Z in the other one, i.e., either

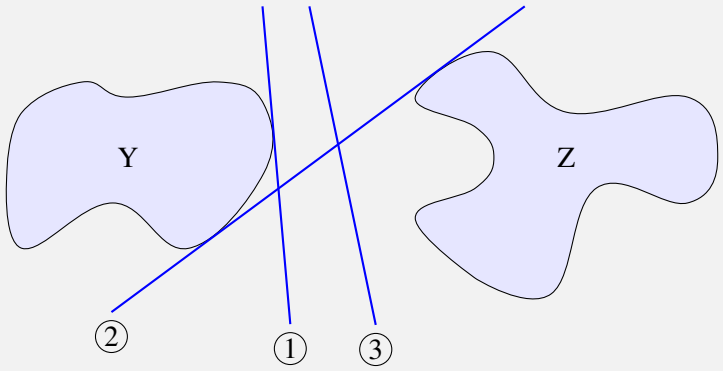
$$a^\top y \leq \beta \leq a^\top z \qquad \forall y \in Y, z \in Z \ , \text{ or}$$
$$a^\top y \geq \beta \geq a^\top z \qquad \forall y \in Y, z \in Z \ .$$

The hyperplane is called *strictly* (Y, Z)*-separating*, or simply *strictly separating*, if the above inequalities are strict.

If $Y = \{y\}$ is a single point, we also write (y, Z)-separating hyperplane instead of $(\{y\}, Z)$-separating hyperplane.

Example 4.44: Separating two sets

The illustration below shows two sets $Y, Z \subseteq \mathbb{R}^2$ together with three separating hyperplanes. Hyperplane 3 is strictly separating the sets whereas hyperplanes 1 and 2 do not separate Y and Z in a strict sense.

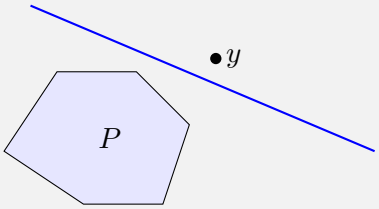


Theorem 4.45: Separating a point from a polyhedron

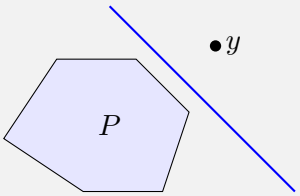
Let $P \subseteq \mathbb{R}^n$ be a polyhedron and $y \in \mathbb{R}^n \setminus P$. Then there is a strictly (y, P)-separating hyperplane.

Example 4.46: Separating a point from a polyhedron

The illustration below shows a hyperplane strictly separating a point from a polytope. Whenever a point is not contained in a polytope, it can always be strictly separated.



Moreover, one can always choose a separating hyperplane with the same normal vector as one of the constraints in any inequality description of P. In particular, if P is full-dimensional, like in the example above, we can use an inequality description that has one inequality per facet (see Corollary 4.19), and therefore separate y with a hyperplane that is parallel to a facet of P, as illustrated below.



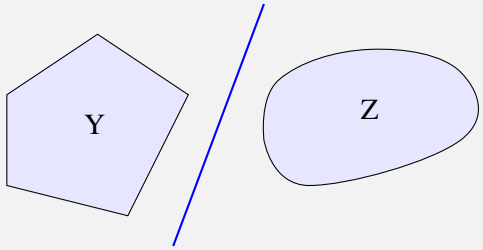
The above separation theorem is a very special case of the following much more general convex separation theorem.

Theorem 4.47

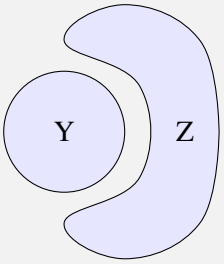
Let $Y, Z \subseteq \mathbb{R}^n$ be two disjoint closed convex sets with at least one of them being compact, then there exists a strictly (Y, Z)-separating hyperplane.

Example 4.48: Separating two sets

The illustration below shows two convex sets strictly separated by a hyperplane, which is always possible.



However, when at least one of the sets is not convex, separation may not be possible anymore as illustrated in the example below.



4.3 Linear duality

It turns out that for every linear program, there exists a closely related linear program, called the *dual problem*. The dual program can be seen as a different way to look at the original problem, which is typically referred to as the *primal*. The close relation between primal and dual can be exploited in several ways. In particular, the dual provides quick ways for showing optimality of an optimal primal solution. Moreover, given an optimal solution for either the primal or the dual often allows for quickly obtaining an optimal solution to the other problem. This connection can sometimes be used to find an optimal solution to the primal more quickly. More precisely, sometimes it is faster to use for example the Simplex Method to solve the dual problem and then derive an optimal primal solution from an optimal dual one. Duality is also key in the context of sensitivity analysis, and has various algorithmic applications, for example in so-called *primal-dual* procedures, which solve a problem by simultaneously exploiting properties of the primal and the dual of a problem.

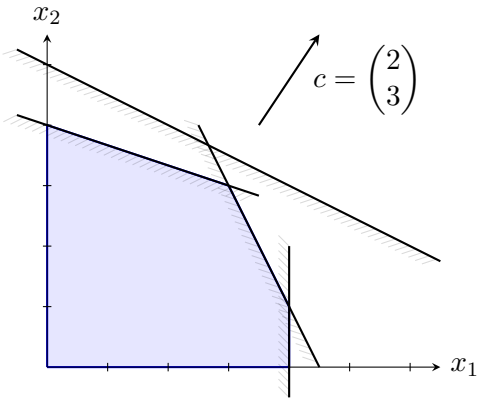
In the next section, we introduce linear duality as a way to find strong upper bounds for the

optimal value of a linear program.

4.3.1 Motivation: finding bounds on optimal value

Consider the linear program in canonical form given below. Its feasible region is highlighted on the right-hand side together with the objective function, which is maximized at the point $(x_1, x_2) = (3, 3)$.

$$\begin{array}{lrcrcl} \max & 2x_1 & + & 3x_2 & & \\ & x_1 & + & 3x_2 & \leq & 12 \\ & 2x_1 & + & x_2 & \leq & 9 \\ & x_1 & & & \leq & 4 \\ & x_1 & + & 2x_2 & \leq & 10 \\ & x_1 & & & \geq & 0 \\ & & & x_2 & \geq & 0 \end{array}$$



Assume that we want to find an upper bound on the optimal value of the above linear program. One way to obtain such a bound is by multiplying the constraint $x_1 + 2x_2 \leq 10$ by 2 to obtain

$$2x_1 + 4x_2 \leq 20 \ .$$

Because any solution to the LP satisfies $x_1, x_2 \geq 0$, we have

$$2x_1 + 3x_2 \leq 2x_1 + 4x_2 \leq 20 \ .$$

Hence, 20 is an upper bound on the optimal LP value. An obvious questions is whether we can obtain a better bound by a similar technique. Indeed, by adding up the constraints $x_1 + 3x_2 \leq 12$ and $x_1 \leq 4$ we get

$$2x_1 + 3x_2 \leq 16 \ ,$$

showing that the optimal LP value is upper bounded by 16. In general, to obtain a valid bound for the objective $2x_1 + 3x_2$, one can try to multiply each '$\leq$'-constraint by some non-negative factor such that a linear constraint $\alpha x_1 + \beta x_2 \leq \gamma$ is obtained with $\alpha \geq 2$, $\beta \geq 3$. Then γ is an upper bound to the optimal value of the LP.

Hence, finding the best such upper bound is itself an LP where we have a variable for each constraint of the original LP, except for the non-negativity constraints. Those variables describe the coefficients in the conic combination of the constraints. Furthermore, there is a constraint

for each variable in the original LP. For the example above it looks as follows.

$$\begin{array}{rrrrrrrrrr} \min & 12y_1 & + & 9y_2 & + & 4y_3 & + & 10y_4 & & \\ & y_1 & + & 2y_2 & + & y_3 & + & y_4 & \geq & 2 \\ & 3y_1 & + & y_2 & & & + & 2y_4 & \geq & 3 \\ & y_1 & & & & & & & \geq & 0 \\ & & & y_2 & & & & & \geq & 0 \\ & & & & & y_3 & & & \geq & 0 \\ & & & & & & & y_4 & \geq & 0 \end{array}$$

This problem is called the *dual problem* to the original problem. The unique optimal solution of this dual problem is $(y_1, y_2, y_3, y_4) = (\frac{4}{5}, \frac{3}{5}, 0, 0)$ and leads to an optimal value of 15. The way how we set up this dual problem implies that 15 is an upper bound to the value of the original LP. As it turns out, the optimal value of the dual LP is even equal to the optimal value of the original LP, which is achieved at the point $(3, 3)$. This is no coincidence. It is at the heart of one of the most important results in linear programming, namely strong duality.

In the following, we start by making the above discussion on constructing the dual more formal before covering some key results linked to linear duality, like strong duality.

4.3.2 Dual of a linear program

Let us revisit the above discussion in more generality. Consider a general linear program in canonical form, which we will call the *primal problem* to distinguish it later from the *dual problem*:

$$\begin{array}{rrcl} \max & c^\top x & & \\ & Ax & \leq & b \\ & x & \geq & 0 \ . \end{array} \tag{PLP}$$

Following the above discussion, its corresponding dual problem is:

$$\begin{array}{rrcl} \min & b^\top y & & \\ & A^\top y & \geq & c \\ & y & \geq & 0 \ . \end{array} \tag{DLP}$$

Notice that for each constraint in the primal, there is a variable in the dual, and vice versa. In particular, if the primal problem has n variables and m constraints (not counting the non-negativity constraints) then the dual problem has m variables and n constraints (not counting the non-negativity constraints).

4.3.2.1 Dualization of LPs not in canonical form

So far, we focussed on dualizing linear programs in canonical form. However, dualitzation can be applied to a linear program in an arbitrary form. Of course, one can simply bring an arbitrary linear program first into canonical form and then dualize it. However, one can also dualize any linear program directly, without first transforming it into canonical form. For example, if we have a constraint of type '$\geq$' in the primal, then we can use a dual variable for this constraint that is non-positive instead of non-negative.

Recall that if we first transform a problem into canonical form, then this can lead to extra variables and/or constraints, which leads to extra constraints and/or variables in the dual. Nevertheless, one can simplify the resulting dual to avoid the introduction of extra variables or constraints.

In the problem sets, we will expand on the above discussion about dualizing linear programs that are not in canonical form. In what follows, we focus on the case of primal problems that are in canonical form.

4.3.2.2 Dual of dual is primal

An interesting property of linear duality is that the dual of the dual is the primal problem. One way to see this is to first transform (DLP) into canonical form:

$$\begin{array}{rrcl} -\max & -b^\top y & & \\ & -A^\top y & \leq & -c \\ & y & \geq & 0 \ , \end{array} \tag{4.5}$$

and then dualizing the above problem, which leads to

$$\begin{array}{rrcl} -\min & -c^\top x & & \\ & -Ax & \geq & -b \\ & x & \geq & 0 \ , \end{array}$$

which is again the primal problem (PLP). As discussed, it is not necessary to first transform (DLP) into (4.5) to dualize it. One can immediately dualize it by again deriving how one can bound the objective value $b^\top y$ with the available constraints.

4.3.3 Weak and strong linear duality

Notice that by the way how we constructed the dual problem (DLP) from the primal problem (PLP), the value of any feasible solution to the dual upper bounds the value of any feasible solution to the primal. This result is known as *weak duality* and is easy to prove formally.

Theorem 4.49: Weak duality

Let x, y be feasible solutions to (PLP) and (DLP), respectively. Then

$$c^\top x \leq b^\top y \ .$$

Proof. The result follows from

$$c^\top x \leq (y^\top A)x = y^\top (Ax) \leq y^\top b = b^\top y \ ,$$

where the first inequality holds due to $c \leq A^\top y$ and $x \geq 0$, and the second inequality follows from $Ax \leq b$ and $y \geq 0$. □

Weak duality already comes with some interesting implications. It implies that if the primal is unbounded, then the dual must be infeasible. Since the primal-dual relation is symmetric, it also implies that if the dual is unbounded then the primal is infeasible. Furthermore, if one can find a primal feasible solution x and a dual feasible solution y such that $c^\top x = b^\top y$, then both x and y are optimal solutions for the primal and dual, respectively. Hence, in this case, the dual solution is an optimality certificate of the primal solution and vice versa.

Notice that weak duality does not rule out that both the primal and dual are infeasible, which is indeed possible.

Example 4.50: Infeasible primal and dual LP

Below is a pair of a primal and dual linear program that are both infeasible.

$$\begin{array}{llllll} \max & 2x_1 & - & x_2 & & \\ & x_1 & - & x_2 & \leq & 1 \\ & -x_1 & + & x_2 & \leq & -2 \\ & x_1 & & & \geq & 0 \\ & & & x_2 & \geq & 0 \end{array} \qquad \begin{array}{llllll} \min & y_1 & - & 2y_2 & & \\ & y_1 & - & y_2 & \geq & 2 \\ & -y_1 & + & y_2 & \geq & -1 \\ & y_1 & & & \geq & 0 \\ & & & y_2 & \geq & 0 \end{array}$$

Notice that the existence of such an example is not surprising due to the following. Consider two independent linear programs in canonical form, one that is infeasible and one for which the dual is infeasible. Then we can simply combine them into a single linear program maximizing the sum of both objectives. It is not hard to check that this leads to a new linear program for which both the primal and dual are infeasible.

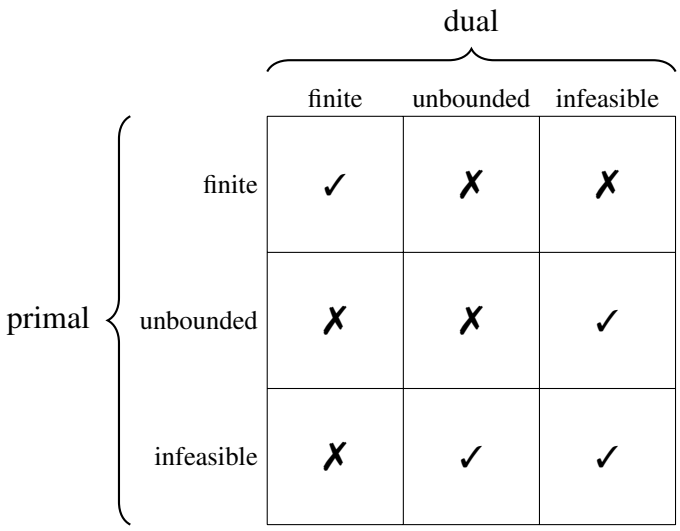
Interestingly, when the primal and dual are both feasible—which is often the most interesting case—then a cornerstone result in linear duality implies that their optimal values are equal. This result is known as *strong duality*.

Theorem 4.51: Strong duality

If the primal has a finite optimum, then also the corresponding dual problem has a finite optimum and their values are equal, i.e., there exists a feasible solution x for (PLP) and a feasible solution y for (DLP) such that $c^\top x = b^\top y$.

Strong duality implies that if an LP has a finite optimum, then the optimality of a solution to that LP can *always* be certified by exhibiting a dual solution with the same value. Moreover, strong duality has numerous implications in various combinatorial optimization problems. In particular, most max-min relations in Combinatorial Optimization, like the max-flow min-cut theorem, can be interpreted as a consequence of strong duality.

We recall that every linear program either (i) has a finite optimum, (ii) is unbounded, i.e., allows for solutions with arbitrarily strong objective values, or (iii) is infeasible. The table below shows in which of those 3 states an LP and its corresponding dual can be, where a check mark (✓) indicates that the corresponding combination is possible and a cross (✗) that it is not.



The above table is symmetric because the dual of the dual is the primal. The first row and first column follow from strong duality. The second row and column follow from weak duality. Finally, the check mark at the bottom right follows from the fact that there are pairs of primal and dual LPs that are both infeasible (see Example 4.50).

4.3.4 Complementary slackness

The complementary slackness theorem formalizes a crucial relation between optimal primal and dual solutions. The relation is very natural, especially when keeping in mind that a dual LP can be interpreted as a way to combine primal constraints to bound the optimal primal objective value. Complementary slackness says that, to obtain a best combination of primal constraints to bound the objective, one can only use primal constraints that are tight at (all) optimal primal solutions. Moreover, as the dual of the dual is again the primal, the statement is symmetric with respect to primal and dual.

Theorem 4.52: Complementary slackness theorem

Consider a pair of primal and dual linear programs with finite optima:

$$\begin{array}{rcl} \max\ c^\top x & & \\ Ax & \leq & b \\ x & \geq & 0 \end{array} \qquad\qquad \begin{array}{rcl} \min\ b^\top y & & \\ A^\top y & \geq & c \\ y & \geq & 0\ . \end{array}$$

Let $\overline{x}$ be a feasible primal solution and $\overline{y}$ a feasible dual solution. Then both $\overline{x}$ and $\overline{y}$ are optimal solutions (for the primal and dual, respectively) if and only if

(i) $(b - A\overline{x})^\top \overline{y} = 0$, and
(ii) $(A^\top \overline{y} - c)^\top \overline{x} = 0$.

Apart from providing an important insight in the relation between optimal primal and dual

solutions, the complementary slackness theorem can often be used to derive an optimal dual solution from a primal one or vice versa.

5 Integer Programming Basics

In short, integer programming is linear programming with the additional requirement that all variables need to take integral values. Integer programs (IPs) allow for modeling a very large class of discrete mathematical optimization problems. In particular, most classical combinatorial optimization problems can easily be modeled as IPs. However, this comes at a price. IPs form an NP-hard problem class and, depending on the problem type at hand, finding optimal (or even feasible) solutions may take excessive time. Nevertheless, strong methods have been developed to deal with many IPs that we face in real-world applications. In this chapter, we provide a brief introduction to some of the key techniques used to solve IPs in practice. This chapter is less focussed on specific theoretical properties, but rather provides a first glimpse into a broadly used and highly influential toolbox to solve IPs.

5.1 Introduction to Integer Programming

As mentioned, integer programs are defined analogously to linear programs with the only difference that all variables are required to take integer values, for example,

$$\begin{array}{rrcl} \max & c^\top x & & \\ & Ax & \leq & b \\ & x & \in & \mathbb{Z}^n \end{array},$$

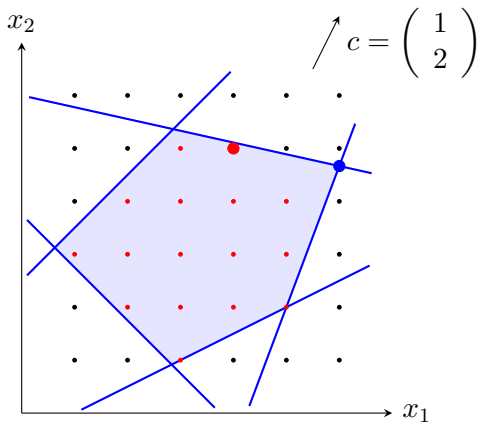
where $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, and $b \in \mathbb{R}^m$. As with linear programming, the above form is just one way to write integer programs. More generally, one can also use equalities or '$\geq$' constraints.

Figure 5.1 shows an example integer program in two dimensions. When forgetting about the integrality constraints, one obtains a linear program, which is called the *linear relaxation* of the IP. Many methods for solving IPs heavily rely on repeatedly solving linear relaxations of certain well-defined sub-problems, because they can be solved very quickly and their optimal value is an upper bound on the IP that corresponds to the relaxation. In particular, branch & bound and also branch & cut procedures fall into this category. The strength of the linear relaxation, which, loosely speaking, is how well the linear relaxation approximates the convex hull of the integer solutions, is a crucial factor impacting the speed of such procedures.

We start with a recreational application of IPs, by showing how they can be used to solve the eight queens puzzle.

5.1.1 IP example: the eight queens puzzle

The eight queens puzzle is a famous problem linked to chess. The goal of the puzzle is to place eight chess queens on a standard 8×8 chessboard so that no two queens threaten each other, i.e.,



$$
\begin{array}{lrcrcl}
\max & x_1 & + & 2x_2 & & \\
 & 2x_1 & + & 9x_2 & \leq & 54 \\
 & -2x_1 & + & 2x_2 & \leq & 5 \\
 & 4x_1 & + & 4x_2 & \geq & 15 \\
 & x_1 & - & 2x_2 & \leq & 1 \\
 & 8x_1 & - & 3x_2 & \leq & 34 \\
 & & & x_1, x_2 & \in & \mathbb{Z}
\end{array}
$$

Figure 5.1: Example of an integer program in two dimensions. The blue area shows the linear relaxation of the IP, i.e., the feasible solutions if one disregards the integrality constraints. The IP only optimizes over the integer points inside the relaxation, which are highlighted in red. The blue point shows the unique optimal solution of the linear relaxation, and the large red one shows the unique optimal solution of the IP.

there is no more than one queen in each row, column, and diagonal. Figure 5.2 shows the squares threatened by a queen and Figure 5.3 shows an example placement of seven queens where there are no unthreatened squares left to place an eighth queen.

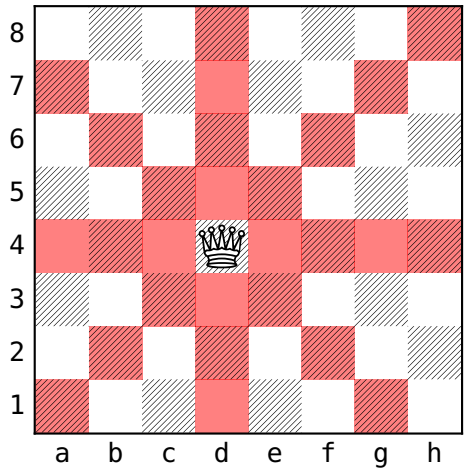


Figure 5.2: The squares in red are those where the shown queen can move to. Any other queen on one of those squares is said to be *threatened* by the shown queen.

To reformulate this combinatorial problem in terms of an integer program, we introduce a binary variable $x_{ij} \in \{0, 1\}$ for each square of the board, which is indexed by a pair $(i, j) \in [8] \times$

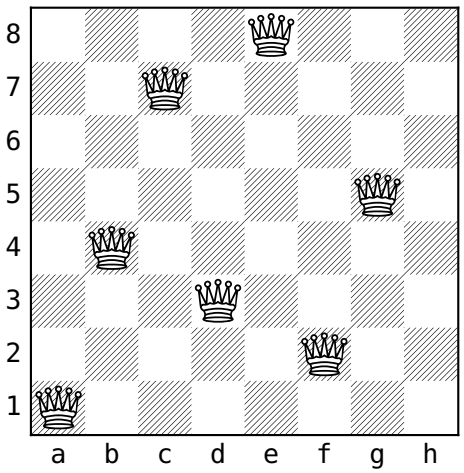


Figure 5.3: Example of a placement of seven queens on the board. Note that there is a queen in each of the files a-g and every square in the h-file is threatened, so there is no valid square to place the eighth queen. (A *file* in chess is a column of the chessboard.)

[8]. Hence, each such binary variable will simply be an integer variable with linear constraints $0 \leq x_{ij} \leq 1$. Moreover, we think of $x_{ij} = 1$ as placing a queen on square (i, j), and of $x_{ij} = 0$ as not doing so.

The constraints now follow in a straightforward way from the statement of the puzzle: For each row, column, and diagonal of the chessboard, the sum of the corresponding variables is at most 1. The objective of the optimization problem is to maximize the sum of all components of the vector x, i.e., the number of queens placed on the board. Thus, we arrive at the following IP:

$$\begin{aligned}
\max \quad & \sum_{i=1}^{8}\sum_{j=1}^{8} x_{ij} & & \\
& \sum_{j=1}^{8} x_{ij} \;\leq\; 1 & & \forall i \in [8] \\
& \sum_{i=1}^{8} x_{ij} \;\leq\; 1 & & \forall j \in [8] \\
& \sum_{\substack{i,j\in[8]:\\ i+j=k}} x_{ij} \;\leq\; 1 & & \forall k \in \{2, 3, \ldots, 16\} \\
& \sum_{\substack{i,j\in[8]:\\ i-j=k}} x_{ij} \;\leq\; 1 & & \forall k \in \{-7, -6, \ldots, 7\} \\
& x \;\in\; \{0,1\}^{8\times 8} \quad . & &
\end{aligned} \tag{5.1}$$

Let us check that the optimal solutions of the IP (5.1) correspond one-to-one to the solutions of the eight queens puzzle. On the one hand, the optimal value does not exceed 8 because a chessboard has 8 rows and there can be at most one queen per row. On the other hand, every solution of the puzzle corresponds to a feasible solution of the integer program attaining the value 8, which is the upper bound. Therefore, the optimal solutions of the integer program are precisely the solutions of the eight queens puzzle.

Figure 5.4 shows an optimal solution to the problem.

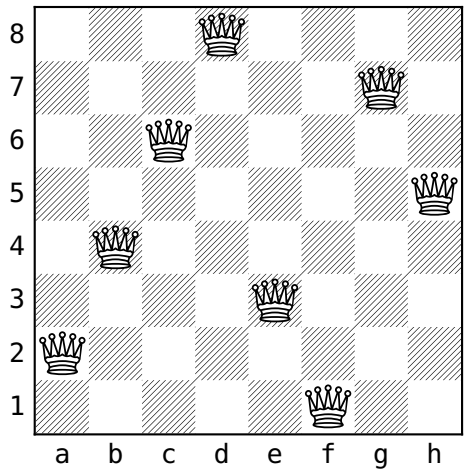


Figure 5.4: A solution to the eight queens puzzle.

It turns out that the eight queens puzzle has 12 distinct solutions that are not related by rotations or reflections, and 92 solutions in total.

Exercise 5.1

Construct an IP to find a solution to the eight queens puzzle that cannot be obtained from the one shown in Figure 5.4 through rotations and reflections.

5.2 Branch & bound

Branch & bound, and various variations thereof, is one of the most common solution approaches for integer programs. The principal idea is to intelligently explore the solution space and to try to cut off large parts of the solution space as soon as one can be sure that there is no optimal solution in them.

To illustrate the approach we consider the following IP with binary variables. The approach

can easily be extended to integer variables with larger ranges.

$$\begin{array}{rrcrcrcrcl} \max & 75x_1 & + & 6x_2 & + & 3x_3 & + & 33x_4 & & \\ & 774x_1 & + & 76x_2 & + & 22x_3 & + & 42x_4 & \leq & 875 \\ & 67x_1 & + & 27x_2 & + & 794x_3 & + & 53x_4 & \leq & 875 \\ & & & & & & & x & \in & \{0,1\}^4 \end{array}$$

We start by replacing the integrality constraints $x \in \{0,1\}^4$ by $x \in [0,1]^4$ and solve the resulting linear relaxation. This leads to the solution shown in box 1 of Figure 5.5 (the number of the box is highlighted with a gray background), i.e.,

$$(x_1, x_2, x_3, x_4) = (1, 0.506, 0.934, 1) \ ,$$

with objective value $z = 113.837$. If this LP solution had been integral, we could have stopped right-away and return it as an optimal solution. Indeed, the LP we solved optimized over a bigger solution set then the actual one, obtained from the original IP by ignoring the integrality requirement on the variables. However, the values of x_2 and x_3 are not integral. We choose one of these variables, say x_2, and *branch* on this variable. More precisely, we create two sub-problems, one in which we force x_2 to be equal to 0 (box 9 in Figure 5.5) and one with $x_2 = 1$ (box 2). (If x_2 had been an integer variable with a larger range instead of a binary one, then we could have branched into two problems where one requires $x_2 \leq 0$ and the other one $x_2 \geq 1$.)

The optimal solution to the original problem is the better of the optimal solutions to the two newly created problems through branching. Branch & bound now recurses on the sub-problems, leading to a so-called branch & bound tree as shown in Figure 5.5. This tree is often constructed in a depth-first order. The numbering of the boxes in Figure 5.5 highlights such a depth-first order in which the sub-problems may be constructed and explored in a branch & bound procedure.

Let us start with the first 5 boxes. Here we successively branch on a fractional variable. In this case only the 5th box, where all variables are set to integral values, leads to a feasible integral solution. If it had happened that already an earlier sub-problem, say the one corresponding to box 4, had been integral, then we would not have branched further. Box 5 is the first feasible solution we found and we save it together with its value. We backtrack to the previous problem, the one corresponding to box 4, and go into the branch $x_3 = 1$. This leads to a problem where all variables got fixed through the branching, and the values to which they got fixed turn out to be infeasible. We therefore backtrack to the closest box above box 6 for which one of the two branches has not been explored yet. This is box 3, and we explore the other branch, i.e., $x_4 = 1$. In this case, the resulting LP is infeasible, and we can therefore stop exploring this branch further. Clearly, if the LP is infeasible, then so is the IP, which has integrality constraints on top of the LP.

A new important step happens at the next box we explore after backtracking. This is box 8, for which the optimal LP value is 42. Because this value is strictly smaller than the value of the best solution we found so far, which was found in box 5 and has value 81, we can stop exploring this box any further. Notice that coincidentally, the optimal solution to the LP of box 8 happens to be integral. However, this is irrelevant for this reasoning. Indeed, if even the linear relaxation cannot improve on the currently best solution, then there is no reason to branch further to obtain

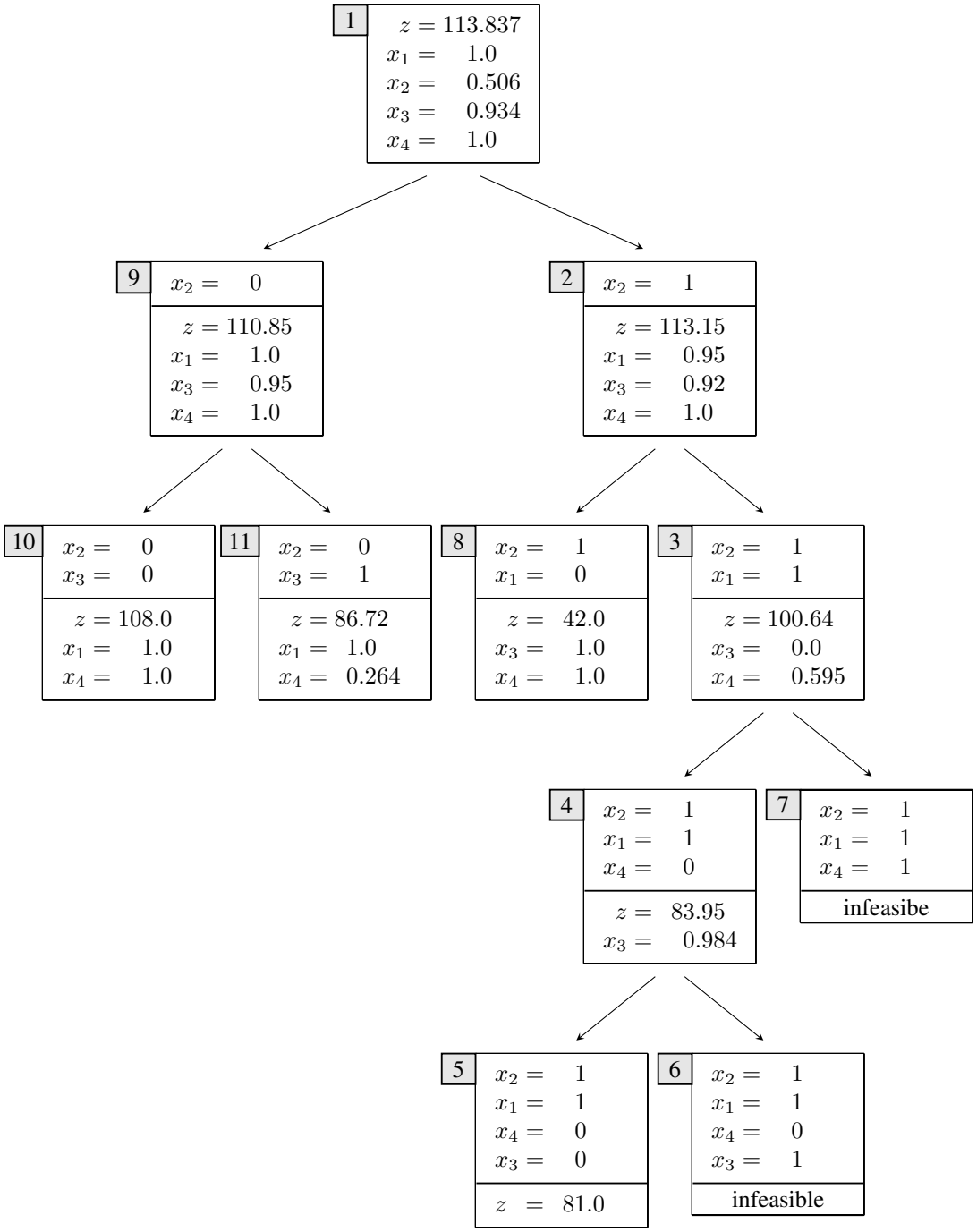


Figure 5.5: A branch & bound tree. Box 1 shows the optimal solution to the linear relaxation of the original problem. This solution is fractional, and we branch on the variable x_2 to obtain two new LP solutions shown in boxes 9 and 2, and so on. The lower part of each box shows the optimal LP solution to the problem when fixing variables as shown in the upper part of the box.

an integral solution. This step is called the *bounding* step of the branch & bound procedure and explains the second part of its name.

We proceed as explained, and find the problem in box 10, whose optimal LP solution happens to be integral and improves on the best solution found previously. The last sub-problem we consider is the one in box 11, where we have an optimal LP value of only 86.72, which is less than the value of 108 of our best solution found so far, coming from box 10. Hence, no further exploration of problems stemming from box 11 is necessary. At this point, the branch & bound tree has been fully explored, and we return the best solution found, which is the one from box 10.

Remark 5.2: Incumbent

The best solution found so far is often called the *incumbent*.

Remark 5.3: Non-uniqueness of branch & bound tree

The tree constructed in this way is not unique, because there are various orders in which the branchings can be performed. First, one can choose the variable on which to branch, and second, one can choose on which of the two children to continue first when doing a branching.

Remark 5.4: Branching on linear expression instead of variables

In our example, a branching was always with respect to a variable. For example, the first branching split the original problem into two sub-problems, one whose solutions fulfill $x_2 = 0$ and one whose solutions fulfill $x_2 = 1$. However, one can also branch on a linear expression. For example, we could have branched on the expression $x_1 + x_2 - x_3$, also splitting the original problem into two sub-problems, one where $x_1 + x_2 - x_3 \leq 0$ and one where $x_1 + x_2 - x_3 \geq 1$. Notice that also this way of splitting turns the optimal solution to the original linear relaxation infeasible in both sub-problems. Sometimes, by identifying strong problem-specific expressions (or also variables) to branch on, branch & bound procedures can be sped up significantly.

Remark 5.5: Speed-ups via strong incumbents

For a branch & bound procedure to be fast, it is crucial that a strong feasible solution is found quickly, because this can be used for the bounding step in the future, i.e., to discard certain sub-problems because even their linear relaxation has a worse objective than the value of the incumbent. This explains why we applied depth-first search to explore the branch & bound tree, with the hope to quickly find a feasible solution. Moreover, branch & bound procedures are typically combined with various heuristics to quickly obtain strong feasible solutions, which then helps to speed up the exploration of the solution space through branch & bound.

Remark 5.6: Speed-ups via strong relaxations

As we discussed, the reason why branch & bound procedures do not need to enumerate over all possible solutions, but can sometimes discard large parts of the solution space, is due to

bounding. The bounding step is based on comparing solutions to LP relaxations with the value of the incumbent. For this to be effective, we do not only need strong incumbents, but it is also paramount to use a strong linear relaxation to start with. The arguably strongest LP relaxation is one that describes the convex hull of all solutions of the IP. Indeed, an optimal vertex solution to this "perfect" relaxation corresponds to an optimal IP solution. For hard problems, there is little hope that we can get a good inequality description of this convex hull. Nevertheless, the identification of strong valid inequalities for the linear relaxation can be a crucial ingredient to solve large IPs.

Remark 5.7: Branch & bound for mixed-integer linear programming (MIP)

Branch & bound techniques can also be applied to so-called *mixed-integer linear programs* (in short, MIPs), which are a mix of LP and IP in the sense that some variables are continuous and others are required to be integral. Hence, an MIP can be formulated as a problem of the form:

$$\max\{c_1^\top x + c_2^\top y \colon A_1 x + A_2 y \leq b, x \in \mathbb{R}^{n_1}, y \in \mathbb{Z}^{n_2}\} \ ,$$

where $c_i \in \mathbb{R}^{n_i}$ and $A_i \in \mathbb{R}^{m \times n_i}$ for $i \in \{1, 2\}$, and $b \in \mathbb{R}^m$.[a] The branch & bound approach can be applied to MIPs by only branching on integer variables. This way it can be used to solve MIPs in the same way we approached IPs.

[a] Analogous to LPs and IPs, where we talked about generic forms like the canonical form, there are different ways to define a generic form for MIPs.

5.3 Branch & cut

Branch & cut is an enhancement of branch & bound procedures through so-called cutting planes. More precisely, consider a sub-problem in the branch & bound procedure where we would normally branch, i.e., the sub-problem

(i) is not infeasible,
(ii) the value of its LP relaxation is strictly better than the value of the incumbent,
(iii) its LP solution is not integral.

Hence, for the example considered in the previous section, this could be a sub-problem corresponding to any of the boxes 1, 2, 3, 4, or 9 in Figure 5.5. Instead of branching at such a node, a branch & cut procedure may first add so-called *cutting planes*. These are linear inequalities that cut off the current optimal LP solution, but do not cut off any integral solution. Figure 5.6 shows again the feasible solutions to our introductory IP for branch & bound together with a cutting plane (in green) that cuts off the current optimal LP solution. By adding a cutting plane, the LP relaxation is strengthened. One then solves this strengthened LP relaxation and continues the procedure with that one.

The stronger LP, obtained after adding a cutting plane, leads to a better upper bound and makes it easier to use bounding, because the optimal value of the LP may have decreased after adding a cutting plane. Also, it is possible that the strengthened LP has an LP relaxation with

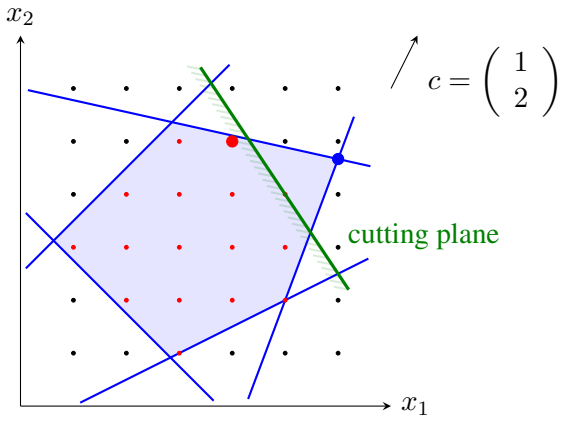


Figure 5.6: Example of a cutting plane. The cutting plane should cut off the current optimal solution to the linear relaxation but must not cut off any integral solutions.

an integral optimal solution, or one that is infeasible; in both cases, no further branching is necessary. In short, the goal of adding cutting planes is to speed up the procedure by reducing the total number of branchings, and hence sub-problems to be solved.

The study of cutting planes is a research area in its own. An in-depth discussion of cutting planes is beyond the scope of this script. Still, to provide a glimpse into how cutting planes can be generated automatically, we provide a brief explanation of one of the most important cut-families used in practice, namely so-called Chvátal-Gomory cuts. When using the Simplex Method to optimize the relaxation, these cuts can readily be derived from the so-called simplex tableau. However, to make the exposition more accessible, we provide an introduction to Chvátal-Gomory cuts that does not require a thorough knowledge of the Simplex Method.

5.3.1 An introductory example for Chvátal-Gomory cuts

To illustrate the idea of Chvátal-Gomory cuts, or short *CG cuts*, we first consider the following simple integer program. The feasible region of its linear relaxation is highlighted in Figure 5.7.

$$\begin{array}{rrcrcl} \max & x_1 & + & x_2 & & \\ & 2x_1 & + & x_2 & \leq & 5 \\ & 2x_1 & + & 5x_2 & \leq & 7 \\ & & & x & \in & \mathbb{Z}^2_{\geq 0} \end{array} \tag{5.2}$$

The linear relaxation of the integer program (5.2) has a unique optimal solution, given by

$$(x_1, x_2) = (2.25, 0.5) \ , \tag{5.3}$$

with an objective value of 2.75. Notice that (5.2) has two optimal solutions, namely $(2, 0)$ and $(1, 1)$, both of which have value 2. Our goal is to introduce a cutting plane that cuts off the optimal fractional solution highlighted above. The idea of CG cuts to achieve this goal is to

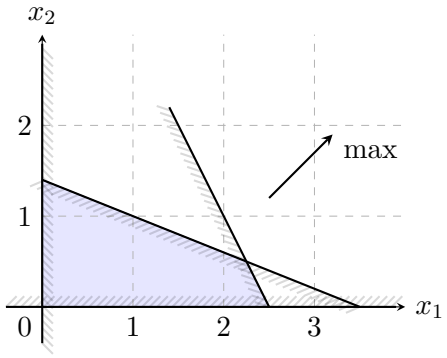


Figure 5.7: Graphical representation of feasible solutions for the linear relaxation of the integer program (5.2).

take a combination of given constraints and then round the coefficients in a way that guarantees that the resulting constraint remains valid for any integer feasible solution, while cutting off the optimal LP solution. We start with a simple example of a CG cut, where, in a first step, we multiply the second constraint by 0.5 to obtain the following equivalent constraint:

$$x_1 + 2.5x_2 \leq 3.5 \ . \tag{5.4}$$

Because $x_2 \geq 0$, we obtain another valid constraint by rounding down the coefficient of x_2, i.e., 2.5, thus obtaining

$$x_1 + 2x_2 = x_1 + \lfloor 2.5 \rfloor x_2 \leq 3.5 \ .$$

Of course, this just led to a weaker constraints than the constraint (5.4) with which we started. However, now we have a valid constraint where all left-hand side coefficients are integers. Because we optimize only over integer values of x_1 and x_2, this implies that the left-hand side will always achieve integer values for any IP-feasible point. Hence, and this is the crucial observation in CG cuts, we can round down the right-hand side to get the following constraint, which is valid for any feasible integer solution:

$$x_1 + 2x_2 \leq \lfloor 3.5 \rfloor = 3 \ . \tag{5.5}$$

Apart from being valid, this constraint cuts off the fractional optimum of the relaxation, highlighted in (5.3). Thus, we can use the constraint (5.5) as a cutting plane in a branch & cut procedure to strictly sharpen the current relaxation.

In summary, we first obtained a constraint that is valid even for the linear relaxation, namely the constraint shown in (5.4). Using the fact that we optimize over non-negative integer variables, we can round down all left-hand side coefficients as well as the right-hand side. Cuts obtained this way are called CG cuts.

Whereas in the above example we started with a scaled version of just a single one of the constraints, one can often obtain stronger CG cuts by first combining several of the given constraints to obtain a valid constraint for the relaxation, whose coefficients will then be rounded.

For example, by multiplying the first constraint by $3/8$, the second one by $1/8$, and adding up these scaled constraints, we obtain the following constraint, which is valid for the LP relaxation:

$$x_1 + x_2 \leq 2.75 \ .$$

This constraint already has integer left-hand side coefficients. Hence, we obtain a stronger constraint that is valid for the IP by rounding down the right-hand side, which leads to

$$x_1 + x_2 \leq \lfloor 2.75 \rfloor = 2 \ .$$

After adding the above constraint to the LP relaxation, we obtain a strengthened LP relaxation whose optimal value is even identical to the optimal value of the IP. This was a coincidence. In general, there may not exist a CG cut whose addition will lead to a new relaxation with same optimal value as the IP. However, one can show that, starting with a bounded rational relaxation, there always exists a sequence of CG cuts that allow for obtaining the "perfect" relaxation, i.e., the convex hull of all integer solutions. However, to achieve this, it may be necessary that later CG cuts build new valid inequalities by combining not just the original inequalities but also inequalities corresponding to previous CG cuts.

Whereas the above example was convenient to showcase CG cuts, we still lack one important ingredient to use them in practice. Namely, we need a way to generate such cuts automatically through a well-defined recipe. In the next section, we highlight one way to obtain CG cuts from an optimal vertex solution to the linear relaxation.

5.3.2 Generating a Chvátal-Gomory cut from optimal vertex LP solution

We now illustrate how one can obtain CG cuts from an optimal vertex solution to the LP relaxation. For this, we assume that the relaxation we start with is bounded, in which case it is either infeasible (and we can stop because this implies that the IP is infeasible) or admits an optimal vertex solution. To exemplify this approach, consider again our introductory IP example for branch & bound, which we repeat below for convenience.

$$\begin{array}{rrcrcrcrcl} \max & 75x_1 & + & 6x_2 & + & 3x_3 & + & 33x_4 & & \\ & 774x_1 & + & 76x_2 & + & 22x_3 & + & 42x_4 & \leq & 875 \\ & 67x_1 & + & 27x_2 & + & 794x_3 & + & 53x_4 & \leq & 875 \\ & & & & & & & x & \in & \{0,1\}^4 \end{array}$$

We show how to add a cutting plane to the linear relaxation of this problem, i.e., the problem corresponding to box 1 in Figure 5.5. As shown in Figure 5.5 and discussed previously, an optimal solution to the corresponding linear relaxation is given by

$$(\bar{x}_1, \bar{x}_2, \bar{x}_3, \bar{x}_4) = (1, 0.506, 0.9337, 1) \ .$$

(The entries are rounded, and for better accuracy, we added a digit to the value of $\bar{x}_3$ compared to box 1 in Figure 5.5.) Moreover, this solution is actually an optimal vertex solution.

To obtain a CG cut that cuts off $(\bar{x}_1, \bar{x}_2, \bar{x}_3, \bar{x}_4)$, we first put the problem in what is known as standard form in the context of linear programming, by introducing slack variables $y_1, \ldots, y_6$.[1]

[1] A linear program in standard form is of the following type: $\max\{c^\top x \colon Ax = b, x \in \mathbb{R}^n_{\geq 0}\}$.

The only difference to the linear programming setting is that, here, the involved variables are required to be integral. This leads to the following equivalent description of the above integer program.

$$
\begin{array}{lrcrcrcrcrcrcrcl}
\max & 75x_1 & + & 6x_2 & + & 3x_3 & + & 33x_4 & & & & & & & \\
 & 774x_1 & + & 76x_2 & + & 22x_3 & + & 42x_4 & +y_1 & & & & & = & 875 \\
 & 67x_1 & + & 27x_2 & + & 794x_3 & + & 53x_4 & & +y_2 & & & & = & 875 \\
 & x_1 & & & & & & & & & +y_3 & & & = & 1 \\
 & & & x_2 & & & & & & & & +y_4 & & = & 1 \\
 & & & & & x_3 & & & & & & & +y_5 & = & 1 \\
 & & & & & & & x_4 & & & & & +y_6 & = & 1 \\
 & & & & & & & & & & & & x & \in & \mathbb{Z}^4_{\geq 0} \\
 & & & & & & & & & & & & y & \in & \mathbb{Z}^6_{\geq 0}
\end{array}
\tag{5.6}
$$

The optimal solution $(\bar{x}_1, \bar{x}_2, \bar{x}_3, \bar{x}_4)$ to the relaxation naturally extends to the space including the newly introduced variables $y_1, \dots, y_6$ through the equations in the above description. This leads to the following optimal vertex solution to the linear relaxation of (5.6) (as before, the entries are rounded):

$$(\bar{x}, \bar{y}) = (\bar{x}_1, \bar{x}_2, \bar{x}_3, \bar{x}_4, \bar{y}_1, \dots, \bar{y}_6) = (1, 0.506, 0.9337, 1, 0, 0, 0, 0.494, 0.0663, 0) \ .$$

Because $(\bar{x}, \bar{y})$ is a vertex of the linear relaxation of (5.6), Proposition 4.21 implies that it is the unique solution to the following equation system, which consists of all equations that correspond to $(\bar{x}, \bar{y})$-tight constraints of (5.6):

$$
\begin{array}{rcrcrcrcrcrcrcl}
774x_1 & + & 76x_2 & + & 22x_3 & + & 42x_4 & +y_1 & & & & & = & 875 \\
67x_1 & + & 27x_2 & + & 794x_3 & + & 53x_4 & & +y_2 & & & & = & 875 \\
x_1 & & & & & & & & & +y_3 & & & = & 1 \\
 & & x_2 & & & & & & & & +y_4 & & = & 1 \\
 & & & & x_3 & & & & & & & +y_5 & = & 1 \\
 & & & & & & x_4 & & & & & +y_6 & = & 1 \\
 & & & & & & & y_1 & & & & & = & 0 \\
 & & & & & & & & y_2 & & & & = & 0 \\
 & & & & & & & & & y_3 & & & = & 0 \\
 & & & & & & & & & & & y_6 & = & 0
\end{array}
\tag{5.7}
$$

Because $(\bar{x}, \bar{y})$ is the unique solution to (5.7), the equation system (5.7) has full column rank. In our example, the system is even a square system, which can be written as follows in matrix

form:

$$\begin{pmatrix} 774 & 76 & 22 & 42 & 1 & 0 & 0 & 0 & 0 & 0 \\ 67 & 27 & 794 & 53 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{pmatrix} = \begin{pmatrix} 875 \\ 875 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}. \tag{5.8}$$

If the above system had not been a square system, we could have continued with an arbitrary full-rank square subsystem.

Some of the variables in the above equation system are set to zero due to tight non-negativity constraints. In our example these are y_1, y_2, y_3, and y_6. Consider now the submatrix of the above system consisting of all rows corresponding to equations not stemming from tight non-negativity constraints—these are the first 6 constraints in our example—and all variables except for the 4 that have been set to zero due to tight non-negativity constraints. This leads to the following matrix:

$$A_B := \begin{pmatrix} 774 & 76 & 22 & 42 & 0 & 0 \\ 67 & 27 & 794 & 53 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}.$$

Notice that the above matrix must have full rank because the matrix in (5.8) has full rank. Hence, we can perform elementary row operations over the first 6 rows of (5.8) such that the submatrix A_B will become an identity matrix. This corresponds to modifying the first 6 equations of (5.8)—which are also equations of (5.6), i.e., the original problem in standard form—by left-multiplying both the coefficient matrix of the first 6 rows and the corresponding right-hand side by A_B^{-1}. This leads to the following equation system, which is implied by the equations in (5.6): (In this example it is actually even equivalent to the equations in (5.6) because the rows of A_B

correspond to all equations of (5.6).)

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0.0133 & -0.0004 & -10.2608 & 0 & 0 & -0.5386 \\ 0 & 0 & 1 & 0 & -0.0005 & 0.0013 & 0.2645 & 0 & 0 & -0.0484 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & -0.0133 & 0.0004 & 10.2608 & 1 & 0 & 0.5386 \\ 0 & 0 & 0 & 0 & 0.0005 & -0.0013 & -0.2645 & 0 & 1 & 0.0484 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{pmatrix} = \begin{pmatrix} 1 \\ 0.506 \\ 0.9337 \\ 1 \\ 0.494 \\ 0.0663 \end{pmatrix} . \tag{5.9}$$

Notice that the right-hand side of the above equation system corresponds to the non-zero values among $(\bar{x}, \bar{y})$. For the reader with further background about the Simplex Method, we highlight that the above equation system is revealed by the so-called *tableau* of the Simplex Method. More precisely, the system (5.9) can be read off the optimal tableau obtained after the pivoting steps.

Because our optimal vertex solution to the relaxation is not integral—for otherwise, the computed related vertex solution would be an optimal solution to the original IP—also the right-hand side of the above system is not integral. This is what we exploit to create a CG cut that cuts off $(\bar{x}, \bar{y})$. More precisely, from any equation of the above system with a fractional right-hand side, we can create a CG cut that cuts off $(\bar{x}, \bar{y})$. Consider for example the second equation:

$$x_2 + 0.0133y_1 - 0.0004y_2 - 10.2608y_3 - 0.5386y_6 = 0.506 \ . \tag{5.10}$$

By rounding down the coefficients of the left-hand side to the next integer, we obtain the following weaker constraint with integral coefficients:

$$x_2 + \lfloor 0.0133 \rfloor y_1 + \lfloor -0.0004 \rfloor y_2 + \lfloor -10.2608 \rfloor y_3 + \lfloor -0.5386 \rfloor y_6 \leq 0.506 \ ,$$

i.e.,

$$x_2 - y_2 - 11y_3 - y_6 \leq 0.506 \ . \tag{5.11}$$

As with our introductory example, all of the left-hand side coefficients are now integral. Hence, we use that any integral solution to our IP, when plugged into the above inequality, will evaluate to an integral left-hand side. Thus, an inequality valid for all IP solutions is obtained by rounding down the right-hand side, leading to

$$x_2 - y_2 - 11y_3 - y_6 \leq \lfloor 0.506 \rfloor = 0 \ . \tag{5.12}$$

Moreover, the optimal vertex solution $(\bar{x}, \bar{y})$ we considered must violate this constraint due to the following. As (5.10) is an equation implied by the original equation system, $(\bar{x}, \bar{y})$ must fulfill it. Notice that by construction of the equation (5.10), all of the variables involved in the constraint, except for x_2, are set to zero by $(\bar{x}, \bar{y})$. Hence, even after rounding down the coefficients of these

variables, the thus obtained valid inequality (5.11) is also fulfilled with equation by $(\bar{x}, \bar{y})$; in other words, the inequality (5.11) is $(\bar{x}, \bar{y})$-tight. However, because we have a non-integral right-hand side, the inequality (5.12) is strictly stronger then the inequality (5.11), and thus is violated by $(\bar{x}, \bar{y})$. Hence, the cutting plane we found, given by (5.12), indeed cuts off the computed optimal vertex LP solution, and we can thus add it to the LP to make progress in a branch & cut procedure.

Finally, we can rewrite (5.12) in the space of the original variables x_1, x_2 ,x_3, and x_4, by substituting y_2, y_3, and y_6 with the equations given in (5.7). This leads to the following Chvátal-Gomory cut in original space:

$$78x_1 + 28x_2 + 794x_3 + 54x_4 \leq 887 \ .$$

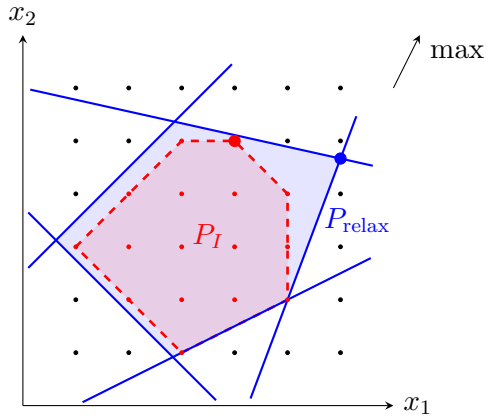
6 Tightness of Formulations

As discussed in Chapter 5, the practical efficiency of branch & bound and branch & cut procedures often depends crucially on the strength/tightness of the linear programming relaxation. A strong relaxation will typically significantly speed up the procedure, because the bounds obtained by solving the relaxations are much closer to the actual optimal integral solution than if a weak relaxation is used. Stronger relaxation bounds lower the threshold for pruning nodes in branch & bound algorithms. This makes it possible to prune nodes that, with a weaker relaxation, would have had to be further explored. Because IP solvers typically rely heavily on (variations of) the natural relaxation of the given IP, it can be of great importance to make sure that this relaxation is strong to speed up the running time. Due to this, it is often advantageous to add constraints to an IP that strengthen the relaxation even though they may not have any impact on the feasible integral solutions.

In the following we expand on these points. We start by discussing some general concepts regarding linear relaxations and their tightness or looseness. We then exemplify the above-mentioned idea of strengthening linear relaxations of integer programs.

6.1 Basics on LP relaxations and tight formulations

Consider again the integer program shown in Figure 5.1, which we repeat below in Figure 6.1 for convenience. We use this to exemplify some general concepts.



$$
\begin{array}{lrcrcl}
\max & x_1 & + & 2x_2 & & \\
\text{s.t.} & 2x_1 & + & 9x_2 & \leq & 54 \\
& -2x_1 & + & 2x_2 & \leq & 5 \\
& 4x_1 & + & 4x_2 & \geq & 15 \\
& x_1 & - & 2x_2 & \leq & 1 \\
& 8x_1 & - & 3x_2 & \leq & 34 \\
& x_1 & & & \in & \mathbb{Z} \\
& & & x_2 & \in & \mathbb{Z}
\end{array}
$$

Figure 6.1: An example integer program in two dimensions. The blue polytope P_{relax} shows the linear relaxation of the integer program. The red points are all feasible points for the integer program. Moreover, the red polytope P_I is the convex hull of all feasible solutions, which is also called the integer hull.

The IP can be written in matrix form as

$$\max\left\{c^\top x\colon Ax \leq b, x \in \mathbb{Z}^n\right\} \ . \tag{6.1}$$

Hence, the integer points over which we optimize are

$$Z \coloneqq \{x \in \mathbb{Z}^n\colon Ax \leq b\} \ .$$

The blue polytope P_{relax} shows the natural linear relaxation, i.e.,

$$P_{\text{relax}} \coloneqq \{x \in \mathbb{R}^n\colon Ax \leq b\} \ .$$

Notice that by definition, the integer points in the linear relaxation are precisely all feasible points, i.e., $Z = P_{\text{relax}} \cap \mathbb{Z}^n$. Clearly, to describe the set Z in terms of an integer program with relaxation P, one can use any polyhedron $P \subseteq \mathbb{R}^n$ such that $Z = P \cap \mathbb{Z}^n$. Any polyhedron $P \subseteq \mathbb{R}^n$ satisfying $Z = P \cap \mathbb{Z}^n$ is called *a (linear) relaxation* of Z. If P_1 and P_2 are two relaxations of the same set Z with $P_1 \subsetneq P_2$, then we say that P_1 is *tighter* than P_2 or, equivalently, that P_2 is *looser* than P_1. The tightest linear relaxation of Z is the convex hull P_I of Z, which is known as the *integer hull*:

$$P_I \coloneqq \text{conv}(Z) \ .$$

Indeed, as previously discussed, the convex hull of a set Z is the smallest convex set containing Z, which implies that P_I is tighter than any other relaxation of Z.

A crucial property of the integer hull P_I is that it allows for solving the original IP as a linear program instead of an integer one. More precisely, we have

$$\max\{c^\top x\colon Ax \leq b, x \in \mathbb{Z}^n\} = \max\{c^\top x\colon x \in P_I\} \ .$$

Moreover, by construction, the vertices of P_I are a subset of Z. Hence, an optimal vertex solution to the LP shown above (if an optimal vertex solution exists) is also an optimal solution to the IP. This implies that if P_I is a polytope and we are able to find a *compact*, i.e., polynomial-size, inequality description of P_I, then we can solve the original integer problem (6.1) by computing a vertex solution to $\max\{c^\top x\colon x \in P_I\}$ through an appropriate polynomial-time linear programming procedure.

This approach of solving an IP through an LP over its integer hull is relevant for problem classes that are polynomial-time solvable. However, for NP-hard problems, there is little hope that there is a small inequality description of P_I due to the common belief that $\mathsf{P} \neq \mathsf{NP}$. In general, we expect that hard problem classes that can be modeled as IPs, like maximum cardinality stable sets, lead to very complex integer hulls with exponentially many facets.

6.2 Illustrative examples

We now provide a few illustrative examples of problems that can easily be captured by integer programs and allow us to showcase how this can be done using relaxations of different strengths.

6.2.1 Matchings

6.2.1.1 Bipartite matchings

We start with maximum weight bipartite matchings. Hence, let $G = (V, E)$ be a bipartite graph with edge weights $w\colon E \to \mathbb{R}_{\geq 0}$ and the task is to find a matching $M \subseteq E$ maximizing $w(M)$. This problem is readily captured by the following integer program.

$$\max \left\{ w^\top x \colon x(\delta(v)) \leq 1 \ \forall v \in V, x \in \mathbb{Z}^E_{\geq 0} \right\} \ , \tag{6.2}$$

where $x(\delta(v)) \coloneqq \sum_{e \in \delta(v)} x(e)$. (In general, for a finite set N, a vector $x \in \mathbb{R}^N$, and a subset $U \subseteq N$, we use $x(U)$ as a shorthand for $\sum_{e \in U} x(e)$.) Notice that the non-negativity constraints together with the constraints $x(\delta(v)) \leq 1$ imply $x_e \leq 1$ for each $e \in E$. Hence, the variables x_e in (6.2) can be thought of as being binary variables. This is a classical way of capturing combinatorial optimization problems like matchings, where, for each edge $e \in E$, the variable x_e can be interpreted to be an indicator of whether we want to include e in M, in which case we set $x_e = 1$, or not, which corresponds to $x_e = 0$.

The natural linear relaxation of (6.2) is obtained by removing the requirement of x being integral, leading to the following linear program.

$$\max \left\{ w^\top x \colon x(\delta(v)) \leq 1 \ \forall v \in V, x \in \mathbb{R}^E_{\geq 0} \right\} \ . \tag{6.3}$$

It turns out that in the case of bipartite matchings, the polytope described by the above linear program is the integer hull of the solutions of (6.2), i.e., it is the integer hull of

$$Z \coloneqq \{ x \in \mathbb{Z}^E_{\geq 0} \colon x(\delta(v)) \leq 1 \ \forall v \in V \} \ . \tag{6.4}$$

Hence, as discussed, in this case one can find a maximum weight matching by finding an optimal vertex solution to (6.3), and there is no need to approach the problem through the IP (6.2) by using IP solver techniques. Observe that due to the strong (actually even perfect/tight) relaxation, also typical IP approaches like branch & bound would find an optimal solution very quickly because they also solve a linear relaxation of the problem early on, and thus also profit from the tight description.

6.2.1.2 General (not necessarily bipartite) matchings

Even if the graph $G = (V, E)$ with edge weights $w\colon E \to \mathbb{R}_{\geq 0}$ is not bipartite, we can still use the same integer program (6.2) to find a maximum weight matching in the bipartite case. Indeed, this works out because the set $Z \subseteq \{0, 1\}^E$ over which we optimize, and which is described in (6.4), corresponds to all characteristic vectors of matchings, even in non-bipartite graphs. (Given a finite set N and a subset U, then U's *characteristic* or *incidence* vector χ^U is the vector in $\{0, 1\}^N$ with $\chi^U_e = 1$ for $e \in U$ and $\chi^U_e = 0$ for $e \in N \setminus U$.)

However, contrary to the bipartite case, the natural relaxation of (6.2) is not tight anymore. Indeed, if G is a triangle with unit edge weights, then the linear relaxation given by (6.3) has an optimal objective value of 1.5, obtained by the vector x that assigns a value of 0.5 to each edge. This is clearly strictly more than the maximum cardinality of a matching in a triangle, which is only 1.

In the case of matchings, an inequality description of the integer hull P_I of all matchings is known:

$$P_I = \left\{ x \in \mathbb{R}^E_{\geq 0} \;\middle|\; \begin{array}{ll} x(\delta(v)) \leq 1 & \forall v \in V \\ x(E[S]) \leq \frac{|S|-1}{2} & \forall S \subseteq V, |S| \text{ odd} \end{array} \right\}, \tag{6.5}$$

where $E[S] \coloneqq \{\{u,v\} \in E \colon u,v \in S\}$. Note that it is of exponential size, and this can be shown to be necessary because the integer hull of all matchings, which is known as the *matching polytope*, can have exponentially many facets, and one needs at least one inequality per facet in the description due to Proposition 4.17. Of course, when attempting to solve a maximum weight matching problem, it is therefore typically not an option to explicitly list all constraints of (6.5) and use them in an IP or LP solver, as just the listing of the constraints would already take exponential time. To efficiently solve a linear program over P_I, one could use an approach known as the Ellipsoid Method, which generates constraints of (6.5) on the fly as needed in a clever way.

Similarly, one can also use the knowledge of the inequality description of P_I in an IP solver, in particular a branch & cut procedure. More precisely, instead of using a generic method to generate cutting planes, like the one we discussed in Chapter 5 to generate Chvátal-Gomory cuts, one can generate cuts corresponding to inequalities in (6.5). For this, we are interested in solving the following cut generating problem. Given a point $y \in \mathbb{R}^E$, we want to decide whether $y \in P_I$ and, if not, return a constraint of (6.5) that is violated by y (we also say *y-violated*). If y is an optimal solution to the relaxed problem in a branch & cut node, then such a constraint can be used as a cutting plane. Because P_I is the integer hull of the points we want to optimize over, we have that whenever $y \notin P_I$, then there is a constraint of (6.5) that is violated. Hence, there always is a cutting plane that can be generated in this way in a branch & cut method whenever needed.

Remark 6.1: Link to separation oracle

For the reader familiar with the Ellipsoid Method, we remark that the above-mentioned cut generating problem is very closely related to the so-called *separation problem* as encountered in the Ellipsoid Method. We recall that in the separation problem as used in the Ellipsoid Method, when applied to some polytope $P \subseteq \mathbb{R}^n$ and point $y \in \mathbb{R}^n$, one needs to decide whether $y \in P$ and, if not, the task is to return a normal vector $c \in \mathbb{R}^n$ such that $P \subseteq \{x \in \mathbb{R}^n : c^\top x < c^\top y\}$. Hence, when generating cutting planes by returning a y-violated constraint of (6.5) if there is one, we also solve the separation problem for P_I. Indeed, the normal vector c of the constraint we return is a solution to the separation problem as used in the Ellipsoid Method. More precisely, the separation problem for the Ellipsoid Method only requires a normal vector c that corresponds to some separating hyperplane. However, this normal vector does not need to be one corresponding to a constraint of (6.5), nor do we need to return the full separating hyperplane as we only need its normal vector in the Ellipsoid Method. Nevertheless, it is very common that when designing separation oracles for the Ellipsoid Method, one computes a violated inequality of some pre-defined description as the one shown in (6.5). Hence, in practice, the two problems are often handled very similarly.
Still, a key difference is that, for cut generation, we normally have to cut off a point y that

is a vertex solution of some explicit LP. In contrast, the Ellipsoid Method needs to be able to separate any point $y \in \mathbb{R}^n$ from P. This is a crucial difference that makes the task of finding a cutting plane for y much easier than designing a general separation oracle as used in the Ellipsoid Method. Recall that the procedure we saw to generate Chvátal-Gomory cuts heavily exploited this fact.

6.2.2 Stable sets

Maximum stable set problems are classical NP-hard graph optimization problems with an impressive number of applications. We start by the definition of a *stable set*, which is also called an *independent set*.

Definition 6.2: Stable set/independent set

Let $G = (V, E)$ be an undirected graph. A set $S \subseteq V$ is a *stable set* (or, equivalently, an *independent set*) if no edge has both endpoints in S.

Figure 6.2 shows an example of a stable set.

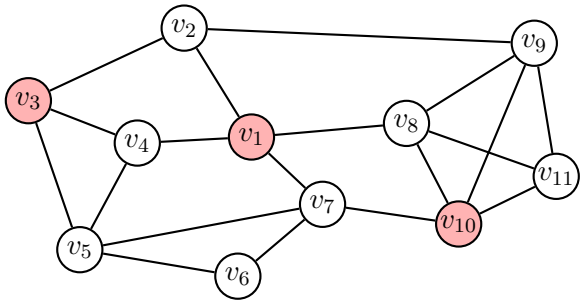


Figure 6.2: Example of a stable set.

The arguably two most canonical problems in this context are the maximum cardinality stable set problem and the maximum weight stable set problem, which generalizes the former.

Maximum cardinality stable set problem

Given an undirected graph $G = (V, E)$, find a stable set of maximum cardinality.

Maximum weight stable set problem

Given an undirected graph $G = (V, E)$ with non-negative vertex weights $w\colon V \to \mathbb{R}_{\geq 0}$, find a stable set $S \subseteq V$ of maximum weight $w(S) := \sum_{v \in S} w(v)$.

6.2.2.1 Canonical stable set formulation

The maximum weight stable set problem allows for the following straightforward formulation in terms of an integer program:

$$\max \left\{ w^\top x \colon x \in \{0,1\}^V \text{ with } x_u + x_v \leq 1 \; \forall \{u,v\} \in E \right\} \quad . \tag{6.6}$$

Unfortunately, the above formulation is often very weak, i.e., its canonical LP relaxation is quite loose. This is nicely exemplified by considering a maximum cardinality stable set problem (hence, w is the all-ones vector) on a graph G that is a clique on n vertices, i.e., $G = K_n$.[1] Clearly, a clique does not admit a stable set of cardinality larger than one. However, the canonical LP relaxation corresponding to (6.6), which is given by

$$\max \left\{ \mathbf{1}^\top x \colon x \in [0,1]^V \text{ with } x_u + x_v \leq 1 \; \forall \{u,v\} \in E \right\} \quad , \tag{6.7}$$

where $\mathbf{1} \in \mathbb{R}^V$ is the all-ones vector, has an optimal value of $n/2$ obtained by the vector x with all entries being equal to 0.5. Hence, the value of an optimal fractional (LP) solution is by a factor of $n/2$ higher then the value of an optimal integral (IP) solution. This factor of $n/2$ is also called the *integrality gap* of the relaxation (6.7). Large integrality gaps are very often an issue in branch & bound procedures, because it is very hard to prune nodes in the bounding step, as these rely on having found an integer solution with an objective value at least as good as the relaxation. Without a good pruning step, branch & bound procedures risk to resemble complete enumeration procedures.

As another example, consider the optimal value of the relaxation (6.7) applied to the graph given in Figure 6.2 (again considering the maximum cardinality stable set problem). One can observe that the optimal value of the relaxation is 5.5, again obtained by a uniform vector with value 0.5 everywhere. However, the largest stable set has only size 4 in this example, which can be achieved by adding v_6 to the stable set highlighted in Figure 6.2.

6.2.2.2 Clique inequalities

An IP formulation with a stronger canonical relaxation as the one shown in (6.6) is obtained by adding so-called *clique inequalities*. In this context, a clique inequality can be imposed for every clique of the given graph $G = (V, E)$. If $S \subseteq V$ are the vertices of a clique in G, then the *clique inequality* corresponding to S is given by

$$x(E[S]) \leq 1 \quad .$$

In words, the total x-value of edges in the clique must not exceed one. Or, combinatorially speaking, a stable set contains a most one vertex of each clique. Even though such clique constraints are implied by the canonical integer formulation (6.6), and therefore do not change the solution set of the integer program, they typically significantly strengthen the natural relaxation (6.7). This can easily be seen by considering again a maximum cardinality stable set problem on the clique K_n. Here, if we impose the clique inequality, then the relaxation is equal to the integer

[1] The clique on n vertices, which is also denoted by K_n, is the simple undirected graph on n vertices where every pair of vertices is connected by an edge. Hence, such a graph has $\binom{n}{2}$ many edges.

hull; in particular, its optimal value is 1 instead of the much weaker value of $n/2$ that we obtained through (6.7).

Also for the example shown in Figure 6.2, the clique inequalities lead to a significantly stronger relaxation. For this consider the *clique formulation*, which contains a clique inequality for each maximal clique.[2] Here, the maximal cliques of size strictly larger than 2 (notice that size-2 cliques simply correspond to edges) are

$$\{v_3, v_4, v_5\}, \{v_5, v_6, v_7\}, \text{ and } \{v_8, v_9, v_{10}, v_{11}\} \ .$$

Hence, by adding the clique inequalities corresponding to these cliques to (6.6), we obtain the following IP for the maximum weight stable set problem for the example shown in Figure 6.2:

$$\max \left\{ w^\top x \;\middle|\; \begin{array}{rcll} x_u + x_v & \leq & 1 & \forall \{u,v\} \in E \\ x_{v_3} + x_{v_4} + x_{v_5} & \leq & 1 & \\ x_{v_5} + x_{v_6} + x_{v_7} & \leq & 1 & \\ x_{v_8} + x_{v_9} + x_{v_{10}} + x_{v_{11}} & \leq & 1 & \\ x & \in & \{0,1\}^V & \end{array} \right\} \ . \tag{6.8}$$

We call the above IP the *clique formulation of stable set*. Notice that some of the constraints in the first constraint family are dominated by later constraints. For example, due to non-negativity of the variables, the constraint $x_{v_3} + x_{v_4} \leq 1$ is implied (even if x is not required to be integral) by the constraint $x_{v_3} + x_{v_4} + x_{v_5} \leq 1$. Hence, we could remove all such constraints from the description. Indeed, the name *clique formulation* is also used for the description that contains one constraint for each maximal clique, including cliques of size 2. This avoids the redundant constraints as explained above (but is sometimes more cumbersome to write down). Because both versions are equivalent even in terms of their relaxations, we call both of them *clique formulations*.

One can observe that for the maximum cardinality stable set problem (i.e., $w = \mathbf{1}$) on the example in Figure 6.2, the clique formulation, which is shown in (6.6), leads to a relaxation with optimal value 4, which equals the cardinality of a maximum cardinality stable set in the example. It is clear that the value of the relaxation is at least 4, because, as previously discussed, there is a stable set of cardinality 4. To see that it is no more than 4, one can observe that the vertices V of the given graph $G = (V, E)$ can be partitioned into 4 cliques, namely

$$\{v_1, v_2\}, \{v_3, v_4, v_5\}, \{v_6, v_7\}, \text{ and } \{v_8, v_9, v_{10}, v_{11}\} \ .$$

Any point $y \in [0,1]^V$ that is feasible for the natural relaxation of (6.8) thus fulfills

$$\begin{aligned} y_{v_1} + y_{v_2} &\leq 1, \\ y_{v_3} + y_{v_4} + y_{v_5} &\leq 1, \\ y_{v_6} + y_{v_7} &\leq 1, \text{ and} \\ y_{v_8} + y_{v_9} + y_{v_{10}} + y_{v_{11}} &\leq 1 \ . \end{aligned}$$

By summing up all four constraints above we obtain $y(V) \leq 4$, as desired.

[2] A clique $G[S]$ for $S \subseteq V$ is *maximal* if for any $v \in V \setminus S$, the graph $G[S \cup \{v\}]$ is not a clique.

Exercise 6.3: Integrality gap for maximum weight stable set

Even though the above observation shows that, for the maximum cardinality stable set problem on the graph shown in Figure 6.2, the natural relaxation of (6.8) has the same objective value as the IP, this does not imply that the relaxation describes the integer hull of all IP solutions. It only implies that there is no gap for the particular objective we considered, which is the all-ones vector in this case. Show that there are non-negative vertex weights for the instance shown in Figure 6.2 such that there is an integrality gap, i.e., the objective of the natural relaxation of (6.8) is strictly larger than the weight of a maximum weight stable set.

One obvious question, when planning to add clique inequalities for stable set problems, is how to find non-trivial cliques from which to create clique inequalities. Depending on the problem it is sometimes easy to identify all maximal cliques. We discuss a well-known example where this is the case in Section 6.2.3, namely interval packing. In other cases, it can make sense to use a heuristic to generate some clique inequalities. Finally, as a comment for the interested reader, it turns out that semidefinite programming techniques allow for defining a (convex but not necessarily linear) relaxation that fulfills all clique inequalities. However, this approach is rarely used in practice, mainly because semidefinite programs (which would correspond in this case to the relaxed problems in our branch & bound approach) typically take significantly longer to (approximately) solve than linear programs.

6.2.3 Interval packings

Interval packing problems often appear when different tasks have to be selected, each running during a predefined interval of time, and at any moment in time no two selected tasks run simultaneously. We start by formally defining one of the most canonical versions of weighted interval packing, which we call *simple weighted interval packing*.

Simple weighted interval packing problem

Given are n (half-open) intervals $[a_i, b_i)$ for $i \in [n]$ with $a_i, b_i \in \mathbb{Z}$ and $a_i < b_i$. Moreover, each interval $i \in [n]$ has a positive weight $w_i \in \mathbb{R}_{>0}$. The task is to find a subset $I \subseteq [n]$ of non-overlapping intervals, i.e., $[a_i, b_i) \cap [a_j, b_j) = \emptyset$ for $i, j \in I$ with $i \neq j$, of maximum weight $w(I) := \sum_{i \in I} w_i$.

Figure 6.3 shows an example instance of the simple interval packing problem and highlights one optimal solution for this instance.

The simple interval packing problem can readily be cast as a maximum weight stable set problem. To this end, we introduce one vertex per interval with weight equal to the weight of the interval, and connect two vertices by an edge if the corresponding intervals overlap. The maximum weight stable set problems resulting from simple interval packing problems are very structured. In particular, they have at most linearly many maximal cliques and these cliques can easily be identified given the intervals. Indeed, the maximal cliques are a subset of the cliques obtained as follows: For each starting point a_i of an interval $i \in I$, consider all intervals that

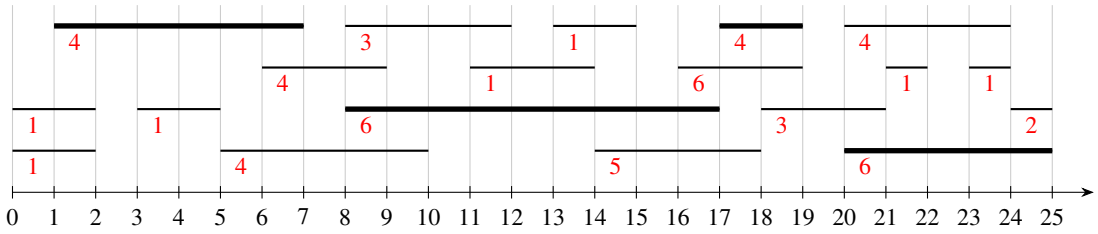


Figure 6.3: Example instance of a simple weighted interval packing problem. The red numbers below each interval indicate the weight of the corresponding interval. The four intervals highlighted as thick black lines are an optimal solution to the problem.

contain a_i. It turns out that the resulting description is tightest possible as its relaxation is the convex hull of all integer solutions.

There are many variations of interval packing problems. For example, instead of forbidding overlaps completely, one can allow that up to a certain number of intervals can overlap at any point. This generalization can capture settings where multiple resources are available to run tasks at the same time.

Another interesting generalization are circular interval packing problems. Here, the intervals are a consecutive subset of the circumference of a circle, and for any angle there can be at most one selected interval covering that angle. Figure 6.4 shows an example instance of this generalization of the simple weighted interval packing problem, together with an optimal selection of intervals. Circular interval packing problems often appear when having to schedule repeating tasks. For example if a schedule has to be created that repeats every day.

Analogously to the non-circular case, one can cast the problem as a maximum weight stable set problem.[3] Whereas in interval graphs, every clique can be described as all intervals containing a fixed point, there are further types of cliques in the case of circular interval graphs that exploit the circularity. More precisely, cliques can consist of intervals that connect to each other around the full cycle. One such example would be a triangle obtained by having one interval from angle 0 to 130, a second one from angle 120 to 250, and a third one from angle 240 to angle 10. Such cliques that exploit the circularity change the nature of the maximal cliques considerably. In particular, contrary to the interval case, in circular interval graphs it is possible to have exponentially many maximal cliques.

Exercise 6.4: Number of maximal cliques in circular interval graph

Show that circular interval graphs can have exponentially many maximal cliques. More precisely, show that there is a constant $c > 0$ such that, for any $n \in \mathbb{Z}_{\geq 2}$, there exists a circular interval graph with n intervals and at least 2^{cn} many maximal cliques.

[3] The corresponding graph, which has a vertex for every circular interval and an edge connecting two intervals if they are overlapping, are called *circular arc graphs*. In the literature, the notion *circular interval graph* is often used for a circular arc graph coming from a family of intervals where no interval contains another one.

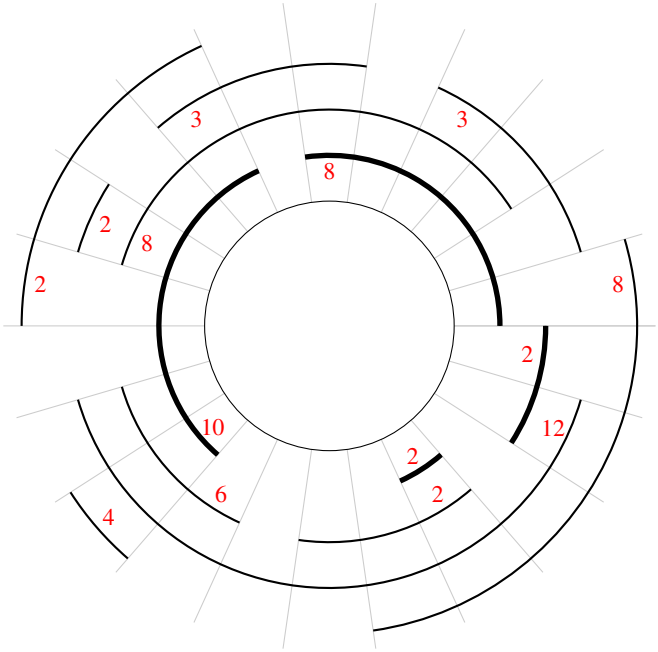


Figure 6.4: Example instance of a simple circular interval packing problem. The circular intervals highlighted as thick black arcs are a subset of non-overlapping intervals of maximum total weight. As with the non-circular version, we assume that all integrals are half-open.

Another key difference to the interval graph case is that the maximal clique formulation, which may be of exponential size due to Exercise 6.4, does in general not describe the integer hull of all interval packings.

Exercise 6.5: Non-tightness of clique formulation for circular interval packing

Show that the relaxation of the clique formulation for the simple circular interval packing problem does not always describe the integer hull.

Moreover, we observe that, depending on the instance, the circular case can sometimes be reduced to few instances of the non-circular case.

Remark 6.6: Reducing circular interval packing to regular interval packing

Sometimes the simple interval packing problem can easily be reduced to the interval packing problem through a limited guessing step. For this, we can identify an angle at which there are only few circular intervals. In the example shown in Figure 6.4 one could pick an angle that is contained in only a single circular interval, for example the one of weight 10. Through a branching operation on that interval, we can split the original problem into two subproblems:

(i) One in which the weight-10 interval is not selected, and
(ii) one in which we select the weight-10 interval.

A crucial observation is that both of these subproblems reduce to regular (non-circular) simple interval packing problems. For case (i) this is immediate because there is an angle not covered by any of the remaining intervals, which readily leads to a reduction to the non-circular case by cutting the cycle at this angle. For case (ii), one obtains a residual instance where we can delete any circular interval that overlaps with the weight-10 interval that we already selected, which also leads to a non-circular simple interval packing problem.

The above remark also hints toward another way to optimize a branch & bound method when dealing with circular interval packing problems. First, as already discussed, it is typically advantageous to use the clique formulation in this case. Moreover, one can identify an angle covered by as few intervals as possible and require the branching procedure to first branch on variables that correspond to the intervals covering this angle. This way, we are sure that when the branching procedure branched on all of these variables, the remaining relaxation will be tight and no further branchings are necessary.

6.2.4 Uncapacitated lot-sizing

Consider the following production planning problem of a single (arbitrarily splittable) good, which is also known as the uncapacitated lot-sizing problem. There are $T \in \mathbb{Z}_{>0}$ discrete time steps. We assume not to have any initial stock. For each time step $t \in [T]$ there is a demand of $d_t \in \mathbb{R}_{\geq 0}$ that must be satisfied. Producing a unit of the good at time step $t \in [T]$ incurs a cost of p_t. Moreover, if a non-zero quantity of goods is produced at some time step $t \in [T]$, then a setup cost of $q_t \in \mathbb{R}_{\geq 0}$ needs to be paid, no matter how large the strictly positive production quantity at time step t is. For each time step $t \in [T]$, the stock at the end of the time step is the sum of all produced goods up to and including time step t minus the sum of all demands up to and including time step t. A storage cost of h_t is incurred per unit of stock at the end of each time step $t \in [T]$. Backlogs are not allowed, i.e., the stock must be non-negative at each time step. The goal is to satisfy all of the demands at minimum total cost.

Below is a mixed-integer program that models the uncapacitated lot-sizing problem, where M is a sufficiently large fixed value. In particular, it suffices to set $M := \sum_{t=1}^{T} d_t$ to obtain a correct model.

$$\begin{aligned} \min \quad & \sum_{t=1}^{T} p_t x_t + \sum_{t=0}^{T} h_t \left(\sum_{\ell=1}^{t} x_\ell - \sum_{\ell=1}^{t} d_\ell \right) + \sum_{t=1}^{T} q_t y_t \\ & \sum_{\ell=1}^{t} x_\ell - \sum_{\ell=1}^{t} d_\ell \geq 0 \qquad \forall\, t \in [T] \\ & x_t \leq M \cdot y_t \qquad \forall\, t \in [T] \\ & x \in \mathbb{R}^T_{\geq 0} \\ & y \in \{0,1\}^T \end{aligned} \tag{6.9}$$

The above mixed-integer linear program (in short MIP), has two types of variables, namely

x and y. For each $t \in [T]$, the continuous variable x_t is the production quantity at time step t, and y_t is an indicator of whether we produce at time step t. More precisely, $y_t = 0$ represents that we are not producing anything at time step t, where $y_t = 1$ implies that we are allowed to produce at time step t. In other words, one can think of $y_t = 1$ as the decision of whether to pay the setup cost at time step t.

The above formulation uses a constraint type generally denoted as *big-M constraint*, namely the constraints $x_t \leq M \cdot y_t$ for $t \in [T]$. The idea of a big-M constraint is to use a binary variable, here y_t, to act as an on/off switch for another expression, here the variable x_t. Indeed, if $y_t = 0$, then x_t is forced to be zero because the big-M constraint imposes $x_t \leq 0$ which, together with the non-negativity constraint on x_t, implies $x_t = 0$. On the other hand, if $y_t = 1$, then the constraint $x_t \leq M \cdot y_t$ does not impose any relevant restriction on the value of x_t because M was chosen to be very large; more precisely at least as large as the largest reasonable value for x_t. (This is why $M := \sum_{t=1}^{T} d_t$ is large enough, because a production quantity of $\sum_{t=1}^{T} d_t$ allows for satisfying all demands. Thus there never is a need to produce at any time step more than M goods.) Hence, the constraint $x_t \leq M \cdot y_t$ captures that we are only allowed to produce at time step t if we pay the setup cost.

Moreover, note that in the first family of inequalities, the expression $\sum_{\ell=1}^{t} x_\ell - \sum_{\ell=1}^{t} d_\ell$ is the stock at the end of time step t. Hence, these inequalities enforce that the stock is non-negative at any time step. Sometimes it is convenient to introduce auxiliary variables to simplify or even strengthen the description of a model. For example, the formulation (6.9) can be equivalently rewritten in an arguably simpler form by introducing, for $t \in [T]$, a variable s_t that represents the stock at time t, and we let $s_0 = 0$, which represents that there is no initial stock. This leads to the following formulation.

$$\begin{aligned} \min \quad & \sum_{t=1}^{T} p_t x_t + \sum_{t=0}^{T} h_t s_t + \sum_{t=1}^{T} q_t y_t \\ & s_t = s_{t-1} + x_t - d_t \quad \forall\, t \in [T] \\ & x_t \leq M \cdot y_t \quad \forall\, t \in [T] \\ & x \in \mathbb{R}^T_{\geq 0} \\ & s \in \mathbb{R}^T_{\geq 0} \\ & y \in \{0,1\}^T \end{aligned} \tag{6.10}$$

Notice that the vector of stock variables $s \in \mathbb{R}^T_{\geq 0}$ only has T entries and not $T+1$ as it does not include s_0, because s_0 is the initial stock, which is given and therefore not a variable.

Remark 6.7: Potential impact of auxiliary variables on branch & bound procedures

Our motivation for introduction the variables s_t was merely for convenience. In particular, the introduction of these variables does not change the description in any relevant way as they can be substituted back using the equality constraints in (6.10). Nevertheless, in some cases, the introduction of such auxiliary variables may change the behavior when using branch & bound procedures. For example, if the production quantities were required to be integral, then also the stocks s_t are integral, and a branch & bound procedure may decide to branch on

one of the newly introduced integer variables s_t. However, whether such a potential change in the behavior of some branch & bound procedures is advantageous heavily depends on the problem at hand. Moreover, if there is good reason to believe that some auxiliary variables make good candidates for branching, then one may want to explicitly instruct the branch & bound procedure to branch on these variables first. Also, in this case, it is typically not even necessary to introduce a new auxiliary variable, because one can branch on a linear expression that corresponds to the auxiliary variable, without actually introducing an auxiliary variable (see discussion in Chapter 5).

We highlight that the variables x and y used in (6.9) are often called *decision variables* as they correspond to decisions we have to take (paying the setup cost and how much to produce at each time step). As mentioned, the introduction of auxiliary variables s_t was primarily for convenience in this case. However, as we will discuss in Chapter 7, it is sometimes advantageous to formulate a description in a higher dimensional space than what one would obtain by only using natural decision variables, as this can allow for obtaining tighter polynomial-size formulations. To achieve this goal, we will introduce additional/auxiliary variables that cannot be expressed as a linear combination of the original (decision) variables. This is a key difference to the auxiliary variables s_t that we introduced to obtain the formulation (6.10), and opens up very interesting and sometimes surprising new possibilities in terms of IP/LP formulations.

Remark 6.8: Crucial weakness of big-M constraints

Big-M constraints are a powerful tool to model a variety of problems. However, they typically lead to extremely loose linear relaxations, resulting in high running times for branch & cut procedures. Indeed, consider the natural linear relaxation of (6.10), which is obtained by replacing $y \in \{0,1\}^T$ by $y \in [0,1]^T$. If M is very large, then a small fractional value of y_t is typically enough to make sure that the constraint $x_t \leq M \cdot y_t$ is fulfilled for relevant values of x_t. Hence, the linear relaxation will typically only pay a small fractional of the setup costs than what one would actually need to pay.
Due to this, formulations with big-M constraints often require branch & bound or branch & cuts procedures to enumerate/branch over a very large fraction of binary variables involved in big-M constraints. This goes against one of the key advantages of branch & bound/cut procedures, namely that we normally only have to explore a small fractional of the whole solution space due to the bounding. Because of this, it is a good guideline to avoid big-M constraints whenever possible, except for very small problems where running time is of little importance.

7 Descriptions in Extended Space

In many settings, there are advantages to describing the solutions to integer or linear programming problems not in the natural original space of the problem, but in a higher-dimensional space, also known as extended space. Moving to a higher-dimensional space can often be interpreted as introducing auxiliary variables that are not strictly necessary for describing all possible solutions of a given problem, but can be very useful in constructing strong linear constraints. In this chapter, we introduce basics about descriptions in extended space and show how they can lead to significantly stronger and more compact formulations. For example, in some cases, they allow us to avoid big-M constraints in a convenient way, or they can be used to obtain a polynomial-sized description of a problem that otherwise would need exponentially many inequalities.

7.1 Introduction

For simplicity, we start by discussing the concept of an extended formulation in the context of linear programming. Here, the feasible regions over which we want to optimize are polyhedra. Hence, let $P \subseteq \mathbb{R}^n$ be a polyhedron. Later, when we talk about integer programs, we will discuss formulations of different strengths in the context of descriptions in extended space. However, to introduce basic concepts of extended formulations, we first focus on different ways to describe the polyhedron P exactly. So far, we focused on inequality descriptions of P in original space, i.e., in $\mathbb{R}^n$, like

$$P = \{x \in \mathbb{R}^n \colon Ax \leq b\} \quad ,$$

for some matrix $A \in \mathbb{R}^{m \times n}$ and vector $b \in \mathbb{R}^m$, where $m \in \mathbb{Z}_{\geq 0}$. A description of P in extended space is a linear inequality/equality description of a polyhedron Q, such that P is an axis-parallel projection of Q. Formally, such an inequality description of Q can be written as

$$Q = \{(x, y) \in \mathbb{R}^n \times \mathbb{R}^p \colon Cx + Dy \leq f\} \quad , \tag{7.1}$$

and fulfills

$$P = \mathrm{proj}_x(Q) \coloneqq \{x \colon (x, y) \in Q\} = \{x \in \mathbb{R}^n \colon \exists y \in \mathbb{R}^p \text{ with } Cx + Dy \leq f\} \quad ,$$

where $\mathrm{proj}_x \colon \mathbb{R}^{n+p} \to \mathbb{R}^n$ is the projection onto the first n coordinates, i.e., $\mathrm{proj}_x((x, y)) = x$ for $(x, y) \in \mathbb{R}^n \times \mathbb{R}^p$, and $C \in \mathbb{R}^{\ell \times n}, D \in \mathbb{R}^{\ell \times p}$, and $f \in \mathbb{R}^\ell$ for some $\ell \in \mathbb{Z}_{\geq 0}$. Such a description, as shown in (7.1), is called a *linear extended formulation* of P, or simply an *extended formulation* of P. Analogous to linear programs, such formulations are also allowed to contain equality or greater-than-or-equal inequalities, which, as in the context of LPs, can always be reduced to the form shown in (7.1). Nevertheless, just as with LPs, it is often convenient

to allow for equality constraints and both types of inequality constraints. The number ℓ of inequalities in (7.1) is called the *size* of the extended formulation. When we have a description with a mix of inequalities and equalities, then only the inequalities are counted toward the size of the formulation.[1]

Notice that any linear program over P, say,

$$\begin{array}{lrcl} \max & c^\top x & & \\ & Ax & \leq & b \\ & x & \in & \mathbb{R}^n \ , \end{array}$$

can easily be rephrased as an equivalent problem over Q as follows:

$$\begin{array}{lrcrcl} \max & c^\top x & & & & \\ & Cx & + & Dy & \leq & f \\ & x & & & \in & \mathbb{R}^n \\ & & & y & \in & \mathbb{R}^p \ . \end{array}$$

Hence, an extended formulation is indeed also useful from an algorithmic point of view, as it allows for solving the original linear program.

Definition 7.1: (Linear) Extension complexity

The extension complexity of a polyhedron $P \subseteq \mathbb{R}^n$ is the smallest size of any linear extended formulation of P.

When dealing with mixed-integer or integer programs, some or all variables in the description of P are required to be integral. Analogous to the case of linear programs, an extended formulation is an inequality description of a set Q with the difference that some of its variables are required to be integral. Moreover, we still want to have the property that the projection of Q onto the x-space is equal to P. In this more general context, apart from the size of the extended formulation—i.e., the number of linear inequalities used—also the number of integer variables used is an important parameter. Because the number of integer variables used can crucially impact how large the size of an extended formulation needs to be, we do not use the notion of extension complexity in the context when integer variables are present. Nevertheless, the notion of linear extension complexity as described in Definition 7.1 is also relevant in this context. In integer programming, for example, we often deal with linear programs because branch & cut procedures heavily rely on linear relaxations.

An obvious question is what benefits there are in describing a solution set in extended space, and then doing optimization in this larger-dimensional space. One interesting property is that, sometimes, one can obtain significantly smaller descriptions in extended space compared to what is possible in original space. To provide some intuition for this, Figure 7.1 shows an example of a 2-dimensional polytope P that needs 8 linear inequalities when described in original space but only 6 when described as an extended formulation in $\mathbb{R}^3$.

[1] Equalities are not counted because they could be removed by using them to eliminate variables until no equality constraint is left, leading to an equivalent problem. This helps to understand why the *size* of an extended formu-

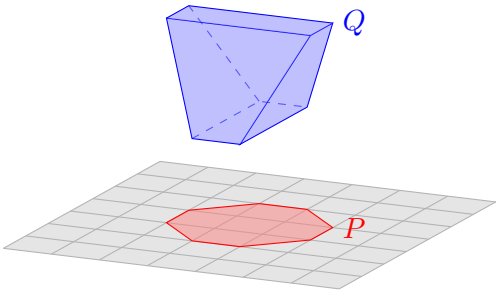


Figure 7.1: An example of a polytope P in $\mathbb{R}^2$, i.e., $n = 2$, for which 8 linear inequalities are necessary to describe it in the original space $\mathbb{R}^2$. However, it admits an extended formulation in $\mathbb{R}^3$, highlighted by the polytope Q, where 6 linear inequalities suffice.

The example shown in Figure 7.1 is just an illustrative toy example, where the difference in terms of used constraints between original and extended space is very small. (Moreover, depending on the case, the number of bits needed to represent a constraint in extended space can be larger than in original space.) However, as we will see in later examples, there are cases where the original polytope P has exponentially many (in the dimension) facets, and therefore needs exponentially many linear constraints to be described in original space due to Proposition 4.17, and there exists an extended formulation whose bit size is only polynomial in the dimension.

This is one key reason why extended formulations are used in integer programming. However, this is certainly not the only reason. Sometimes, additionally introduced variables are well-suited for branching operations in a branch & cut algorithm, or they allow for a more concise description of cutting planes. Also, very often, auxiliary variables can lead to a more natural and clear description. We have seen such a case in Section 6.2.4, when talking about the uncapacitated lot-sizing problem. In this context, we first considered the mixed-integer formulation (6.9), which we then simplified to the arguably more elegant and easier-to-parse description (6.10) by introducing variables s_t representing the stock at time $t \in [T]$.

7.2 Examples

We now exemplify how descriptions in extended space can be used to obtain strong IP formulations for a variety of problems. In particular, we also show cases in which a well-chosen extended formulation leads to a strong polynomial-sized description circumventing the use of big-M constraints that show up in some natural descriptions in original space.

lation is defined this way. However, we typically do not eliminate equality constraints explicitly as this would lead to less intuitive formulations with a constraint matrix that may also be less sparse.

7.2.1 Matchable vertices in bipartite graphs

We start with the following optimization problem that has a natural description in extended space. For a graph $G = (V, E)$, we call a vertex set $S \subseteq V$ *matchable* if there exists a matching $M \subseteq E$ such that no vertex in S is exposed by M.

Maximum weight matchable set of size at most k

Input: An undirected graph $G = (V, E)$ with vertex weights $w\colon V \to \mathbb{R}_{\geq 0}$ and $k \in \mathbb{Z}_{>0}$.
Task: Find a maximum weight set $S \subseteq V$ of matchable vertices of size at most k.

Note that the solution space of this problem is the family of all subsets of vertices that are *matchable*. Hence, the natural space for a description of this solution set is $\{0,1\}^V$, with one variable x_v per vertex $v \in V$, using the usual interpretation that $x_v = 1$ indicates $v \in S$. The constraint that at most k vertices can be picked is easily captured by the linear constraint $x(V) \leq k$.

It remains to make sure that the chosen vertices indeed correspond to a matchable set. To this end, it is natural to move to an extended space and introduce edge variables $y \in \{0,1\}^E$ to define a matching that makes sure that the set described by the x-variables is matchable. This leads to the following natural formulation in extended space $\mathbb{R}^{V \times E}$.

$$\begin{aligned} \max \quad & w^\top x & & & \\ & y(\delta(v)) & \leq & 1 & \forall v \in V \\ & x(v) & \leq & y(\delta(v)) & \forall v \in V \\ & x(V) & \leq & k & \\ & x & \in & \{0,1\}^V & \\ & y & \in & \{0,1\}^E & \end{aligned} \tag{7.2}$$

Exercise 7.2

Consider the problem of finding a maximum weight matchable set of size at most k restricted to bipartite graphs. Show that in this case, the optimal value of the natural linear relaxation of (7.2) is equal to the value of an optimal integer solution.
It turns out that showing this statement is equivalent to showing that the projection of the natural linear relaxation of (7.2) to the space of the x-variables describes the convex hull of all integer solutions in the x-space. Hence, the projection of the relaxation is tightest possible.

7.2.2 Uncapacitated lot-sizing

A crucial issue with the mixed-integer formulation (6.10) that we presented for the uncapacitated lot-sizing problem is the use of big-M constraints, which lead to very weak linear relaxations. We now present a significantly stronger (actually even tight) formulation in extended space. To

this end, we introduce a new family of variables. More precisely, for all $t, \ell \in [T]$ with $t \leq \ell$, we introduce a variable $z_{t,\ell} \in [0,1]$, which represents the fraction of the demand at time ℓ that is covered by production at time t. Hence, these variables carry fine-grained information about when produced goods are used, which one cannot derive from only the overall production quantity x_t at time t. Notice that this interpretation of the z-variables naturally leads to the following relation between them and the x-variables.

$$x_t = \sum_{\ell=t}^{T} d_\ell z_{t,\ell} \quad \forall\, t \in [T] \ .$$

In words, the total production quantity x_t at time t is the sum over the current and all future time steps ℓ of the production quantities $d_\ell z_{t,\ell}$ at time t for the demand at time ℓ. Moreover, because each demand needs to be satisfied, we have

$$\sum_{t=1}^{\ell} z_{t,\ell} = 1 \quad \forall\, \ell \in [T] \ .$$

Using these new variables and the above relations, we create the following stronger mixed-integer formulation of the uncapacitated lot-sizing problem.

$$\begin{aligned}
\min \quad & \sum_{t=1}^{T} p_t x_t + \sum_{t=0}^{T} h_t s_t + \sum_{t=1}^{T} q_t y_t & & \\
& z_{t,\ell} \leq y_t && \forall\, t, \ell \in [T] \text{ with } t \leq \ell \\
& x_t = \sum_{\ell=t}^{T} d_\ell z_{t,\ell} && \forall\, t \in [T] \\
& \sum_{t=1}^{\ell} z_{t,\ell} = 1 && \forall\, \ell \in [T] \qquad (7.3) \\
& s_t = s_{t-1} + x_t - d_t && \forall\, t \in [T] \\
& x \in \mathbb{R}^T_{\geq 0} & & \\
& s \in \mathbb{R}^T_{\geq 0} & & \\
& y \in \{0,1\}^T & & \\
& z_{t,\ell} \in [0,1] && \forall t, \ell \in [T] \text{ with } t \leq \ell
\end{aligned}$$

For the above formulation to be correct, there is an important technicality to be considered. Namely, the above formulation is incorrect if $d_1 = 0$. In this case, the formulation (7.3) requires $z_{1,1} = 1$ and therefore $y_1 = 1$. Hence, we need to pay the setup cost q_1, even though it may be best not to produce during the first time step because the first demand is zero. There are several ways to fix this problem. One is to only use variables $z_{t,\ell}$ for indices $t, \ell \in [T]$ with $t \leq \ell$ and $d_\ell > 0$. This leads to a slight variation of the formulation (7.3) where all z-variables $z_{t,\ell}$ with $d_\ell = 0$ and constraints involving such variables are removed. For simplicity of exposition, we assume $d_1 > 0$, in which case any solution needs to pay the first setup cost y_1, to satisfy the first

demand, and the problem does not appear. Note that the same problem does not reappear for later time steps $\ell \in [T]$ with zero demand, i.e., $d_\ell = 0$. Indeed, in this case we can simply set $z_{1,\ell} = 1$ without having to pay additional unnecessary setup costs.

To discuss the strength of the formulation (7.3), and compare it to the strength of our prior big-M formulation (6.10), we consider the corresponding linear relaxations.

Let $Q_1 \subseteq \mathbb{R}^T \times \mathbb{R}^T \times \mathbb{R}^T$ be the feasible solutions to the natural linear relaxation corresponding to (6.10), i.e.,

$$Q_1 := \left\{ (x,y,s) \in \mathbb{R}^T_{\geq 0} \times [0,1]^T \times \mathbb{R}^T_{\geq 0} \colon \begin{array}{rcll} s_t & = & s_{t-1} + x_t - d_t & \forall\, t \in [T] \\ x_t & \leq & M \cdot y_t & \forall\, t \in [T] \end{array} \right\} ,$$

where $M := \sum_{t=1}^T d_t$.

Analogously, we denote by $Q_2 \subseteq \mathbb{R}^T \times \mathbb{R}^T \times \mathbb{R}^T \times \mathbb{R}^Z$, where $Z := \{(t,\ell) \in T \times T \colon t \leq \ell\}$, the feasible solutions of the natural linear relaxation corresponding to (7.3), i.e.,

$$Q_2 := \left\{ \begin{array}{l} (x,y,s,z) \in \mathbb{R}^T_{\geq 0} \times [0,1]^T \times \mathbb{R}^T_{\geq 0} \times [0,1]^Z \colon \\ \qquad \begin{array}{rll} z_{t,\ell} \leq y_t & & \forall\, (t,\ell) \in Z \\ x_t = \displaystyle\sum_{\ell=t}^{T} d_\ell z_{t,\ell} & & \forall\, t \in [T] \\ \displaystyle\sum_{t=1}^{\ell} z_{t,\ell} = 1 & & \forall\, \ell \in [T] \\ s_t = s_{t-1} + x_t - d_t & & \forall\, t \in [T] \end{array} \end{array} \right\} .$$

Finally, we denote by $Q_I \subseteq \mathbb{R}^T \times \{0,1\}^T \times \mathbb{R}^T$ all feasible solutions to the mixed-integer problem (6.10) (and not its relaxation), which corresponds to all actual solutions of the uncapacitated lot-sizing problem. Hence, the tightest linear relaxation of the problem (in the variables (x,y,s)) is $\mathrm{conv}(Q_I)$.

We start by observing that the formulation (7.3) is indeed at least as strong as (6.10), in terms of their relaxations.

Theorem 7.3

$\mathrm{proj}_{x,y,s}(Q_2) \subseteq Q_1$.

Proof. We have to show that any point $(\overline{x},\overline{y},\overline{s},\overline{z}) \in Q_2$ satisfies $(\overline{x},\overline{y},\overline{s}) \in Q_1$. Notice that in both descriptions, we have the requirement $(x,y,s) \in \mathbb{R}^T_{\geq 0} \times [0,1]^T \times \mathbb{R}^T_{\geq 0}$, and the constraints $s_t = s_{t-1} + x_t - d_t$ for $t \in [T]$. Hence, it remains to check that $\overline{x}_t \leq M \cdot \overline{y}_t$ for $t \in [T]$, which holds because

$$\overline{x}_t = \sum_{\ell=t}^{T} d_\ell \overline{z}_{t,\ell} \leq \sum_{\ell=t}^{T} d_\ell \overline{y}_t \leq M \overline{y}_t \ ,$$

where the used relations follow from the constraint families of Q_2 and from $M := \sum_{\ell=1}^T d_\ell$. □

Moreover, it is not hard to see that the relaxation corresponding to (7.3) can be significantly stronger than the one corresponding to (6.10). Indeed, it is possible to choose points in Q_1 such that some entry of y, say y_1, is quite small (think, e.g., of $y_1 = 0.1$), but still, one can fulfill the full demand of some later time steps by the production at time 1. This is a crucial issue of M-big constraints, which we previously discussed. This problem does not exist with Q_2.

It turns out that one can even show that Q_2 corresponds to a tightest linear programming relaxation of the problem. We state this property below without proof.

Theorem 7.4

$\text{proj}_{x,y,s}(Q_2) = \text{conv}(Q_I)$.

Exercise 7.5

Create a concrete instance of the uncapacitated lot-sizing problem showing that the linear relaxation corresponding to (6.10), which, we recall, is

$$\min \left\{ \sum_{t=1}^{T} p_t x_t + \sum_{t=0}^{T} h_t s_t + \sum_{t=1}^{T} q_t y_t \colon (x, y, s) \in Q_1 \right\} ,$$

can provide solutions that are significantly cheaper (and therefore weaker) than the relaxation corresponding to (7.3), which is

$$\min \left\{ \sum_{t=1}^{T} p_t x_t + \sum_{t=0}^{T} h_t s_t + \sum_{t=1}^{T} q_t y_t \colon (x, y, s, z) \in Q_2 \right\} .$$

To exemplify this difference, provide an instance where both relaxations have strictly positive optimal objective value, and the optimal value of the first relaxation is no more than half the one of the second relaxation.

7.2.3 Directed Steiner Tree

The Steiner Tree problem is a basic network connectivity problem, closely related to spanning trees. Whereas a spanning tree is required to connect all vertices of a graph $G = (V, E)$, a Steiner Tree only needs to connect a given subset $T \subseteq V$ of the vertices, which are also called *terminals*. For a set of terminals $T \subseteq V$, we call an edge set $U \subseteq E$ a *Steiner Tree (for T)* if $(V(U), U)$ is a spanning tree, where $V(U) \subseteq V$ are all vertices touched by at least one edge of U, and $T \subseteq V(U)$. We start with a formal definition of the Steiner Tree problem.

Minimum Steiner Tree problem

Input: An undirected graph $G = (V, E)$ with non-negative edge lengths $\ell \colon E \to \mathbb{R}_{\geq 0}$ and a set of terminals $T \subseteq V$.
Task: Find a Steiner Tree $U \subseteq E$ (for T) of smallest length $\ell(U)$.

Notice that for $T = V$, we recover the minimum spanning tree problem.

Depending on the literature, it is sometimes required that the leaf vertices of a Steiner Tree need to be terminals. Clearly, this does not change the problem because any Steiner Tree with a leaf vertex that is not a terminal can trivially be transformed into another Steiner Tree of no larger length by repeatedly removing edges incident to leaf vertices that are not terminals. Even though we do not have this requirement in our definition, when providing examples of Steiner Trees, we typically choose them such that all leaf vertices are terminals.

Steiner Tree problems and variations thereof appear in a broad set of applications where selected locations need to be connected in a cheapest possible way, like for example in placement problems appearing in the context of very large-scale integration (VLSI), which is a well-known problem in chip design.

The directed Steiner Tree problem is a natural extension of the Steiner Tree problem to directed graphs $G = (V, A)$. The difference to the (undirected) Steiner Tree problem is that one terminal $r \in T$ has a special role by being designated as a root, and the task is to find an arc set such that the root can reach all other terminals. Hence, a *directed Steiner Tree* is an arc set $U \subseteq A$ such that $(V(U), U)$ is a spanning tree when disregarding arc directions, and each terminal can be reached from r in the graph $(V(U), U)$. We formally define the problem below.

Minimum directed Steiner Tree problem

Input: A directed graph $G = (V, A)$ with non-negative arc lengths $\ell\colon A \to \mathbb{R}_{\geq 0}$, a set of terminals $T \subseteq V$, and a root $r \in T$.
Task: Find a directed Steiner Tree $U \subseteq A$ (for T) of smallest length $\ell(A)$.

The special case $T = V$ leads to the minimum r-arborescence problem. An *r-arborescence* in a directed graph $G = (V, A)$ with $r \in V$ is an arc set $U \subseteq A$ such that (i) when disregarding arc directions, U is a spanning tree, and (ii) all arcs of U are directed away from r; in other words, there is a unique path from r to any other vertex in (V, U). Thus, the condition of $U \subseteq A$ being a directed Steiner Tree can be rephrased as $(V(U), U)$ being an r-arborescence in $G[V(U)]$.

Notice that the (undirected) Steiner Tree problem can easily be reduced to its directed version by bidirecting all edges and assigning to both arcs obtained through the bidirection of an edge e the length of the edge e. The Steiner Tree problem is well-known to be NP-hard, which, due to the highlighted reduction, also applies to its directed version.

In the following, we discuss three formulations for the directed Steiner Tree problem.

7.2.3.1 Flow formulation based on big-M constraints

A crucial aspect in finding a formulation of the Steiner Tree problem (and other connectivity-related problems) is to make sure to add constraints that ensure connectivity between the different terminals. One option to do this is by using flows. More precisely, given a directed Steiner Tree $U \subseteq A$, if we think of installing capacities of $|T| - 1$ on each arc of U, then it is possible to simultaneously send one unit of flow from r to each other terminal. Conversely, by requiring that the selected arcs U allow for sending such a flow, we make sure that all terminals are in the

same connected component of the selected arcs, and arcs are also directed appropriately such that there is a path from the root to each of the terminals. These conditions do not strictly imply that the selected arcs U are such that $(V(U), U)$ forms a spanning tree, but they make sure that U contains a Steiner Tree. Because we are minimizing a non-negative length function, there is no benefit in having superfluous arcs, and the problems are thus identical. We now leverage this idea to obtain the following mixed-integer programming formulation of the directed Steiner Tree problem.

$$\begin{aligned}
&\min \sum_{a \in A} \ell(a) x(a) \\
f(\delta^+(v)) - f(\delta^-(v)) &= \begin{cases} |T| - 1 & \text{if } v = r \\ -1 & \text{if } v \in T \setminus \{r\} \\ 0 & \text{if } v \in V \setminus T \end{cases} \quad \forall\, v \in V \\
f(a) &\leq (|T| - 1) x(a) \quad \forall\, a \in A \\
x &\in \{0,1\}^A \\
f &\in \mathbb{R}^A_{\geq 0}
\end{aligned} \tag{flow MIP 1}$$

Note that (flow MIP 1) is a mixed-integer formulation of the directed Steiner Tree problem in extended space. Indeed, the description is in an extended space because apart from the variables x, which correspond to the canonical space of the problem, we use additional flow variables $f \in \mathbb{R}^A_{\geq 0}$. Unfortunately, the linear relaxation of this formulation is quite weak because the constraints $f(a) \leq (|T| - 1)x(a)$ are essentially big-M constraints, where one can think of $|T| - 1$ as being M.

Exercise 7.6

Show that for any $k \in \mathbb{Z}_{>0}$, there is an instance of the directed Steiner Tree problem such that the optimal value of the natural linear relaxation of (flow MIP 1) is by a factor of $|T| - 1$ smaller than the length of a shortest directed Steiner Tree.

7.2.3.2 Flow formulation avoiding big-M constraints

The reason why we obtained big-M constraints in (flow MIP 1) is that we needed a large factor of $|T|-1$ in front of $x(a)$ in the constraints $f(a) \leq (|T|-1)x(a)$, because it may be that the path in the Steiner Tree from r to every other terminal $u \in T \setminus \{r\}$ needs to traverse the arc a. One way to avoid this issue, at the expense of extra variables, is to introduce separate flow variables for each non-root terminal $u \in T \setminus \{r\}$. More precisely, observe that in any directed Steiner Tree $U \subseteq A$, when installing unit capacities on each arc of U, we can send a unit flow from r to any of the other terminals. We highlight that these flows cannot be sent simultaneously, which is why this formulation differs from (flow MIP 1). Hence, instead of one flow vector $f \in \mathbb{R}^A_{\geq 0}$, we introduce for each $u \in T \setminus \{r\}$ a flow vector $f^u \in \mathbb{R}^A_{\geq 0}$, which is required to be an r-u flow of value one. This leads to the following formulation.

$$\min \sum_{a \in A} \ell(a) x(a)$$

$$f^u(\delta^+(v)) - f^u(\delta^-(v)) = \begin{cases} 1 & \text{if } v = r \\ -1 & \text{if } v = u \\ 0 & \text{if } v \in V \setminus \{u, r\} \end{cases} \qquad \forall\, u \in T \setminus \{r\},\, v \in V \tag{flow MIP 2}$$

$$f^u(a) \leq x(a) \qquad \forall\, u \in T \setminus \{r\},\, a \in A$$

$$x \in \{0,1\}^A$$

$$f^u \in \mathbb{R}^A_{\geq 0} \qquad \forall\, u \in T \setminus \{r\}$$

As usual, both (flow MIP 1) and (flow MIP 2) are valid formulations for the directed Steiner Tree Problem. In particular, for any directed Steiner Tree $U \subseteq A$, one can find solutions (x, f) for (flow MIP 1) and $(x, \{f^u\}_{u \in T \setminus \{r\}})$ for (flow MIP 2) such that $x = \chi^U$ is the characteristic vector of U. Conversely, for any solution vector to (flow MIP 1) or (flow MIP 2), the vector $x \in \{0,1\}^A$ of the solution vector is the characteristic vector of an arc set that contains a directed Steiner Tree. The key difference between the formulations (flow MIP 1) and (flow MIP 2) is that their relaxations can have vastly different strengths.

We start by observing that the relaxation of (flow MIP 2) is always at least as strong as the relaxation of (flow MIP 1). To this end, let $Q_1 \subseteq [0,1]^A \times \mathbb{R}^A_{\geq 0}$ be all points (x, f) that are feasible for the natural linear relaxation of (flow MIP 1). Analogously, let $Q_2 \subseteq [0,1]^A \times \mathbb{R}^{(T \setminus \{r\}) \times A}_{\geq 0}$ be the set of all points $(x, \{f^u\}_{u \in T \setminus \{r\}})$ that are feasible for the natural linear relaxation of (flow MIP 2).

Theorem 7.7

$\text{proj}_x(Q_2) \subseteq \text{proj}_x(Q_1)$.

Proof. Let $(x, \{f^u\}_{u \in T \setminus \{r\}}) \in Q_2$. Then one can easily check that by defining $f \in \mathbb{R}^A_{\geq 0}$ to be the sum of the flow vectors f^u over $u \in T \setminus \{r\}$, we obtain a point $(x, f) \in Q_1$. □

Even though (flow MIP 2) often has a strong linear relaxation, it turns out that there are examples where its integrality gap is super-constant (more precisely, it grows with $|T|$). However, if we use (flow MIP 2) to model an undirected Steiner Tree problem, by replacing each edge by two bidirected arcs of same length, the integrality gap is no more than 2. For both the directed Steiner Tree and undirected Steiner Tree problem, determining the precise worst-case integrality gap of (flow MIP 2) is still an open problem.

7.2.3.3 Equivalent formulation in original space

Finally, we present a formulation in original space whose relaxation is the same as $\text{proj}_x(Q_2)$. To avoid extending the space, we must refrain from adding additional flow variables. Instead, we require that the chosen arcs $x \in \{0,1\}^A$ must fulfill that for any $u \in T \setminus \{r\}$ and any r-u cut $S \subseteq V$, there must be at least one arc leaving the cut S. This leads to the following formulation.

$$\begin{aligned} \min \sum_{a \in A} \ell(a) x(a) & \\ x(\delta^+(S)) &\geq 1 \qquad \forall\, S \subseteq V,\, r \in S,\, T \setminus S \neq \emptyset \\ x &\in \{0,1\}^A \end{aligned} \tag{IP 3}$$

Analogous to the previous formulations, let $Q_3 \subseteq [0,1]^A$ be the solutions of the natural linear relaxation of (IP 3), which is obtained by replacing $x \in \{0,1\}^A$ by $x \in [0,1]^A$. We start by observing that the relaxation of (IP 3) indeed corresponds to the one of (flow MIP 2) when projected onto the x-space.

Theorem 7.8

$Q_3 = \mathrm{proj}_x(Q_2)$.

Proof. We start by showing $Q_3 \supseteq \mathrm{proj}_x(Q_2)$. To this end, let $(\overline{x}, \{\overline{f}^u\}_{u \in T \setminus \{r\}}) \in Q_2$. We need to show that $\overline{x}$ fulfills the cut constraints of (IP 3). Hence, let $S \subseteq V$ with $r \in S$ and $T \setminus S \neq \emptyset$. Let $u \in T \setminus S$. Because $\overline{f}^u$ is an r-u flow of value one in G when using $\overline{x}$ as capacities, we have by the weak max-flow min-cut theorem (Theorem 2.7) that any r-u cut must have value at least 1. Thus, $\overline{x}(\delta^+(S)) \geq 1$ as desired.

We now show $Q_3 \subseteq \mathrm{proj}_x(Q_2)$. Hence, let $\overline{x} \in Q_3$, and we need to show that, for each $u \in T \setminus \{r\}$, the graph $G = (V, A)$ with capacities given by $\overline{x}$ admits a r-u flow of value one. Because $\overline{x}$ fulfills the constraints of (IP 3), we have that every r-u cut has value at least one. Hence, in particular, the minimum r-u cut has value at least one. Thus, by the strong max-flow min-cut theorem (Theorem 2.8), the value of a maximum r-u flow is at least one. Consequently, there exists an r-u flow $\overline{f}^u$ of value one for each $u \in T \setminus \{r\}$. □

Even though the relaxations of the formulations (flow MIP 2) and (IP 3) are identical when only considering the x-space, there are crucial differences between the two when using these formulations to solve the directed Steiner Tree problem. The extended formulation (flow MIP 2) can be used directly with an off-the-shelf branch & bound/cut solver. One key disadvantage of it is the higher dimension in which it lives, as it uses $|T| \cdot |A|$ many variables. The formulation (IP 3), which is in original space, has exponentially many constraints, and a complete explicit description of the formulation can therefore not be used in off-the-shelf solvers whenever the problem instance is not extremely small. However, this formulation can be very useful when combined with a tailored cutting plane approach that generates cutting planes whenever needed by solving the separation problem of the relaxation of (flow MIP 2). It is hard to say which approach is better, as this may depend on the particular problem at hand, and potential additional constraints in case the directed Steiner Tree problem only describes part of the conditions of a larger optimization problem.

Moreover, we note that the strength of the relaxations of (IP 3) (and thus also of (flow MIP 2)) can heavily depend on the problem. In particular, if $|T| \in \{2, |V|\}$, which corresponds to the shortest s-t path problem and the minimum weight r-arborescence problem, respectively, the

relaxation is tight, in the sense that its optimal value is the same as the value of an optimal solution to the directed Steiner Tree problem.[2]

Exercise 7.9

Consider the formulation (IP 3) for the case of two terminals $T = \{r, u\}$. Show that, in this case, the optimal value of the relaxation Q_3—which, we recall, is the natural linear relaxation of (IP 3)—is equal to the length of a shortest r-u path.

7.2.4 Spanning trees

Spanning trees are a very common substructure in a variety of network optimization problems. In such contexts, a strong linear inequality description can be very helpful. Whereas spanning trees often appear as one among several types of constraints, we will focus here only on the isolated problem of describing spanning trees. As usual, such a description can be extended with further constraints. Spanning trees are another nice example for extended formulations, because the tightest formulation in original space, i.e., the convex hull of all characteristic vectors of spanning trees, which is also known as the *spanning tree polytope*, has exponentially many facets. Thus, it requires exponentially many linear constraints by Proposition 4.17.

The theorem below contains a well-known description of the spanning tree polytope, which we state without proof. One can show that for a complete graph on at least 2 vertices, each of the inequalities in the description below is facet-defining.

Theorem 7.10: Spanning tree polytope

Let $G = (V, E)$ be an undirected graph. Then the polytope $P_{\mathrm{ST}} \subseteq \mathbb{R}^E$ described below is the convex hull of all characteristic vectors of spanning trees in G, i.e., it is the spanning tree polytope.

$$P_{\mathrm{ST}} = \left\{ x \in \mathbb{R}^E_{\geq 0} \;\middle|\; \begin{array}{ll} x(E) = |V| - 1 & \\ x(E[S]) \leq |S| - 1 & \forall S \subsetneq V,\ |S| \geq 2 \end{array} \right\}$$

Even though showing that the inequality description in Theorem 7.10 indeed describes the spanning tree polytope is non-trivial, it is not hard to verify that the analogous integer formulation for spanning trees, shown below, is correct. More precisely, the integer formulation below allows for finding a minimum weight spanning tree in a graph $G = (V, E)$, where edge weights are given by $\ell \colon E \to \mathbb{R}_{\geq 0}$.

[2] We highlight that, in this case, even for $|T| = 2$, the set Q_3 is not the convex hull of all s-t paths, but of all arc sets that contain an s-t path. However, because arc lengths are non-negative, the optimal objective value is the same when finding a minimum length arc set that contains an s-t path or a minimum length s-t path. Moreover, given an arc set $U \subseteq A$ that contains an s-t path, one can use BFS to find an s-t path $P \subseteq U$ in linear time.

$$\begin{aligned} \min \sum_{e \in E} \ell(e) \cdot x(e) & \\ x(E) &= |V| - 1 \\ x(E[S]) &\leq |S| - 1 \quad \forall S \subsetneq V, |S| \geq 2 \\ x &\in \{0,1\}^E \end{aligned} \tag{7.4}$$

Proposition 7.11

For any graph $G = (V, E)$, the feasible solutions $x \in \{0,1\}^E$ to (7.4) are precisely the characteristic vectors of spanning trees in G.

Proof. We start by discussing that for every spanning tree $T \subseteq E$, its characteristic vector $\chi^\top \in \{0,1\}^E$ is a solution to (7.4). We rely on the characterization of spanning trees given by Theorem 2.2 (v), i.e., a subset of edges $T \subseteq E$ is a spanning tree if and only if $|T| = |V| - 1$ and T does not contain any cycles. Note that an edge set $T \subseteq E$ does not contain any cycle if and only if

$$|E[S] \cap T| \leq |S| - 1 \qquad \forall\, S \subseteq V \text{ with } |S| \geq 2\ . \tag{7.5}$$

Indeed, if an edge set T contains a cycle over some vertex set S, then the constraint in (7.5) corresponding to S is violated. Conversely, if T violates a constraint of (7.5) corresponding to some set $S \subseteq V$ with $|S| \geq 2$, then $T \cap E[S]$ has at least $|S|$ many edges; however, a largest cycle-free set of edges in $E[S]$ is a spanning tree over S, which has only $|S| - 1$ many edges. Thus, T must contain a cycle among the edges in $T \cap E[S]$.

It remains to rephrase the characterization that a set $T \subseteq E$ is a spanning tree if and only if $|T| = |V| - 1$ and $|T|$ fulfills (7.5) in terms of characteristic vectors. This leads to $T \subseteq E$ being a spanning tree if and only if $\chi^\top(E) = |V| - 1$ and $\chi^\top$ fulfills (7.5), as desired. □

One way to obtain a compact, i.e., polynomial-size, extended formulation for spanning trees is by using a flow formulation as we did for the directed Steiner Tree problem. More precisely, as mentioned previously, the relaxation of the directed Steiner Tree formulation (flow MIP 2) is tight when all the vertices are terminals, which leads to the minimum r-arborescence problem. Hence, we can get a compact and tight linear formulation for spanning trees by first replacing each edge $\{u, v\} \in E$ in the undirected graph by two antiparallel arcs (u, v) and (v, u), both with length $\ell(\{u, v\})$, and then using the formulation of (flow MIP 2) for minimum weight r-arborescences. To map things back to the edge space $\mathbb{R}^E$ in which spanning trees are naturally described, one can introduce auxiliary variables $z \in \mathbb{R}^E$ and set $z(\{u, v\}) = x((u, v)) + x((v, u))$. The projection of the natural linear relaxation of this description to the z-space then results in the spanning tree polytope.

Theorem 7.12 provides another compact way to express the spanning tree polytope through an extended formulation. For simplicity, even though we require $x \in \mathbb{R}^E_{\geq 0}$ to live in the edge

space, we also use the notation $x_{\{u,v\}}$ for distinct pairs of vertices $u, v \in V$ if $\{u, v\} \notin E$ and interpret any such term as zero.

Theorem 7.12

Let $G = (V, E)$ be a simple undirected graph. Then the polytope $P \subseteq \mathbb{R}^E$ described below is the spanning tree polytope, i.e., $P = P_{\mathrm{ST}}$.

$$P = \left\{ x \in \mathbb{R}^E_{\geq 0} \;\middle|\; \begin{array}{ll} \exists \text{ values } z_{v,w,u} \in \mathbb{R}_{\geq 0} \text{ for } \{v,w\} \in E \text{ and } u \in V \setminus \{v,w\} \text{ s.t.:} & \\ x_{\{v,w\}} = z_{v,w,u} + z_{w,v,u} & \forall \{v,w\} \in E \\ & \forall u \in V \setminus \{v,w\} \\ x_{\{v,u\}} + \sum_{\substack{w \in V \setminus \{v,u\}: \\ \{v,w\} \in E}} z_{v,w,u} = 1 & \forall v, u \in V, v \neq u \end{array} \right\}$$

Notice that the description of polytope P in Theorem 7.12 is indeed an extended formulation. More precisely, a polyhedron is defined on the variables x and z, whose projection onto the x-space is the spanning tree polytope.

Before providing a formal proof of Theorem 7.12, we give some intuition for the interpretation of variables involved in the description shown in Theorem 7.12. The variables x_e for $e \in E$ can be interpreted as the edges used in the spanning tree, i.e., $x_e = 1$ means that we use edge e in the spanning tree and $x_e = 0$ that we do not. Notice that we have $x_e \in [0, 1]$ because the upper bound follows from the second family of equality constraints in the description of P in Theorem (7.12), as it implies that one can add non-negative values to x_e to obtain 1. The variables $z_{v,w,u}$ for $\{v, w\} \in E$ and $u \in V \setminus \{v, w\}$ have the following interpretation for a given spanning tree $T \subseteq E$. If $\{v, w\} \in T$ and u is in the same connected component as w in the graph $(V, T \setminus \{\{v, w\}\})$, then we set $z_{v,w,u} = 1$, otherwise $z_{v,w,u} = 0$. Finally, if $\{v, w\} \notin T$, then we set $z_{v,w,u} = z_{w,v,u} = 0$.

Notice that, given a spanning tree T, if we set the values of x and z as just explained, then they indeed fulfill the constraints in the description of P due to the following. We start with the first family of equality constraints. For every edge $\{v, w\} \in E$ and $u \in V \setminus \{v, w\}$, we are in one of the following three situations:

(i) $\{v, w\} \notin T$, in which case we have $x_{\{v,w\}} = 0$, $z_{v,w,u} = 0$, and $z_{w,v,u} = 0$;
(ii) $\{v, w\} \in T$ and u is in the connected component of $(V, T \setminus \{\{v, w\}\})$ that contains w, in which case we have $x_{\{v,w\}} = 1$, $z_{v,w,u} = 1$, and $z_{w,v,u} = 0$; or
(iii) $\{v, w\} \in T$ and u is in the connected component of $(V, T \setminus \{\{v, w\}\})$ that contains v, in which case we have $x_{\{v,w\}} = 1$, $z_{v,w,u} = 0$, and $z_{w,v,u} = 1$.

In all three cases, the constraint $x_{\{v,w\}} = z_{v,w,u} + z_{w,v,u}$ is fulfilled.

We now check that the second family of equality constraints in the description of P is satisfied by x and z as defined above. Hence, let $\{v, w\} \in E$. If $\{v, w\} \in T$, then $x_{\{v,w\}} = 1$ and

$z_{v,u,w} = 0$ for all $u \in V \setminus \{v, w\}$. Thus, the constraint

$$x_{\{v,w\}} + \sum_{\substack{u \in V \setminus \{v,w\}: \\ \{v,u\} \in E}} z_{v,u,w} = 1 \tag{7.6}$$

is satisfied. Otherwise, if $\{v, w\} \notin T$, then $x_{\{v,w\}} = 0$ and precisely one of the terms $z_{v,u,w}$ in the sum in (7.6) is equal to one and all other ones are equal to zero. This is the case because the sum term sums over all $u \in V \setminus \{v, w\}$ that are neighbors of v and there is precisely one neighbor that lies on the unique v-w path in T. Only for this neighbor u, we have that w is in the same connected component as u in the graph $(V, T \setminus \{\{v, u\}\})$.

Note that the above discussion shows $P_{\mathrm{ST}} \subseteq P$, i.e., the spanning tree polytope is contained in the polytope P described in Theorem 7.12. Indeed, the vertices of P_{ST} are precisely the characteristic vectors of spanning trees, and, by the above discussion, all of those are contained in P. Hence, to show Theorem 7.12, it remains to show $P \subseteq P_{\mathrm{ST}}$.

Proof of Theorem 7.12. Let $P \subseteq \mathbb{R}^E_{\geq 0}$ be the polytope as described in Theorem 7.12, and let $x \in P$. As discussed, it remains to show $P \subseteq P_{\mathrm{ST}}$. We show this statement by showing that x fulfills all the constraints of the spanning tree polytope P_{ST} as described in Theorem 7.10.

To this end, let $z_{v,w,u} \in \mathbb{R}_{\geq 0}$ for $\{v, w\} \in E$ and $u \in V \setminus \{v, w\}$ be values such that x and z fulfill the constraints of the description of P. Because $x \in P$, such values z do exist. Let $S \subseteq V$ with $|S| \geq 2$. We have

$$\begin{aligned}
(|S| - 2) \cdot x(E[S]) &= (|S| - 2) \cdot \sum_{\{v,w\} \in E[S]} x_{\{v,w\}} \\
&= \sum_{\{v,w\} \in E[S]} \sum_{u \in S \setminus \{v,w\}} (z_{v,w,u} + z_{w,v,u}) \\
&= \sum_{\substack{v,u \in S: \\ v \neq u}} \sum_{\substack{w \in S \setminus \{v,u\}: \\ \{v,w\} \in E}} z_{v,w,u} \\
&\leq \sum_{\substack{v,u \in S: \\ v \neq u}} \sum_{\substack{w \in V \setminus \{v,u\}: \\ \{v,w\} \in E}} z_{v,w,u} \\
&= \sum_{\substack{v,u \in S: \\ v \neq u}} (1 - x_{\{v,u\}}) \\
&= |S|(|S| - 1) - 2x(E[S]) \ ,
\end{aligned} \tag{7.7}$$

where the first equality follows from $x_{\{v,w\}} = z_{v,w,u} + z_{w,v,u}$ for $\{v, w\} \in E$ and $u \in V \setminus \{v, w\}$ because x and z satisfy the constraints in Theorem 7.12, and the inequality follows from (7.6). By rearranging terms, we get $x(E[S]) \leq |S| - 1$ as desired. Finally, for $S = V$, the inequality in (7.7) is satisfied with equality, thus implying $x(E) = |V| - 1$. In summary, x fulfills all constraints of P_{ST} as described in Theorem 7.10. Hence, $P \subseteq P_{\mathrm{ST}}$ as desired. □

7.3 Union of polyhedra

Sometimes the set of solutions to a problem can be described as the union of solutions to a few well-structured sub-problems. In such cases, it is natural to aim at obtaining a strong formulation of the original problem via formulations for the subproblems. Fortunately, this can be achieved for a wide variety of settings, due to the very general result stated below.

Theorem 7.13

Let $k \in \mathbb{Z}_{\geq 1}$ and $n \in \mathbb{Z}_{\geq 1}$. For $i \in [k]$, let $P^i := \{x \in \mathbb{R}^n \colon A^i x \leq b_i\}$, where $A^i \in \mathbb{R}^{m_i \times n}, b^i \in \mathbb{R}^{m_i}$ for some $m_i \in \mathbb{Z}_{\geq 0}$, and let $C^i := \{x \in \mathbb{R}^n \colon A^i x \leq 0\}$. Then, the polyhedron

$$P := \operatorname{conv}\left(\bigcup_{i=1}^{k} P^i\right) + \sum_{i=1}^{k} C^i$$

satisfies $P = \operatorname{proj}_x(Q)$, where

$$Q := \left\{ (x, x^1, x^2, \dots, x^k, \delta) \in \mathbb{R}^n \times (\mathbb{R}^n)^k \times \mathbb{R}^k_{\geq 0} \;\middle|\; \begin{array}{ll} A^i x^i \leq \delta_i b^i & \forall i \in [k] \\ \sum_{i=1}^{k} x^i = x & \\ \sum_{i=1}^{k} \delta_i = 1 & \end{array} \right\} .$$

Proof. We start by showing $P \subseteq \operatorname{proj}_x(Q)$. Hence, let $x \in P$. By definition of P, the point x can be written as

$$x = y + \sum_{i=1}^{k} q^i \ , \tag{7.8}$$

for some $y \in \operatorname{conv}(\bigcup_{i=1}^{k} P^i)$ and $q^i \in C^i$ for $i \in [k]$. Thus, we can write y as follows. There are points $y^{i,j} \in P^i$ for $i \in [k]$ and $j \in [h^i]$ for some $h^i \in \mathbb{Z}_{\geq 0}$, and coefficients $\lambda^{i,j} \in \mathbb{R}_{>0}$ such that

$$\begin{aligned} y &= \sum_{i=1}^{k} \sum_{j=1}^{h^i} \lambda^{i,j} y^{i,j} \ , \text{ and} \\ 1 &= \sum_{i=1}^{k} \sum_{j=1}^{h^i} \lambda^{i,j} \ . \end{aligned} \tag{7.9}$$

We first observe that there always is a way to write y as such a convex combination with $h^i \in \{0, 1\}$ for each $i \in [k]$. In words, in the above convex combination describing y, for each $i \in [k]$, the convex combination needs at most one point in P^i. To this end, let

$$I := \left\{ i \in [k] \colon h^i > 0 \right\} \ ,$$

and we define, for $i \in I$,

$$\lambda^i := \sum_{j=1}^{h^i} \lambda^{i,j} \ , \text{ and} \tag{7.10}$$

$$y^i := \frac{1}{\lambda^i} \sum_{j=1}^{h^i} \lambda^{i,j} y^{i,j} \ .$$

Note that $y^i \in P^i$ because y^i is a convex combination of points in P^i. Thus, we have that y is the following convex combination of $\{y^i\}_{i \in I}$:

$$y = \sum_{i \in I} \lambda^i y^i \ . \tag{7.11}$$

We now finish this part of the proof by defining a point in Q whose projection onto the x-space equals the point x. For $i \in [k]$, we set:

$$x^i := \begin{cases} \lambda^i y^i + q^i & \text{if } i \in I \ , \\ q^i & \text{if } i \in [k] \setminus I \ , \end{cases}$$

and

$$\delta_i := \begin{cases} \lambda^i & \text{if } i \in I \ , \\ 0 & \text{if } i \in [k] \setminus I \ . \end{cases}$$

It remains to observe that $(x, x^1, x^2, \ldots, x^k, \delta) \in Q$. We have

$$\sum_{i=1}^{k} x^i = \sum_{i \in I} \lambda^i y^i + \sum_{i=1}^{k} q^i = y + \sum_{i=1}^{k} q^i = x \ ,$$

where the first equality follows from our definition of the x^i vectors, the second one follows from (7.11), and the last one is implied by (7.8). Moreover, we have

$$\sum_{i=1}^{k} \delta_i = \sum_{i \in I} \lambda^i = \sum_{i \in I} \sum_{j=1}^{h^i} \lambda^{i,j} = \sum_{i=1}^{k} \sum_{j=1}^{h^i} \lambda^{i,j} = 1 \ ,$$

where the first equality is an immediate consequence of our definition of δ_i, the second one follows from (7.10), the third one from the definition of I, and the last one from (7.9).

Finally, we have to check that $A^i x^i \leq \delta_i b^i$ for each $i \in [k]$. If $i \notin I$, then $x^i = q^i$ and $\delta_i = 0$, which implies as desired $A^i x^i = A^i q^i \leq 0 = \delta_i b^i$, where the inequality follows from $q^i \in C^i$. Otherwise, if $i \in I$, we have

$$A^i x^i = A^i(\lambda^i y^i + q^i) = \lambda^i A^i y^i + A^i q^i \leq \lambda^i A^i y^i \leq \lambda^i b^i \ ,$$

where the first inequality follows from $q^i \in C^i$ and the second one from $y^i \in P^i$.

We now show the converse direction, i.e., $\text{proj}_x(Q) \subseteq P$. Hence, let $(x, x^1, x^2, \dots, x^k, \delta) \in Q$, and we have to show that $x \in P$. Let

$$I \coloneqq \{i \in [k] \colon \delta_i > 0\} \ .$$

We start by observing that $\frac{x^i}{\delta_i} \in P^i$ for $i \in I$ because of $A^i x^i \leq \delta_i b^i$ and $\delta_i > 0$. Moreover, because $x = \sum_{i=1}^k x_i$, we have

$$x = \left(\sum_{i \in I} \delta_i \cdot \frac{x^i}{\delta_i} \right) + \sum_{i \in [k] \setminus I} x^i \ .$$

Note that $\sum_{i \in I} \delta_i \cdot (x^i/\delta_i)$ is a convex combination of the points $x^i/\delta_i \in P^i$, and is thus in $\text{conv}(\bigcup_{i=1}^k P^i)$. Hence, to obtain $x \in P$, it suffices to show $x^i \in C^i$ for $i \in [k] \setminus I$. This holds because for $i \in [k] \setminus I$, we have

$$A^i x^i \leq \delta_i b^i = 0 \ ,$$

where the equality follows from $i \in [k] \setminus I$. □

We highlight that even though Theorem 7.13 is stated for polyhedra described with less-than-or-equal constraints, it easily carries over to descriptions using a mix of less-than-or-equal, greater-than-or-equal, and equality constraints. A canonical way to deal with this more general case is to first transform the descriptions to use less-than-or-equal constraints only, and then apply the theorem. After applying the theorem, one can bring the inequalities back to their original form. By shortcutting this line of reasoning, Theorem 7.13 can directly be applied to descriptions with equalities and inequalities of mixed type without first transforming them into less-than-or-equal constraints. The statement as employed in the theorem has the advantage that it simplifies the notation, and its generalization to different constraint types is straightforward.

In the following, we show example applications of Theorem 7.13.

7.3.1 Sets of even size

Let N be a finite set, and assume that we are interested in subsets of N of even cardinality. Formally, the feasible sets of our problem form the family

$$\mathcal{I} \coloneqq \{I \subseteq N \colon |I| \text{ is even}\} \ .$$

Like with other examples, one may not face such an even size constraint in isolation, but rather in combination with other constraints. Even for such settings, it is important to understand how to obtain a strong formulation for even size sets, which can then be combined with other constraints. Hence, for the sake of illustrating Theorem 7.13, we consider here the even size constraint in isolation.

We are interested in obtaining a tight formulation for $\mathcal{I}$, i.e., one that describes the polytope

$$P_{\mathcal{I}} \coloneqq \text{conv}\left(\{\chi^I \colon I \in \mathcal{I}\}\right) \ .$$

Note that $P_{\mathcal{I}}$ has exponentially many facets. More precisely, it corresponds to the hypercube $[0,1]^N$ with the "odd corners" cut off, where an *odd corner* is a vertex $y \in \{0,1\}^N$ with $\|y\|_1$ odd. Actually, one can observe that the number of facets of $P_{\mathcal{I}}$ is equal to $2^{|N|-1}$ (assuming $|N| \geq 3$), where each facet corresponds to an odd corner that got cut off. Hence, despite its simple definition, any formulation of $P_{\mathcal{I}}$ in the original space $\mathbb{R}^N$ requires at least $2^{|N|-1}$ many constraints, due to Corollary 4.19. In the following, we observe how a tight and compact, i.e., polynomial size, description can be obtained in extended space by leveraging Theorem 7.13.

The family $\mathcal{I}$ can naturally be partitioned by grouping sets of same cardinality together. More formally, let $k := \lfloor |N|/2 \rfloor$, and define

$$\mathcal{I}_{2i} := \{I \subseteq N : |I| = 2i\} \quad \forall i \in \{0, \dots, k\} \ ,$$

which partition the family $\mathcal{I}$. Thus

$$\mathcal{I} = \bigcup_{i=0}^{k} \mathcal{I}_{2i} \ . \tag{7.12}$$

For each $\mathcal{I}_{2i}$, it is not hard to get a tight description in original space. Indeed, one can easily verify that, for $i \in \{0, \dots, k\}$,

$$P_{\mathcal{I}_{2i}} := \operatorname{conv}\left(\{\chi^I : I \in \mathcal{I}_{2i}\}\right) = \left\{x \in [0,1]^N : x(N) = 2i\right\} \ .$$

As mentioned, we can apply Theorem 7.13 to formulations with arbitrary linear constraint types ('$\leq$', '$\geq$', and '$=$'). Hence, this leads to the following polytope $Q \in \mathbb{R}^N \times (\mathbb{R}^N)^{k+1} \times \mathbb{R}^{k+1}$ (we note that the occurrence of $k+1$ in the dimension is due to the fact that we described $\mathcal{I}$ as the union of $k+1$ sets):

$$Q := \left\{ (x, x^0, x^1, \dots, x^k, \delta) \in \mathbb{R}^N \times (\mathbb{R}^N)^{k+1} \times \mathbb{R}^{k+1} \;\middle|\; \begin{array}{ll} x^i \geq 0 & \forall i \in \{0, \dots, k\} \\ x^i_j \leq \delta_i & \forall i \in \{0, \dots, k\},\ j \in N \\ x^i(N) = \delta_i \cdot 2i & \forall i \in \{0, \dots, k\} \\ \sum_{i=0}^{k} x^i = x & \\ \sum_{i=0}^{k} \delta_i = 1 & \end{array} \right\} \ .$$

We first observe that Theorem 7.13 indeed guarantees $\operatorname{proj}_x(Q) = P_{\mathcal{I}}$. To this end, note that the descriptions of the polytopes $P_{\mathcal{I}_{2i}}$, which can be written as

$$P_{\mathcal{I}_{2i}} = \left\{x \in \mathbb{R}^N : x \geq 0, x \leq 1, x(N) = 2i\right\} \ ,$$

fulfill

$$C^i := \{x \in \mathbb{R}^N : x \geq 0, x \leq 0, x(N) = 0\} = \{0\} \ .$$

Hence, Theorem 7.13 implies the desired result because

$$\begin{aligned}\operatorname{proj}_x(Q) &= \operatorname{conv}\left(\bigcup_{i=0}^{k} P_{\mathcal{I}_{2i}}\right)\\ &= \operatorname{conv}\left(\bigcup_{i=0}^{k} \operatorname{conv}(\operatorname{vertices}(P_{\mathcal{I}_{2i}}))\right)\\ &= \operatorname{conv}\left(\bigcup_{i=0}^{k} \operatorname{vertices}(P_{\mathcal{I}_{2i}})\right)\\ &= \operatorname{conv}\left(\operatorname{vertices}(P_{\mathcal{I}})\right)\\ &= P_{\mathcal{I}} \ ,\end{aligned}$$

where the second equality follows from Proposition 4.32, the third one is a basic relation that we leave as an exercise, and the penultimate one follows from Theorem 4.32 and (7.12).

Moreover, note that the description of Q is compact. It has only polynomially many variables and constraints. More precisely, the dimension of Q is $|N| + |N|(k+1) + k + 1 = O(|N|^2)$, and the number of constraints is $|N|(k+1) + |N|(k+1) + (k+1) + 1 + 1 = O(|N|^2)$.

7.3.2 Circular interval packings

We now discuss how to obtain a tight extended formulation for circular interval packings. The key observation that we use is that a (non-circular) interval packing has a simple tight description in original space, as we discussed in Section 6.2.3. We start by recalling this description. Hence, consider a (non-circular) interval packing problem. Thus, we are given n (half-open) intervals $[a_i, b_i)$ for $i \in [n]$ with $a_i, b_i \in \mathbb{Z}$ and $a_i < b_i$. Feasible sets are subsets of non-overlapping intervals. We refer to intervals by their indices, and thus describe the family $\mathcal{I} \subseteq 2^{[n]}$ of feasible sets of intervals as subsets of the interval indices $[n]$:

$$\mathcal{I} \coloneqq \{I \subseteq [n] \colon [a_i, b_i) \cap [a_j, b_j) = \emptyset \ \forall i, j \in I, i \neq j\} \ .$$

As mentioned in Section 6.2.3, the following is a tight formulation of the family $\mathcal{I}$:

$$P_{\mathcal{I}} \coloneqq \operatorname{conv}\left(\{\chi^I \colon I \in \mathcal{I}\}\right) = \left\{ x \in \mathbb{R}^n_{\geq 0} \colon \sum_{\substack{j \in [n]\colon \\ a_i \in [a_j, b_j)}} x_j \leq 1 \quad \forall i \in [n] \right\} \ . \tag{7.13}$$

Now assume that we are given an instance of a circular interval packing problem. We first identify a point on the circle that is covered by as few circular intervals as possible. Let us denote this point by y, and assume that the circular intervals covering it are $I_1, \ldots, I_k$. Every circular interval packing contains at most one interval among the intervals $\{I_1, \ldots, I_k\}$. Hence, there are $k+1$ options: either no interval among these intervals gets selected or precisely one. Each of these options reduces to a regular (non-circular) interval packing problem. For example, if no interval covers y, then we can remove the intervals $I_1, \ldots, I_k$ from the instance, and cut the

circle at the point y to obtain a regular interval packing problem. Otherwise, if a single interval I_j, for some $j \in [k]$, covers y, then we can include I_j in the packing and remove all intervals that overlap with I_j. Again, the residual instance obtained that way will simply be a regular interval packing problem because, among the remaining intervals, there is none that covers y. In short, we partitioned the set of all solutions to the circular interval packing problem into $k + 1$ parts, each of which can be described by a description as shown in (7.13). Finally, an application of Theorem 7.13 leads to a compact extended formulation of the tightest description for the circular interval packing problem.

We note that this way of obtaining a tight compact extended formulation for circular interval packings is essentially a polyhedral version of the reduction from circular interval packing to regular interval packing mentioned in Remark 6.6.